AUTOMATED VERIFICATION OF COMPLETE SPECIFICATION WITH SHAPE INFERENCE

LE QUANG LOC

M.Eng. in Computer Science

Ho Chi Minh City University of Technology

A THESIS SUBMITTED

FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE

NATIONAL UNIVERSITY OF SINGAPORE

2014

DECLARATION

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

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

Le Quang Loc

18 August 2014

Acknowledgements

No guide, no realization.

I am deeply grateful to Professor Chin Wei-Ngan, a very conscientious advisor that I could ever ask or even hope for. I was extremely lucky to have worked with Wei-Ngan. Wei-Ngan spent countless hours to listen to my half-baked ideas, to share his thoughts, and to help refine the ideas to attain this thesis. Wei-Ngan's patience, enthusiasm, and encouragement kept me moving. I would like to thank Wei-Ngan for his continuous support and all the things he taught me, on both research and non-research matters, during the last five years.

A big thank you goes to my thesis committee members, Professor Khoo Siau-Cheng, Professor Aquinas Hobor and Dr. Radu Iosif. I sincerely appreciate the interest they all showed and the amount of time that they committed to meeting during the work‘s progression and to reading once this report was submitted. I thank Professor Dong Jin Song for chairing the committee.

I also thank to my collaborators along the way: Cristian Gherghina, Shengchao Qin, Asankhaya Sharma, Florin Craciun, Minh-Thai Trinh, Cristina David, and Razvan Voicu. A special thank you goes to Cristian and Shengchao for their advice, comments, and insightful ideas. I am very grateful to Shengchao for his careful reading of the final revision of this report. Many thanks to Andrey Rybalchenko for his useful comments on the Second-Order Bi-Abduction work, to Duc-Hiep Chu, a very enthusiastic friend, for his advice on how to conduct a good research and to grow scientific research network, and to Quang-Trung Ta, Andreea Costea, Minh Luan Nguyen, Trung Quy Phan, Long H. Pham, Duy-Khanh Le, Ton-Chanh Le and Phuong Nguyen for their constructive feedback on some of our works. For interesting discussions and entertaining moments, I would like to express my gratitude to my friends: Ninh Pham, Truong Khanh Nguyen, Jamilet Serrano, Huu Hai Nguyen, Abhijeet Banerjee, and many more. It is my pleasure to discuss with you about both research topics and life experience.

I gratefully acknowledge School of Computing, which provided me with the financial support and a very nice working environment.

I thank my parents for their great upbringing and support throughout my life. I thank my wife, Hoai-Chau, for the love of my life, for her understanding, patience, and constant support. And to my little son, Sam: we are best friends, forever.

Le Quang Loc

Singapore, August 15, 2014

Table of Contents

Automated Verification of Complete Specification with Shape Inference

Abstract

To achieve the highest Evaluation Assurance Level, mission-critical software components are required to be specified by formal specification and be verified by a proof system [3]. However, existing verification systems focus mostly on good (safe) scenarios of functional properties (nothing bad will happen), while real world programs often contain bad scenarios. To bridge this gap, the thesis presents a solution for specifying, verifying and synthesizing both good and bad scenarios of heap-manipulating programs.

In the first part of this thesis, we present a *complete* specification mechanism that can specify both *good* and *bad* scenarios of program executions. A good execution is one that takes any permitted input and produces the expected output without any errors. A bad execution is one that takes some input but leads to some unexpected error. We present a verification system that supports complete specification. Our proposed system is capable of ensuring good scenarios (from safety proving) and detecting bad scenarios (from errors validation). A key principle of our proposal is a lattice of program status at the logic level, that is used to denote good and bad program states, and a new calculus to support systematic reasoning in the presence of errors.

In the second part of this thesis, we propose to automate verification system with specification inference. In the context of heap-manipulating programs, specification inference captures the analysis of shapes to describe abstractions for data structures used by each method. While previous shape analysis proposals rely on using a predefined vocabulary of shape definitions (typically limited to singly-linked list segments), our approach is able to synthesize, from scratch, a set of shape abstractions that is needed for ensuring memory-safe operations. The key concept behind our novel proposal is a *second-order bi-abduction* mechanism. With bi-abduction, we infer missing information that helps verifiers to either prove memory safety (for the good scenarios) or disprove it (for the bad scenarios). In this second-order mechanism, we use unknown predicates (or second-order variables) as place-holders for shape predicates that are to be synthesized. Our second-order bi-abduction generates missing information as a *set of relational assumptions* on the unknown predicates that are obtained directly from proof obligations gathered by our verification process.

We next propose a transformational approach on each gathered set of relational assumptions. Our approach includes derivation and normalization steps. While the *derivation* infers sound definition for each unknown predicate, the *normalization* step further simplifies those definitions into a more concise, understandable and re-usable predicate form.

We have implemented the proposals in a prototype system and evaluated them by using the system to specify, verify, and synthesize specifications of programs with complex data structures. The experimental results demonstrate the viability of our proposals in inferring memory-safe specification and the verification of programs with complete specifications.

Keywords: Second-Order Bi-Abduction, Specification Inference, Complete Specification, Shape Analysis, Shape Synthesis, Separation Logic.

Thesis Advisor: Associate Professor Chin Wei-Ngan, Computer Science Department, SoC-NUS.

List of Tables

List of Figures

Chapter 1

Introduction

Reliable software, especially safety critical systems found in aeronautics, avionics and banking, should meet safety requirements that conform to regulation standards [53]. To uphold these standards, the software should be verified by automatic software verification systems. Software verification is a long-standing and important problem. Recently, software verification has received much attention with a number of commercially viable systems, such as Infer [22] at Facebook, Astree [15] at Airbus, Codesonar [75] at GammaTech, Dafny [96] and Slayer [13] at Microsoft and Parfait [34] at Oracle.

Software verification is the art of using formal mathematics to prove or disprove the correctness of a given program with respect to certain formal specifications. Software verification can be classified into two major flavors: static analysis and deductive verification. *Static analysis* automatically computes properties about the behavior of a program without (or with little) users' guidance. An important foundation of static analysis is the *abstract interpretation* framework proposed by Cousot and Cousot [39], which is a framework for sound and terminating analyses based on partially ordered set and fixpoint computation. Static analysis can be fully automatic and scalable. However, it is typically not very expressive; as it is designed to work on a predefined set of properties over a fixed set of abstract domains. In the literature, static analysis has been studied to compute reachability property [34], points-to property [71], shape of pointers [128], termination [37], and so on. This technique has also been used to prove the absence of some classes of errors, such as division-by-zero [59], out of bound [40], and memory errors (e.g. null dereference and leaks) [13, 22]. The techniques have been well studied over several abstract domains such as linear equalities [84], linear congruences [63], octagons [109], octahedra [35], polyhedron [41], and string manipulations [56].

Deductive verification is the art of generating mathematical proof obligations from program and its annotated specification, based on a set of deduction rules. The truth of those obligations guarantees the conformance of the program to its specification. The obligations are discharged by either automatic theorem provers (e.g. Omega [125] and Mona [85]), or satisfiability modulo theories (SMT) solvers (e.g. Z3 [45]). Design by Contract [108] is a good representative of deductive verification. It provides a good design for deductive verification systems and requires software designers to specify requirement formally and have method's correctness checked by an automatic proof system. Deductive verification approach is quite expressive since the properties that need to be analyzed are not hard-wired. Instead, they are flexible and are meant to be guided by user-provided specifications.

The main disadvantage of the deductive approach is that it typically requires users to understand the targeted software in detail and to manually provide specifications for each software component or method. However, writing specifications is typically avoided by developers [117]. This is mostly due to the high cost and time consuming nature of writing and maintaining up-to-date specifications. For new and especially legacy systems, it may be too much work to write functional specifications for every method. Even when a system has been developed with a set of written specifications, software maintenance efforts may require each affected specification to be refined to reflect its improved functionality. Automating or semi-automating the specification writing and maintaining processes would be much desired.

As a solution for automating deductive verification, *specification inference* is a technique that uses static analysis to synthesize specifications in order to guarantee the absence of some kinds of errors [27, 40]. In the context of heap manipulating programs, specification inference relies on capabilities of shape analysis. Given a program, *shape analysis* infers shapes of pointers at program locations that are required for memory safety. For recursive methods, existing shape analyses typically require shape annotations on inputs and outputs. The past decade has seen rapid development of shape analyses in automatic verification systems. Based on abstraction domains, the analysis on shape can be divided into three major groups: (1) three-valued Logic (TVLA) [81, 133], (2) graph types [86, 110, 85], and (3) Separation Logic [9, 20, 57, 73, 130]. TVLA, pioneered by Sagiv, Reps and Wihhelm, is one of the earliest shape analysis framework which used very generic and powerful abstractions based on three-valued logic. Graph types together with pointer assertion logic, invented by Moeller and Schwartzbach, provides a highly expressive mechanism to specify and verify invariants of complex data structures. Separation logic, proposed by O'Hearn and Reynolds [115, 116], has been recently established as an excellent abstraction to reason on heap-manipulating programs. Shape analysis on separation logic can efficiently handle a wide range of data structures, from simply-linked data structures (variants of lists and trees [9, 20]) to complex nested data structures [68, 73], and can be extended to handle pure properties [14, 28, 70, 102, 105, 107, 135].

Although specification language and automatic verification have been well studied, it is still far from the expectation of the software community. We shall discuss several challenges that are faced by software verification systems next.

1.1 Challenges of Automated Verification Systems

As shown in the previous section, three main components of a deductive verification system are specification, verification, and inference. In this section, we highlight limitations of existing systems over these components towards completeness and automation. We also outline two key challenges from dealing with these limitations.

(1) Specifying and Reasoning about Errors.

Although there are numerous specification and verification systems, existing systems focus on expressing good (safe) scenarios of functional properties and missing out on potential bad scenarios (errors) since they use the idealistic assumption that analyzed programs should be safe. However, real world programs often contain errors. For example, methods of Linux kernel Application Programming Interface (API) contain both safety and errors. They typically return outputs with explicit status through numbers, non-negative for safety and negative for errors. For reasoning on errors, there are static analyses, like [67, 87], that detect bugs on handling those returns of the Linux kernel-level and OpenSSL code. In the deductive verification approach, there are verification systems, like those based on JML [19] and Spec# [8], that attempt to indirectly specify and verify bad scenarios via exception mechanism. However those exception-based approaches are neither general nor effective. They currently handled bad scenarios at the program level that are supported by program verifiers, but they have not been integrated into entailment procedures. Hence, they can neither handle sophisticated errors that arise from entailment checks, nor support error explanation, nor capture dead code, nor handle non-terminated loops. Designing and implementing a *specification* and modular *verification* for both good and bad scenarios are important and represent the first step towards handling real world programs.

(2) Inferring Specification of Heap-Manipulating Programs.

Specification inference of heap-based programs relies on shape analysis. Current shape analysis mechanisms typically infer specifications for memory safety with a predetermined set of shape predicates [13, 20, 28, 105]. However, discovering arbitrary shape abstractions can be rather challenging, as linked data structures span a wide variety of forms, from singly-linked lists, doubly-linked lists, circular lists, to tree-like data structures. Furthermore, such abstractions would also need to cater to various specializations, such as strictly non-empty structures or segmented structures (e.g. list/tree segments) with outward pointing references. It is interesting and challenging to develop a mechanism from first principle that would be capable of inferring complicated shape specifications, from scratch, directly from heap-manipulating programs. We shall show how this can be done in this thesis.

1.2 My Thesis

This thesis proposes solutions to overcome the above challenges for automated deductive verification systems. The thesis has been developed in the context of a specification and verification system for heap-manipulating programs. Our technical starting point is a semi-automatic verification system presented in [33, 114] where users provide formal specification for each method with the correctness of each method certified by an automatic verifier. On dealing with error scenarios, we propose a novel mechanism towards complete specification and verification. On automated inference, we first describe a principled shape analysis as a first step towards the discovery of shape specifications that can be used by our automated verification system. After that, we present a transformational approach to the inferred shape predicates to obtain concise and usable specifications.

Towards Complete Specification.

We propose a stronger specification language for expressing functional requirements. Regarding *complete specifications*, while authors in [123] aim to express all properties of class invariants in *good* postconditions, our approach is a complement to theirs; as we aim to express both *good* and *potential bad* scenarios in preconditions. Furthermore, we shall provide a verification system to support this new specification mechanism.

In order to specify and verify programs with both good and bad scenarios, we will introduce new notations at the logic level that are used to distinguish good and bad program states. We will also provide a calculus to determine program states during verification. We will show how to integrate the calculus into a separation logic entailment procedure and extend it to verify heap-manipulating programs and to support error explanation.

Towards Specification Inference.

We propose a solution for specification inference that can support a wide range of programs that manipulate complex data structures. Our core proposal is an entailment procedure with second-order bi-abduction mechanism used within a modular verification framework that can support shape abstraction discovery. With second-order feature, we introduce an entailment procedure that can support unknown predicates using second-order variables as place-holders. Through bi-abduction, we incorporate capability of abduction and frame inference into the entailment procedure. The abduction capability helps our procedure to infer missing information of antecedent in order to either prove or disprove entailment. The frame inference capability helps the entailment procedure discover part of antecedent which is not required in consequent of the current entailment. Furthermore, such frame inference capability is critical to support modular verification systems that are expected to work on a per method basis.

More concretely, we propose an entailment procedure that can generate missing information as a set of relational assumptions over the unknown predicates to either prove (i.e. in inferring specification of good scenarios) or disprove (i.e. in inferring specification of bad scenarios) proof obligations. We also propose a modular verifier that accepts the unknown predicates in program states, generates proof obligations for memory safety, invokes the above entailment procedure to discharge the obligations, and accumulates the set of relational assumptions over the unknown predicates. For soundness, the truth of each set of relational assumptions inferred can guarantee the conformance of input program to the correctness of its memory safety proof.

Our proposed entailment mechanism works with pointer-based programs to support inference of shape specifications that ensures memory safety. This yields a novel approach to shape analysis that works on arbitrary data structures and provides direct support for recursive procedures. We present a bi-abductive entailment procedure in separation logic that supports *unknown* shape predicates. A key part of our proposal is the capability for generating a set of relational assumptions over the unknown predicates. These assumptions are then refined into predicate definitions, by a follow-up predicate derivation and normalization steps.

Using abduction for inference is not new, as it was deployed in [48, 61] to generate missing preconditions and in [49] to infer inductive invariants. However, those proposals were limited to numerical domains. In the shape domain, bi-abduction was described in [20] for generating missing assumptions in a modular shape analysis algorithm. However, this algorithm uses a fixed set of shape predicates based on variants of list data structure. In contrast, we propose second-order variables to support arbitrary shape predicates. Thus, our proposal propels automated verification systems to a higher level of both automation and expressiveness. The closest to our proposal is a shape analysis presented in [16]. This analysis proposes a novel way to synthesize inductive predicates by ensuring both memory safety and termination. Unlike ours, this proposal is based on cyclic proving mechanism and is currently limited to a simple imperative language with only loops but not methods.

Transformational Approach to Shape Predicates.

Shape analysis, which naively follows the structure of programs, may produce predicates that are overtly complex. As an intermediate output of shape analysis, the inferred set of relational assumptions, is not immediately usable by automated verification systems. We proceed to derive definition for each unknown predicate and further normalize these definitions into more concise and re-usable form. Our design considers soundness and usability. For soundness, the derivation should distinguish shape predicates in pre-conditions from those in post-conditions; since the former may be safely strengthened, while the latter may only be safely weakened. For usability, the normalization should transform inferred shape predicates into a fragment whose expressiveness is as close as possible to the capability of existing verification systems.

Our fragment of shape predicates was adapted by those presented in [33, 76, 114]. This fragment requires all predicate parameters to be involved in the predicate definition, and each predicate to have a single root pointer. As such, we shall syntactically detect the violation of the above form and provide a semantic-based mechanism for its normalization.

1.3 Contributions

This thesis makes three technical contributions.

Complete Specification with an Error Calculus.

We present basic mechanisms that could be used to support the verification of *complete* specifications. These can be used to uniformly specify and verify both safe and unsafe execution scenarios. Our key research contributions are:

- We propose a novel calculus, based on a four-point lattice domain, for verifying safety and/or the absence of must/may errors.
- We extend this calculus to support concise error explanation that gives priority to must errors.
- We design a specification mechanism for error-based scenarios
- We provide an implementation of the error calculus in separation logic with support for user-defined predicates and lemmas, so as to support verification for functional correctness with error validation.

Shape Analysis via Second-Order Bi-Abduction.

We propose a shape analysis via the second-order bi-abductive mechanism. We make the following contributions.

- We design a novel entailment procedure in separation logic to support inference via bi-abduction which uses a combination of abduction and frame inference. This procedure performs abduction to infer missing information in antecedent that is required for the validity of entailment. It also infers residual heaps that are not needed for the entailment to hold. More concretely, this entailment supports unknown shape predicates (second-order variables) and builds relational assumptions (over the shape predicates) that are required for the validity of entailment. We also present two novel features, guarded context and a scheme for instantiation, that are used to guide this bi-abduction mechanism.

- We develop a *sound* and *modular* shape analysis that is applied on a per method basis. Most existing shape analyses require global analyses or re-verification, as they are unable to directly infer memory-safe (or sound) heap preconditions. For example, bi-abduction in [20] requires its method's inferred pre-condition to be re-verified due to the use of over-approximation on heap pre-condition.

- We provide an implementation of the second-order bi-abduction mechanism within a modular shape analysis.

Transformational Approach to Shape Predicate.

We present an approach to deriving and normalizing shape predicates from a set of relational assumptions. Our technical contribution includes:

- We propose a set of sound derivation rules for *solving* each set of relational assumptions. This helps to derive suitable definition for each unknown shape predicate.

- We describe a set of normalization operations to transform predicate definitions into simplified and re-usable form. Those operations include (1) detecting and eliminating dangling predicate, (2) detecting and eliminating useless parameters, (3) predicate splitting, and (4) predicate reuse. The first operation detects unaccessed pointers through the identification of dangling predicates. The useless parameter elimination operation removes unused parameters of predicates. The splitting operation decomposes complex predicates into multiple simplier predicates. The reuse operation semantically matches inferred shape predicates with existing predicates. These operations will help reduce the complexity of predicates and can enhance the usability for automated verification system.

- We give a preliminary discussion on inferring complete shape specification.

- We provide an implementation and experiments on *shape inference*, that has been systematically integrated into an existing automated verification system.

1.4 Outline of the Thesis

The rest of this thesis is organized as follows.

- Chapter 2 gives background information that forms the basis of our research. It introduces literature review, specification language, entailment procedure, and a motivating example.

- Chapter 3 presents a novel specification mechanism that forms the basis for a complete verification system. The main contribution of this chapter is a lattice domain with four status values that are combined with program states.

- Chapter 4 proposes a mechanism for shape analysis. The main contribution of this chapter is a novel second-order bi-abductive entailment procedure of separation logic. This entailment takes antecedent and consequent as inputs and produces residues states and a set of relational assumptions.

- Derivation and normalization approaches to shape predicates are introduced in Chapter 5. The main contribution of this chapter are sets of rules and an algorithm to derive sound but concise and usable shape predicates.

- Chapter 6 concludes the thesis with a summary of our research achievements and also discusses future works.

Chapter 2

Preliminaries

First, we review several known automatic verification systems. After that, we describe a specification language and entailment procedure used in this thesis. Finally, we illustrate our contributions through a motivating example.

2.1 Existing Verification System

2.1.1 Specification Language

Formal specification languages at the method level have been well studied. There are several well known specification systems, such as Java Modeling Language (JML) [19], Spec# [8], Larch/C++ [93], Alloy [79], and Vienna Development Method (VDM) [4, 82]. Those specification systems provide notations for formally specifying behaviours and interfaces of methods. Their syntax can express safety scenarios with normal and exception-oriented pre-condition/post-condition, object-oriented features (modifiers, visibility, inheritance), frame and case specifications. In the following, we discuss in detail JML [19] and Spec# [8] specification systems.

JML.

JML [19] is a specification language used to specify interfaces and behaviors of Java programs. JML is a comprehensive modelling language. It provides notations for standard pre- and post-conditions, frame conditions (with *Assignable* clause), both normal execution (with *normal_behavior* clause) and abnormal execution (with *exceptional_behavior* clause and *ensure false*), and multiple specification cases. However, exceptions are not technically the same as errors since the former may be handled but not the latter. Besides, JML provides *pure method* that helps to leverage on its underlying programming language. While this mechanism is powerful, it is not totally side-effect free since new heap nodes may be allocated by such pure functions. We note that such pure methods are not classified as pure formula in the domain of separation logic.

Spec#.

Spec# [8] is a specification language that is built on top of the Boogie automatic program verifier. Spec# specification language provides notations to specify standard pre- and post-conditions, exceptions and constraints on data fields of objects for C# programs. In particular, Spec# presents a hierarchical design on exceptional specifications towards modular reasoning. For example, exceptional specifications are categorized according to preconditions proving (*client failures*) and postconditions proving (*provider failures*). Like JML, it also provides programmers with a mechanism to declare classes of exceptions as either *checked* or *unchecked*. Spec# supports the *otherwise* keyword to capture the rest of input domain [6]. However, this notation was mainly used to denote unchecked exceptions (rather than complete preconditions).

2.1.2 Automatic Verification System

Recently, research in verification has achieved several important milestones. Verification systems can automatically verify large and real-world source code, such as Linux kernel (Forester [57, 73]), and Windows drivers (Slayer [9, 13]). They can also support various programming languages (C [33, 36], Java [29] and C# [40]), handle a large range of input programs (such as complex data structures [13, 33], and concurrency - VCC [36]), and targeted at a large range of defects (type error [101], null dereference [13, 33, 34, 74], functional correctness violation [13, 33, 96], and deadlocks [88] without running the program).

In the following, we discuss three verification systems that are capable of reasoning about heap-based programs.

Dafny.

Dafny [99] is an automatic program verification that can be used to verify functional correctness of heap-manipulating programs. It includes a specification language which is based on JML [19] and Spec# [8], and a program verifier which supports pointer-based programs. The specification language consists of standard pre- and post-conditions, (explicit) framing constructs and terminating metrics. Especially, Dafny specification language supports ”ghost” mathematical functions (like pure methods in JML and Spec#). These functions use the same syntax as its programming language and thus can be deployed for both verification and program code. Furthermore, the functions can be used to construct concise and modular pre-, post-conditions and assertions.

Dafny follows the approach of modular verification and relies on Boogie system [7] to verify programs. It does this even to establish proof of lemma which is encoded as a kind of method verification. Dafny transforms input program into the Boogie intermediate verification language. The soundness of Dafny verifier is reduced to the soundness of the Boogie verification system. Dafty system can be used to specify and verify some challenging algorithms, including Schorr-Waite algorithm [99].

Dafny system is still actively being developed and is a good tool for ensuring reliable software. Recently, the system has been extended with two important and challenging features: induction [97] and co-induction [98]. These new features enhance the expressiveness of Dafny verification system.

Smallfoot.

Smallfoot [12] is one of the first verification system based on separation logic. It was incrementally developed based on separation logic [115] with strong semantic foundations [10, 11, 20, 25] and an evolution of practical tools [9, 12, 13, 23, 24, 50, 138]. Smallfoot verification system consists of three key components: specification language, proof obligation generation and decision procedure. Specification language of Smallfoot is based on a practical and decidable fragment of separation logic with spatial conjunction predicate ($*$), points-to predicate ($\mapsto$), and list segment predicate [10]. The decision procedure of Smallfoot has been proven to be both sound and complete, and can infer residual heap of entailment check [12]. Smallfoot analyses program based on symbolic execution paradigm and generates proof obligations for modular reasoning that is potentially scalable [11].

For better automation, Smallfoot was latter extended with some techniques on shape analysis over the above fragment [50]. This shape analysis infers heap-based invariants on program pointers that guarantee the absence of memory errors. The same shape analysis was further extended to the abstract domain with pointer arithmetic [23]. Later, its abstraction operation was improved to provide better scalability [9, 138]. Finally, to fully support modular shape analysis, it was integrated with abduction to obtain a combined mechanism, called bi-abduction [20]. The scalability of this technique was confirmed by the experimental results in [57]. Recently, there have been several important improvements to this fragment. For example, decision procedure via graph technique [38, 69], decision procedure via superposition [118, 119], and

GRASS reduction [120].

Smallfoot is not only an excellent verification platform for reasoning with complex heap-based programs, as it has pioneered a new research direction on the use of separation logic.

HIP/SLEEK.

HIP/SLEEK [33, 114] is a deductive verification system in separation logic. It consists of a specification language, the entailment procedure SLEEK and the modular verifier HIP.

HIP/SLEEK introduced an expressive specification language. This is one of the first automated verification system that directly reasons with *user-defined predicates* in separation logic. This system also supported separation logic reasoning with non-heap pure domains; HIP/SLEEK proposed a fragment of separation logic that combined standard heap features with pure constraints on Presburger arithmetic, polynomial real arithmetic, and monadic bag/set domains. This combined domain was beyond the (dis)equality domains used by prior work [12, 118]. The specification language was enhanced (i) to be more complete with multiple pre- and post-conditions [30], and (ii) to be even more concise, precise and efficient with case specification [60] and immutability annotation [44].

SLEEK is one of the first entailment proving procedures for separation logic with frame inference capability. For entailment checking of inductive shape predicates, SLEEK introduced a procedure based on *unfolding and folding* operations. [1] The entailment check proves that (i) all matching models of the antecedent would be subsumed by models of the consequent; and (ii) irrelevant part of the antecedent will be inferred as residual frame. Firstly, the matching of heap part is performed until heap in the consequent is empty. After that, the

[1]More detail about SLEEK entailment procedure will presented in section 2.3.

entailment in separation logic is reduced (or approximated) to a sound implication in pure logic. Finally, the implication of the pure part is checked semantically through external SMT solvers and theorem provers. For efficiency, a technique for pruning unfeasible disjuncts to enhance the unfolding on inductive predicates was proposed [32].

SLEEK was also one of the first system to make extensible use of lemma mechanism [113], a semi-automatic mechanism for induction proving in separation logic. This mechanism allows users to declare lemmas manually and SLEEK will apply those lemmas automatically during proof search. Lemmas may be used to relate abstractions, i.e. relate different predicates so as to provide more comprehensive reasoning. These lemmas are also considered as induction assumptions and are automatically deployed to support inductive proofs. The automation of induction proving, without explicitly supplied lemmas, was later proposed through the cyclic proving mechanism [17].

HIP is a modular verifier. It transforms imperative program based on symbolic execution and automatically generates sound proof obligations for checking correctness of the input program against user-provided specifications. In turn, those obligations are discharged by the SLEEK entailment procedure. Beside a core imperative language [114], HIP was also extended to object oriented language [31].

Recently, the fragment of separation logic with user-defined predicates has been the focus of active research. There are many new emerging studies, both theorically and practically, on the logic fragment, including issue of completeness of the fragment [134], techniques based on cyclic proof [17, 18], DRYAD [126], GRASS approach [120, 121, 122], and techniques based on automata [76, 77].

This thesis aims to enhance the HIP/SLEEK system to an automated verification system for *complete* specification. First, HIP/SLEEK system will be

$$\begin{array}{lll}
\text{Disj. formula} & \Phi & ::= \Delta \mid \Phi_1 \vee \Phi_2 \\
\text{Formula} & \Delta & ::= \exists \bar{v}\cdot(\kappa \wedge \pi) \\
\text{Spatial formula} & \kappa & ::= \texttt{emp} \mid x{\mapsto}c(f_i : v_i) \mid \texttt{P}(\bar{v}) \mid \kappa_1{*}\kappa_2 \\
\text{Pure formula} & \pi & ::= b \mid \alpha \mid i \mid \varphi \mid \neg\alpha \mid \pi_1 \wedge \pi_2 \\
\textit{Boolean formula} & b & ::= \texttt{true} \mid \texttt{false} \mid v \mid b_1 = b_2 \\
\text{Ptr (Dis)Equality} & \alpha & ::= v_1{=}v_2 \mid v{=}\texttt{NULL} \mid v_1{\neq}v_2 \mid v{\neq}\texttt{NULL} \\
\textit{Linear arithmetic} & i & ::= a_1{=}a_2 \mid a_1{\leq}a_2 \\
 & a & ::= k^{\texttt{int}} \mid v \mid k^{\texttt{int}}{\times}a \mid a_1{+}a_2 \mid -a \\
 & & \quad \mid \mathit{max}(a_1,a_2) \mid \min(a_1,a_2) \\
\textit{Bag constraint} & \varphi & ::= v{\in}\texttt{B} \mid \texttt{B}_1{=}\texttt{B}_2 \mid \texttt{B}_1{\sqsubset}\texttt{B}_2 \\
 & \texttt{B} & ::= \texttt{B}_1{\sqcup}\texttt{B}_2 \mid \texttt{B}_1{\sqcap}\texttt{B}_2 \mid \texttt{B}_1{-}\texttt{B}_2 \mid \{\} \mid \{v\}
\end{array}$$

$$\texttt{P} \in \textbf{Pred} \quad c \in \mathit{Node} \quad f_i \in \mathit{Fields} \quad v, v_i, x, y \in \mathit{Var} \quad \bar{v} \equiv v_1 \ldots v_n$$

Figure 2-1: Fragment of Separation Logic

supported with a complete specification mechanism to capture both good and bad scenarios (see [91] and Chapter 3). After that the system will be empowered with second-order bi-abduction for heap-based specification inference (see [90], Chapter 4 and Chapter 5).

2.2 Specification Language

Syntax. Our specification language is based on separation logic [78, 127]. We restrict our interest to a practical fragment of separation logic with spatial conjunction operator ($*$), points-to predicate ($\mapsto$), and user-defined predicate [114]. Currently, our system does not support the separating implication operator ($-*$) since it is based on a forward reasoning system which does not usually require this operator. Note that $-*$ has been mainly used to express the weakest preconditions for backward reasoning systems [78, 115]. We have thus omitted $-*$ for simplicity.

The fragment of separation logic used in this thesis is presented in Figure 2-1. A formula (symbolic heap) Δ consists of spatial formula and pure formula.

Separation logic introduces two core features: spatial conjunction ($*$) predicate to express two disjoint heap regions; points-to ($\mapsto$) predicate to express a heap with one memory cell. The points-to predicate $x{\mapsto}c(f_i : v_i)$ asserts that x points to an object of data type `c` with fields $\texttt{f}_\texttt{i}$ and their downstream pointers $\texttt{v}_\texttt{i}$. Each `C` data structure has a corresponding points-to predicate that expresses an allocated object. Furthermore, the logic also supports *user-defined* predicates $\texttt{P}(\bar{v})$ which denotes a set of (unbounded) objects. Those predicates help to concisely express complex heap-based data structures. Pure formula is in the form of first-order logic of a combination of (dis)equality α (on pointers), linear arithmetic i and bag φ domains. Note that $v_1 \neq v_2$ and $v \neq \texttt{NULL}$ are just short forms for $\neg(v_1 = v_2)$ and $\neg(v = \texttt{NULL})$, respectively. To express different scenarios for shape predicates, the fragment supports disjunction Φ over formulas.

Semantics Concrete heap models assume a fixed finite collection *Node*, a fixed finite collection *Fields*, a disjoint set *Loc* of locations (heap addresses), a set of non-address values *Val*, with $\texttt{NULL} \in \mathit{Val}$ and $\mathit{Val} \cap \mathit{Loc} = \emptyset$. With this, we define:

$$\begin{array}{lcl} \mathit{Heaps} & \overset{\text{def}}{=} & \mathit{Loc} \rightharpoonup_{fin} (\mathit{Node} \rightarrow \mathit{Fields} \rightarrow \mathit{Val} \cup \mathit{Loc})\} \\ \mathit{Stacks} & \overset{\text{def}}{=} & \mathit{Var} \rightarrow \mathit{Val} \cup \mathit{Loc} \end{array}$$

where $dom(f)$ returns the domain of function f. e is the empty heap that is undefined everywhere.

In our system, pure domains include integer domain (`Ints`), bag of *Val* (2^{Val}), and boolean. The evaluation for pure expressions are determined by valuations as follows:

$$s(a) \in \texttt{Ints} \qquad s(\texttt{B}) \in 2^{\mathit{Val}} \qquad s(b) \in \{\texttt{true}, \texttt{false}\}$$

The semantics is given by a forcing relation: $s, h \models \Phi$ that forces the stack

s and heap h to satisfy the constraint Φ where $h \in \mathit{Heaps}$, $s \in \mathit{Stacks}$, and Φ is a separation logic formula.

The semantics is presented as in Figure 2-2.

$s \models \pi_1 \wedge \pi_2$	`iff`	$s \models \pi_1$ and $s \models \pi_2$
$s \models v_1 \oslash v_2$	`iff`	$\models s(v_1) \oslash s(v_2)$, where $\oslash \in \{=, \neq\}$
$s \models a_1 \oslash a_2$	`iff`	$\models s(a_1) \oslash s(a_2)$, where $\oslash \in \{=, \leq\}$
$s \models \mathtt{B}_1 \oslash \mathtt{B}_2$	`iff`	$\models s(\mathtt{B}_1) \oslash s(\mathtt{B}_2)$, where $\oslash \in \{\in, =, \sqsubset, \sqcup, \sqcap, -\}$
$s, h \models \mathtt{emp}$	`iff`	$h = e$
$s, h \models v{\mapsto}c(f_i : v_i)$	`iff`	$l{=}s(v), \mathrm{dom}(h){=}\{l \rightarrow r\}$ and $r(c, f_i){=}s(v_i)$
$s, h \models \mathtt{p}(\bar{v})$	`iff`	$(s(\bar{v}), h) \in [\![\mathtt{p}(\bar{v})]\!]$
$s, h \models \kappa_1 * \kappa_2$	`iff`	$\exists h_1, h_2 \cdot h_1 \# h_2$ and $h{=}h_1{\cdot}h_2$ and
		$s, h_1 \models \kappa_1$ and $s, h_2 \models \kappa_2$
$s, h \models \mathtt{true}$	`iff`	always
$s, h \models \mathtt{false}$	`iff`	never
$s, h \models \exists v_1, ..., v_n{\cdot}(\kappa \wedge \pi)$	`iff`	$\exists \alpha_1 ... \alpha_n \cdot s(v_1{\mapsto}\alpha_1 * ... * v_n{\mapsto}\alpha_n), h \models \kappa$
		and $s(v_1{\mapsto}\alpha_1 * ... * v_n{\mapsto}\alpha_n) \models \pi$
$s, h \models \neg\Phi$	`iff`	$s, h \not\models \Phi$
$s, h \models \Phi_1 \vee \Phi_2$	`iff`	$s, h \models \Phi_1$ or $s, h \models \Phi_2$

Figure 2-2: Semantics of Specification Language

As pure formula is independent from heap, semantics of pure formula only depends on stack valuations. The model relation for pure formula $s \models \pi$ denotes that the formula π evaluates to true in s.

Note that $h_1 \# h_2$ denotes that heaps h_1 and h_2 are disjoint, i.e. $\mathrm{dom}(h_1) \cap \mathrm{dom}(h_2) = \emptyset$; $h_1 \cdot h_2$ denotes the union of two disjoint heaps. *emp* asserts that h is empty. With points-to predicate $v{\mapsto}c(f_i : v_i)$, h is a singleton heap function. Set of models of a shape predicate $\mathtt{p}(\bar{v})$ is interpreted as its least fixpoint set [18].

2.2.1 User-Defined Predicate

Definition 1 (Shape Predicate)**.** *A shape predicate* $\mathtt{P}$ *is defined as*

$$\mathtt{P}(\bar{v}) \;\equiv\; \bigvee_{i=1}^{n} (\exists \bar{w}_i \cdot \; \Delta_i) \quad \mathit{inv}\text{: } \pi;$$

where

- `P` *is predicate name.*
- $\bar{v}$ *is a set of formal parameters including pointers to heap and pure parameters for expressing structural properties (size, bag of values).*
- $\bigvee_{i=1}^{n} \exists \bar{w}_i \cdot\ \Delta_i = (\exists \bar{w}_1 \cdot\ \Delta_1) \vee ... \vee (\exists \bar{w}_n \cdot\ \Delta_n)$ *is a definition.* $\exists \bar{w}_i \cdot\ \Delta_i$ *(i* $\in$ *1...n) is a branch of the disjunction.*
- π *is* predicate invariant. π *expresses superset of all possible models of* `P` *via a pure constraints on stack.*

Predicate invariants are over-approximation and are used in checking entailment among formulas. Users can choose not to supply predicate invariants as our systems can infer those automatically too.

Branches containing (mutually) recursive user-defined predicates are called recursive branches. Otherwise, they are base branches.

Definition 2 (Root Parameter)**.** *Given shape predicate* `P` *with the following definition:*

$$\mathtt{P}(\bar{v}) \ \equiv\ \bigvee_{i=1}^{n}(\exists \bar{w}_i \cdot\ r_{i1}{\mapsto}c(\bar{v}_{i1}) *...* r_{ik}{\mapsto}c(\bar{v}_{ik}) * \mathtt{P_1}(\bar{w}_{i1}) *...* \mathtt{P_j}(\bar{w}_{ij}) \wedge \pi_i) \quad \mathit{inv}\text{: } \pi;$$

A parameter r $\in \bar{v}$ *is a root if for all i from 1 to n, one of following four conditions holds:*

- *r points-to an allocated heap:* $r \ \in\ \{r_{i1}, ..., r_{ik}\}$.
- *r equals to* `NULL`*:* π_i *contains r=*`NULL` *formula.*
- *r equals to another parameter:* π_i *contains r=s formula, where* $s \in \bar{v}$.
- *r is a root parameter of another shape predicate:* $\exists m \in 1...k \cdot r \in \bar{w}_{im}$ *and r is a root pointer of the predicate* P_m.

For example, we define the `lsegn` predicate to describe a list segment with length property as follows:

$\texttt{data}\ \texttt{c}_1\ \{\ \texttt{c}_1\ \texttt{next};\}$*// data structure declaration*

$\texttt{pred}\ \texttt{lsegn(root,s,n)}\ \equiv\ \texttt{emp} \wedge \texttt{root=s} \wedge \texttt{n=0}$

$\qquad \vee\ \exists\ \texttt{q},\texttt{n}_1 \cdot \texttt{root}{\mapsto}\texttt{c}_1\texttt{(q)}{*}\texttt{lsegn(q,s,n}_1\texttt{)}\wedge\texttt{n}_1\texttt{=n}-1 \wedge \texttt{root}{\neq}\texttt{s}$

$\quad inv\text{: } n{\geq}0;$

The first parameter of `lsegn` is a root parameter.

Our specification language is expressive enough to describe complex data structures, e.g. binary search trees, balance trees [114], trees with parent pointer and tree with linked leaves [76, 90]. For example, we define balance trees as follows:

$\texttt{data}\ \texttt{c}_2\ \{\ \texttt{c}_2\ \texttt{left};\ \texttt{c}_2\ \texttt{right};\}$*// data structure declaration*

$\texttt{pred}\ \texttt{avln(root,n,h)}\ \equiv\ \texttt{emp} \wedge \texttt{root=NULL} \wedge \texttt{n=0} \wedge \texttt{h=0}$

$\qquad \vee\ \exists\ \texttt{l,r,n}_1\texttt{,n}_2\texttt{,h}_1\texttt{,h}_2 \cdot \texttt{root}{\mapsto}\texttt{c}_2\texttt{(l,r)}{*}\texttt{avln(l,n}_1\texttt{,h}_1\texttt{)}{*}\texttt{avln(r,n}_2\texttt{,h}_2\texttt{)}\wedge$

$\qquad\quad \texttt{n=n}_1\texttt{+n}_2\texttt{+1}\wedge\texttt{h=1+max(h}_1\texttt{,h}_2\texttt{)}\wedge-1{\leq}\texttt{h}_1-\texttt{h}_2{\leq}1$

$\quad inv\text{: } n{\geq}0 \wedge h{\geq}0;$

Note: It is required that mutually recursive predicates have at least one base branch each. Reasoning on mutually recursive predicates without any base branch required co-inductive proofs [98], which is beyond scope of this thesis. For example, our current system cannot handle the following infinite predicate: $\texttt{I(x)}\ \equiv\ \exists\ \texttt{q} \cdot \texttt{x}{\mapsto}\texttt{node(\_,q)}{*}\texttt{I(q)}$

Unfolding User-Defined Predicate. The function $unfold(\Delta, \texttt{P}, \bar{t})$ unfolds once the *first* user-defined predicate `P` with actual parameter $\bar{t}$ of the formula Δ. The step is formalized as follows:

$$\frac{\begin{array}{c}\mathtt{P}(\bar{v}){\equiv}\bigvee_{i=1}^{n}(\exists\bar{w}_i\cdot\ \kappa_i{\wedge}\pi_i)\quad \mathit{fresh}\ \bar{w'}_i\quad \rho_i{=}[\bar{w'_i}/\bar{w}_i]\quad \kappa'_i=\kappa_i[\rho_i]\quad \pi'_i=\pi_i[\rho_i]\\ \rho_0{=}[\bar{t}/\bar{v}]\qquad \kappa''_i=\kappa'_i[\rho_0]\quad \pi''_i=\pi'_i[\rho_0]\end{array}}{\mathit{unfold}(\exists\bar{w}_0\cdot\ \mathtt{P}(\bar{t}){*}\kappa_0{\wedge}\pi_0,\mathtt{P},\bar{t})\leadsto\bigvee_{i=1}^{n}(\exists\bar{w}_0\cup\bar{w}'_i\cdot\ \kappa_0*\kappa''_i{\wedge}\pi_0{\wedge}\pi''_i)}$$

In the first line, the function looks up the definition of P, refreshes the existential quantifiers. In the second line, formal parameters are substituted by the corresponding actual arguments. Finally, substituted definition is combined (and normalized) with residual formula as shown in the RHS of $\leadsto$.

Implicit Heap Constraints. Our language does not allow dereference pointers on pure constraints, e.g. $x.p \neq \mathtt{NULL}$, $\mathtt{lsegn}(x,\mathtt{NULL},n){\wedge}x.p\neq\mathtt{NULL}$. This kind of pure constraints implicitly captures constraints on heaps. We require the explicit form, e.g. $x{\mapsto}c_1(p)\wedge p\neq\mathtt{NULL}$, $x{\mapsto}c_1(p){*}\mathtt{lsegn}(p,\mathtt{NULL},n_1){\wedge}p\neq\mathtt{NULL}{\wedge}n_1{=}n{-}1$, which can mostly be obtained by unfolding relevant predicates.

2.2.2 User-Defined Lemma.

Nguyen et. al. [113] proposed a mechanism that allows users to interact with the entailment procedure. Users can provide lemmas to express predicates relationships and a procedure automatically applies those lemmas as alternative predicate unfolding, where possible. Moreover, this is done as a proof search. Lemma can be defined as either weakening or strengthening as follows:

Lemma	::= Weakening Lemma | Strengthening Lemma
Weakening Lemma	::= $\kappa_1{\wedge}\pi_1 \rightarrow \exists w^*{\cdot}\kappa_2{\wedge}\pi_2$
Strengthening Lemma	::= $\kappa_1{\wedge}\pi_1 \leftarrow \exists w^*{\cdot}\kappa_2{\wedge}\pi_2$

For weakening lemma, it requires that $\kappa_1{\wedge}\pi_1$ must contain a user-defined predicate with an explicit root pointer. Similarly for strengthening lemma, it requires that $\kappa_2 \wedge \pi_2$ must contain a user-defined predicate with an explicit root pointer.

For instance, user can define the lemma to express a relationship between list segment `lseg` and acyclic list `lls`, as follows:

$$\text{lemma} \quad \texttt{lseg}(\texttt{root},p,n) \wedge p{=}\texttt{NULL} \leftrightarrow \texttt{lls}(\texttt{root},n)$$

where the definition of the acyclic list `lls`

$$\begin{array}{l} \texttt{pred lls(root,n)} \ \equiv \ \texttt{root=NULL} \wedge \texttt{n=0} \\ \qquad \vee \ \exists\, \texttt{q}, \texttt{n}_1 \cdot \texttt{root} {\mapsto} \texttt{c}_1\texttt{(q)} * \texttt{lls(q,n}_1\texttt{)} \wedge \texttt{n}_1 \texttt{=n} - 1 \end{array}$$

($\leftrightarrow$ is a shorthand of both weakening and strengthening lemmas.)

Nguyen et. al. [113] also proposed an approach to proving user-defined lemmas. That can be encoded as a lemma proving step, named `lemma_check`. For example, to denote the proving of the above lemma, we capture it as follows:

$$\text{lemma\_check} \quad \texttt{lseg}(\texttt{root},p,n) \wedge p{=}\texttt{NULL} \leftrightarrow \texttt{lls}(\texttt{root},n)$$

2.3 Entailment Procedure of Separation Logic.

2.3.1 Overview

There are large number of proposals on entailment procedures for separation logic. We shall now highlight some state-of-the-art procedures in the literature. Smallfoot [11] is the first practical entailment procedure for separation logic. This solver was customised to work with variants of linked list, supports frame inference, but not induction proving. Some optimization on segment feature for the fragment of Smallfoot [11] was presented in [38] (using graph technique),

[118] (using superposition calculus), and [111] (using SMT reduction). However, those proposals did not show how to infer frame (as in [11]) and thus cannot work with modular verification mechanism for heap-manipulating programs. More specially, GRASS-based decision procedures support all shape predicates that can be reduced to some special predicates in GRASS logic, such as SLLB for list segment [120], and GRIT for tree data structures [121]. This reduction approach is promising; as it can check both valid and invalid entailment, support frame inference, abduction and combination of shape and pure properties. However, this approach has not supported induction proving and currently works with just a predefined set of shape predicates.

To handle the combination of heap and pure domains, DRYAD [103, 126] and GRASS approaches [122] make use of the combination capability of SMT-solvers (i.e. Z3 [45]). Also SLEEK [114] employs proof slicing technique [92].

For more expressive predicates, the decision procedures presented in [77, 114, 126] make a good effort to enhance the power of separation logic solvers. Solvers presented in [114, 126] support a combination with the pure domains, but they are not complete. Recently, Iosif et. al. [76] presented a neat proposal for a decidable fragment of separation logic with user-defined predicates. They also provided an implementation together with a sound and complete decision procedure in [77]. It is interesting to see how further this procedure can be extended to handle pure constraints and inductive proving in future.

The cyclic prover [17] showed how to perform automatic induction for the entailment procedure of separation logic. This prover can support fairly expressive shape predicates in separation logic. Unfortunately, this technique has supported neither pure constraints, nor show how frame inference can be done. Hence, cyclic technique is not applicable for general-purpose program verification.

In the following, we discuss in detail SLEEK entailment procedure that is used in the verification system of this thesis.

2.3.2 SLEEK

$$\frac{\begin{array}{c}[\underline{\textbf{ENT-PTO-MATCH}}]\\ \pi_1 \implies r{=}r' \\ \rho{=}[\bar{v}/\bar{w}] \quad \pi_{eq} = \text{freeEQ}(\rho, V) \\ \kappa_1{\wedge}\pi_1{\wedge}\pi_{eq} \vdash^{\kappa * r\mapsto c(\bar{v})}_{V-\bar{w}} \Delta_2[\bar{v}/\bar{w}] \leadsto \Phi_R\end{array}}{r{\mapsto}c(\bar{v}){*}\kappa_1{\wedge}\pi_1 \vdash^{\kappa}_{V} r'{\mapsto}c(\bar{w}){*}\Delta_2 \leadsto \Phi_R} \qquad \frac{\begin{array}{c}[\underline{\textbf{ENT-PRED-MATCH}}]\\ \pi_1 \implies r{=}r' \\ \rho{=}[\bar{v}/\bar{w}] \quad \pi_{eq} = \text{freeEQ}(\rho, V) \\ \kappa_1{\wedge}\pi_1{\wedge}\pi_{eq} \vdash^{\kappa * \mathtt{P}(\{r,\bar{v}\})}_{V-\bar{w}} \Delta_2[\bar{v}/\bar{w}] \leadsto \Phi_R\end{array}}{\mathtt{P}(\{r, \bar{v}\}){*}\kappa_1{\wedge}\pi_1 \vdash^{\kappa}_{V} \mathtt{P}(\{r',\bar{w}\}){*}\Delta_2 \leadsto \Phi_R}$$

$$\frac{\begin{array}{c}[\underline{\textbf{ENT-UNFOLD}}]\\ \Delta_{u_1} \vee ... \vee \Delta_{u_n} = \mathit{unfold}(\mathtt{P}(\bar{v}){*}\Delta_1, \mathtt{P}, \bar{v}) \\ \Delta_{u_i}{*}\Delta_1 \vdash^{\kappa}_{V} \Delta_2 \leadsto \Delta_{R_i} \quad i{=}1...n\end{array}}{\mathtt{P}(\bar{v}){*}\Delta_1 \vdash^{\kappa}_{V} \Delta_2 \leadsto \bigwedge\{\Delta_{R_i} \mid i{\in}1...n\}}$$

$$\frac{\begin{array}{c}[\underline{\textbf{ENT-FOLD}}]\\ (\Delta^r, \kappa^r, \pi^r){\in}\mathit{fold}^{\kappa}(\mathtt{P}(p_1\bar{v_1}){*}\kappa_1{\wedge}\pi_1, p_1{\mapsto}c(\bar{v_2})) \\ (\pi^a, \pi^c){=}\mathit{split}^{\{v_2^*\}}_{V}(\pi^r) \quad \Delta^r{\wedge}\pi^a{\vdash}^{\kappa^r}_{V}(\kappa_2{\wedge}\pi_2{\wedge}\pi^c) * \Phi\end{array}}{\mathtt{P}(p_1, \bar{v_1}){*}\kappa_1{\wedge}\pi_1 \vdash^{\kappa}_{V} p_1{\mapsto}c(\bar{v_2}){*}\kappa_2{\wedge}\pi_2 \leadsto \Phi}$$

$$\frac{\begin{array}{c}[\underline{\textbf{ENT-EMP}}]\\ \mathtt{XPURE}(\kappa_1{*}\kappa){\wedge}\pi_1 \implies \exists V{\cdot}\pi_2\tau\end{array}}{\kappa_1{\wedge}\pi_1 \vdash^{\kappa}_{V} \pi_2 \leadsto \kappa_1{\wedge}\pi_1}$$

$$\frac{\begin{array}{c}[\underline{\textbf{ENT-LHS-EX}}]\\ \mathit{fresh}\ w \\ \Delta_1[w/v] \vdash^{\kappa}_{V} \Delta_2 \leadsto \Phi_R\end{array}}{\exists v \cdot \Delta_1 \vdash^{\kappa}_{V} \Delta_2 \leadsto \Phi_R} \qquad \frac{\begin{array}{c}[\underline{\textbf{ENT-RHS-EX}}]\\ \Delta_1 \vdash^{\kappa}_{V\cup\{w\}} (\Delta_2[w/v]) \leadsto \Delta_i \\ \Phi_R{=}\exists\ w \cdot \Delta_i\end{array}}{\Delta_1 \vdash^{\kappa}_{V} (\exists\ v \cdot \Delta_2) \leadsto \Phi_R}$$

Figure 2-3: Basic Inference Rules for Entailment Checking

SLEEK is an entailment procedure of separation logic that supports user-defined inductive shape predicates and frame inference.

As separation logic is a sub-structural logic, we have to account for heap memory as a resource. Thus, entailment in separation logic is typically supported with a frame inference capability [33, 116], similar to the following format:

$$\Phi_1 \vdash \Phi_2 \leadsto \Phi_3$$

whereby antecedent Φ_1 entails Φ_2 with a residue frame captured by Φ_3. Logically, the above entailment is equivalent to $\Phi_1 \Longrightarrow \Phi_2{*}\Phi_3$ where Φ_3 may contain existential variables that have been instantiated and pure formula that were already established in Φ_1.

For each entailment check, our procedure proves that (i) all matching models of the antecedent would be subsumed by models of the consequent; (ii) irrelevant part of the antecedent will be inferred as residual frame. Firstly, the matching of heap part is performed until heap in the consequent is empty ($[\underline{\textbf{ENT}-*-\textbf{MATCH}}]$, $[\underline{\textbf{ENT}-\textbf{UNFOLD}}]$, $[\underline{\textbf{ENT}-\textbf{FOLD}}]$ inference rules). After that, the implication of the pure part is checked semantically through external SMT solvers and theorem provers ($[\underline{\textbf{ENT}-\textbf{XPURE}}]$ inference rule). The inference rules are presented in Figure 2-3. Typically, an entailment is performed as follows.

- **Matching.** This matches up identified heaps of LHS and RHS. Starting from identified root pointers, the procedure keeps matching all their reachable heaps with both points-to (with $[\underline{\textbf{ENT}-\textbf{PTO}-\textbf{MATCH}}]$ rule) and user-defined predicate matching (with $[\underline{\textbf{ENT}-\textbf{PRED}-\textbf{MATCH}}]$ rule). The former (latter) matches two points-to (user-defined, resp.) predicates in antecedent and consequent if they have identified root and stores matched points-to predicates in a footprint heap. After that, it unifies the corresponding fields of matched roots by using auxiliary function freeEQ(ρ, V): freeEQ($[u_i/v_i]_{i=1}^{n}, V$) $= \bigwedge_{i=1}^{n}\{u_i = v_i \mid v_i \notin V\}$.

- **Unfolding-Folding.** This derives alternative heap chains, connected points-to or user-defined predicates. When the procedure is unable to make any progress on matching, it will look up alternative chains for matching through unfolding heap predicates. For termination, SLEEK restricts user-defined predicates to within a well-founded form. For unfolding in the antecedent ($[\underline{\textbf{ENT}-\textbf{UNFOLD}}]$ rule), the unfolding performs a case split in the proof. For unfolding in the consequent ([**ENT−FOLD**] rule), instead of unfolding the RHS and doing proof search over case split, SLEEK performs folding in the LHS. This helps instantiate existential parameters for better completeness. The `unfold` operator is defined in section 2.2. The `fold` operation is defined next.

- **Over-approximate Reduction.** We reduce entailment checking on separation logic to implication checking on first-order logic. When the consequent becomes an empty heap, the procedure performs a sound reduction to transform the entailment to an implication on first-order logic, i.e. $\pi_a \implies \pi_c$. Technically, to perform such implication checking, the following satisfiability check is performed: $\texttt{sat}(\pi_{\mathbf{a}} \wedge \neg(\pi_{\mathbf{c}}))$. If it returns `unsat`, the result of the implication is valid; otherwise, the result of the implication is unknown.

Folding User-Defined Predicate. The formalism of the `fold` function on predicate `P` is as follows:

$$\frac{\begin{array}{c}\mathtt{P}(\bar{v}){\equiv}\bigvee_{i=1}^{n}(\exists\bar{w}_i\cdot\ \kappa_i{\wedge}\pi_i) \quad \rho_0{=}[\bar{t}/\bar{v}] \\ \kappa{\wedge}\pi \vdash_{\bar{t}}^{\kappa'} \bigvee_{i=1}^{n}(\exists\bar{w}_i\cdot\ \kappa_i{\wedge}\pi_i)[\rho_0] \rightsquigarrow \{\Delta_i,\kappa_i,V_i,\pi_i\}_{i=1}^{n} \quad W_i{=}V_i{-}\bar{t}\end{array}}{\mathtt{fold}^{\kappa'}(\kappa{\wedge}\pi,\mathtt{P},\bar{t}) \rightsquigarrow \{\Delta_i,\kappa_i,\exists W_i\cdot\pi_i\}_{i=1}^{n}}$$

First it looks up suitable branches of the predicate `P` via entailment checks. These checks require a special revision of the entailment procedure which returns three more components: consumed heap nodes (κ_i), existential variables used (V_i), and final consequent π_i.

When folding a user-defined predicate $\mathtt{P}(\bar{v})$, pure constraints related to $\bar{v}$ are important. The constraints related to parameters of $\bar{v}$ that are free will be moved to LHS of the entailment as it can help support instantiation of existential variables. Otherwise, they are kept in the RHS. This processing of

2.2.4 $\texttt{lsegn}(q_2, \texttt{NULL}, n_2) \wedge n{>}0 \wedge p{=}q_3 \wedge n_2{=}n{-}1 \wedge q_3{=}v$
$\quad\vdash^{\kappa_2}_{\emptyset} \texttt{emp} \leadsto (\Delta_2, \text{valid})$ **EMP**

2.2.3 $\texttt{lsegn}(q_2, \texttt{NULL}, n_2) \wedge n{>}0 \wedge p{=}q_3 \wedge n_2{=}n{-}1 \wedge q_3{=}v \vdash^{\kappa_2}_{\emptyset} \texttt{emp}$ **MATCH**

2.2.2 $p{\mapsto}c_1(q_2){*}\texttt{lsegn}(q_2, \texttt{NULL}, n_2) \wedge n{>}0 \wedge p{=}q_3 \wedge n_2{=}n-1$
$\quad\vdash^{\kappa_1}_{\emptyset} q_3{\mapsto}c_1(v) \leadsto (\Delta_3, \text{valid})$ **RHS−EX**

2.2.1 $p{\mapsto}c_1(q_2){*}\texttt{lsegn}(q_2, \texttt{NULL}, n_2) \wedge n{>}0 \wedge p{=}q \wedge n_1{=}n-1$
$\quad\vdash^{\kappa_1}_{\emptyset} \exists q \cdot (q{\mapsto}c_1(v))$ **LHS−EX**

2.2 $\exists q_1, n_1 \cdot p{\mapsto}c_1(q_1){*}\texttt{lsegn}(q_1, \texttt{NULL}, n_1) \wedge n{>}0 \wedge p{=}q \wedge n_1{=}n-1$
$\quad\vdash^{\kappa_1}_{\emptyset} \exists q \cdot (q{\mapsto}c_1(v))$ **UNFOLD(Ind)**

2.1.1. $\texttt{false} \vdash^{\kappa_1}_{\emptyset} \exists q{\cdot}(q{\mapsto}c_1(v)) \leadsto (\Delta_1, \text{valid})$ **EMP**

2.1 $p = \texttt{NULL} \wedge n{>}0 \wedge p{=}q \wedge n{=}0 \vdash^{\kappa_1}_{\emptyset} \exists q \cdot (q{\mapsto}c_1(v))$ **UNFOLD(Base)**

1 $\texttt{lsegn}(p, \texttt{NULL}, n){\wedge}n{>}0{\wedge}p{=}q \vdash^{\kappa_1}_{\emptyset} \exists q \cdot (q{\mapsto}c_1(v))$ **MATCH**

0 $x{\mapsto}c_1(p){*}\texttt{lsegn}(p, \texttt{NULL}, n){\wedge}n{>}0 \vdash^{\texttt{emp}}_{\emptyset} \exists q{\cdot}(x{\mapsto}c_1(q){*}q{\mapsto}c_1(v))$
$\quad\leadsto (\Delta_1 \vee \Delta_3, \text{valid})$

Figure 2-4: SLEEK Entailment Procedure: An Example.

pure constraints was implemented through the function *split* [33].

Example. We illustrate the entailment procedure through the example shown in Figure 2-4. In the proof, the LHS of the entailment has been performed bottom-up while the result in RHS is computed top-down. For simplicity, we discard some intermediate RHS, and write **UNFOLD** instead of $\boxed{\textbf{ENT−UNFOLD}}$ to annotate rule applied at each step. The steps from 2.1* correspond to base case of the unfolding over predicate $\texttt{lsegn}(p, \texttt{NULL}, n)$. The steps from 2.2* correspond to the inductive case of the unfolding over predicate $\texttt{lsegn}(p, \texttt{NULL}, n)$. The footprints accumulated during this checking are $\kappa_1{=}x{\mapsto}c_1(p)$, and $\kappa_2{=}\kappa_1 * q{\mapsto}c_1(\texttt{NULL})$. The residual heap in two cases of the unfolding is: $\Delta_1{=}\texttt{false}$, $\Delta_2{=}\texttt{lsegn}(q_1, \texttt{NULL}, n_2){\wedge}n{>}0{\wedge}p{=}q{\wedge}n_2{=}n{-}1$, and $\Delta_3 = \exists q \cdot \Delta_2$.

2.4 A Motivating Example

We present a motivating example to highlight our contributions. We employ the fragment of separation logic in section 2.2 to express heap abstraction.

```
1. struc node { struc node *next };
2. struc node* get_last (struc node *x) {
3.   if (x->next==NULL)
4.    return x;
5.   else {
6.     return get_last(x->next);
7.   }
8. }
```

Figure 2-5: Motivating Example: Code of `get_last` Method.

We illustrate our proposal through the `get_last` method presented in Figure 2-5. The `get_last` example is to get the last element of a singly-linked list x.

2.4.1 Complete Specification with an Error Calculus

Specification is used to describe the relation of input and output in a program. The `get_last` method has two return points at lines *4*, and *6*. Both the returns describe explicit *normal execution*. Besides, the program contains an implicit assumption at the memory dereference on the pointer x at line *3*. That is, for memory safety, the input linked list x must have at least one element. Otherwise, the program raises a runtime error as an *abnormal execution*.

We classify the former (normal) execution as good scenarios and the latter (abnormal) execution as bad scenarios. We expect that users would like to capture both these scenarios in the same specification and be able to verify its implementation against the expected specification. We would like to emphasize that existing specification approaches focus on only specifying good scenarios of functional properties and typically ignore bad scenarios. For example, existing separation logic approaches would declare abstraction for the data structure `node` and the linked list `lseg`, as follows:

$$\begin{array}{l} \texttt{data node} \; \{ \; \texttt{node next}; \} \textit{// data structure declaration} \\ \texttt{pred lseg(root,s)} \; \equiv \; (\texttt{emp} \wedge \texttt{root=s}) \\ \qquad \vee \; (\exists \; \texttt{q} \cdot \texttt{root} {\mapsto} \texttt{node(q)} {*} \texttt{lseg(q,s)} \wedge \texttt{root} {\neq} \texttt{s}); \end{array}$$

and capture the functional specification of the `get_last` program as:

$$\begin{array}{ll} \texttt{requires} & \texttt{lseg(x,NULL)} \wedge \texttt{x} {\neq} \texttt{NULL} \\ \texttt{ensures} & \texttt{lseg(x,last)} \; {*} \texttt{last} {\mapsto} \texttt{node(NULL)} \wedge \texttt{res=last} \wedge \texttt{x} {\neq} \texttt{NULL}; \end{array} \qquad (1)$$

where `res` is a reserved variable to denote the result of the method.

For bad scenarios, the basis of our proposal is the identification of a lattice domain with four points (that are partially ordered) and a calculus, called *error calculus*. While the four-point domain is used to capture the status of each program state, the calculus is meant to combine and capture program states during symbolic execution. The four points that are used to characterize program status are as follows:

- $\perp$: denotes dead code or non-terminated loops of execution.
- $\surd$: denotes normal program execution.
- $\mho$: denotes expected abnormal execution.
- $\top$: denotes unknown execution. That is, it could either be $\perp$, or $\surd$ or $\mho$.

As such, to capture the requirement on the *erroneous* execution of the `get_last` method, the user can provide the following specification:

$$\texttt{requires} \;\; \texttt{x=NULL} \;\; \texttt{ensures} \; (\texttt{true}) \; \mho;$$

To express a *complete* specification for both good and bad scenarios, we integrate the four-point status into the *structured* specification presented in [60]. For example, the specification of the get_last can be constructed as in Figure 2-6.

```
case {
  x=NULL →  ensures (true ) ℧;
  x≠NULL →
     requires  lseg(x,NULL)
     ensures   lseg(x,last) ∗last↦node(NULL) ∧ res=last √;
 }
```
(2)

Figure 2-6: Complete Specification of get_last Method.

Moreover we also propose a verification mechanism to support this complete specification language. The proposal provides an entailment procedure on pure logic to determine the status of each implication. This is being also tightly integrated into a separation logic entailment procedure and a forward symbolic reasoning mechanism based on a set of Hoare triple rules.

2.4.2 Shape Analysis via Second-Order Bi-Abduction

A main disadvantage of deductive verification systems is the need for manually writing methods' specification. To overcome this, we will highlight specification inference in the next two subsections. In the context of pointer-based programs, we propose the second-order bi-abduction for shape analysis. That analysis will help to infer shape specifications for error-free programs.

Current shape analysis mechanisms [13, 20, 57] using list segment abstraction are unable to capture the pre-condition for the get_last method under which memory-safe operation could be carried out. In this thesis, we propose a shape synthesis to infer shape specification that guarantees memory safety. In the context of the get_last method, our shape analysis would introduce the following *unknown predicates* as input:

$$\texttt{requires}\ \texttt{H(x)} \qquad \texttt{ensures}\ \texttt{G(x,res)};$$

whereby `res` is a reserved keyword to represent the return value of the method.

Our shape analysis shall be supported by a separation logic verifier that is able to collect and solve the proof obligations involving the unknown predicates that must hold in order to guarantee memory safety. The analysis can infer the

```
A1.  H(x) ⇒ x↦node(q)∗U(q)
A2.  U(q)∧q≠NULL ⇒ H(q)
A3.  x↦node(q)∗G(q,res)∧q≠NULL ⇒ G(x,res)
A4.  x↦node(q)∗U(q)∧q=NULL∧res=x ⇒ G(x,res)
```

Figure 2-7: Result of the Shape Analysis on `get_last` Method.

set of four relational assumptions (in separation logic) form as in Figure 2-7; where `U` is new unknown predicate that was introduced during the synthesis. These four assumptions `A1`, `A2`, `A3` and `A4` have been inferred while the proposed second-order bi-abductive entailment procedure discharges the proof obligations generated for safety. More concretely, `A1` has been generated at right before line 3 to ensure absence of null-dereference at line 3; `A2` has been generated right after `return` statement at line 4 to ensure postcondition; `A3` has been generated right before line 6 to meet pre-condition of the function call at line 6; and `A4` has been generated right after `return` statement at line 6 to ensure postcondition. This set of assumptions indirectly expresses the expected shape abstraction of predicates. Its validity guarantees the safety of the `get_last` program.

The use of the separation logic formalism facilitates abstractions through inductive shape predicates expressed using separation logic formulas. Furthermore we may leverage on a bi-abduction mechanism for pure properties [135] to refine the resulting shape predicates with other relevant properties of interests.

2.4.3 Transformational Approach to Shape Predicates

Derivation. The set of assumptions above is not easily understood by programmers. For example, it is non-trivial to figure that `U` is an acylic list, `H` and `G` are full lists with at least one element and the second parameter of the predicate `G` points to the last element. From the output of the shape analysis presented in Figure 2-7, we shall now derive definition for each shape predicate. The results are as follows:

$$\begin{array}{l}
\mathtt{H(x)} \;\equiv\; \mathtt{x{\mapsto}node(q){*}U(q)} \\
\mathtt{U(x)} \;\equiv\; \mathtt{emp{\wedge}x{=}NULL} \quad \vee \quad \mathtt{x{\mapsto}node(q){*}U(q)} \\
\mathtt{G(x,res)} \;\equiv\; \mathtt{x{\mapsto}node(NULL)} \wedge \mathtt{res{=}x} \quad \vee \quad \mathtt{x{\mapsto}node(q){*}G(q,res)}
\end{array}$$

Using them, the specification for the `get_last` method can now be refined to:

$$\mathtt{requires}\ \mathtt{x{\mapsto}node(q){*}U(q)} \qquad \mathtt{ensures}\ \mathtt{G(x,res)} \tag{3}$$

Specification (3) and predicate definitions inferred in this section are quite precise, but are still less understandable than the user-supplied specification (1) given in section 2.4.1. Such inferred specification and shape predicates would be subjected to a normalization phase, as described next.

Normalization. To illustrate the important of the normalization, consider the method `append` in Figure 2-8. This method allocates a new node and appends it at the end of the input list. It employs the method `get_last` to return the last element. Assume that specification of the method `get_last` at (3). Let us examine a *modular* verification (like [13, 114, 90]) on the method `append` with given specification at lines 2, 3 in Figure 2-8. At line 9, in order to ensure memory

```
struc node* append (struc node* x)
 requires lseg(x,NULL)
 ensures ∃p · lseg(x,p) *p↦node(NULL)
{
  struc node*tmp = (struc node*) malloc(sizeof(struc node));
  tmp->next = NULL;
  if (x != NULL) {
     struc node* last = get_last(x);
  /* α=G(x, last) */
    last->next=tmp;//fail here
    tmp = x;
  }
  return tmp;
}
```

Figure 2-8: Code of append Method.

safety of the pointer dereference, the state α must imply that the pointer *last* be allocated. This means the following entailment must hold:

$$\texttt{G}(\texttt{x}, \texttt{last}) \vdash \exists q \cdot \texttt{last}{\mapsto}\texttt{node}(\texttt{q}) \qquad (4)$$

with G being the predicate inferred in the previous section. We believe that such entailment is beyond the capability of existing entailment procedures in separation logic. To overcome this problem, one solution is to transform the definition of the predicate G such that the G explicitly expresses that pointer last as a cutpoint that is allocated. In the following, we implement that solution through a transformation.

First, we present a normal form which is based on the fragment of separation logic with inductive predicates [76]. This form syntactically restricts to ensure that each predicate describes heap region that is accessible via one root pointer. Shape predicates with zero or more root pointers could be transformed into this normalized form. Second, we present a semantic transformation via a lemma mechanism. We extend the lemma mechanism in [113] to support lemma synthesis:

$$\texttt{lemma\_infer}\ [\bar{\texttt{U}}]\ \texttt{LHS}\ \rightarrow\ \texttt{RHS}$$

$$\texttt{lemma\_infer}\ [\bar{\texttt{U}}]\ \texttt{LHS}\ \leftarrow\ \texttt{RHS}$$

where $[\bar{\texttt{U}}]$ is a set of *unknown* shape predicates that are to be inferred. To infer a lemma, we first make an assumption on the validity of the lemma. To support inductive proof, this lemma itself would be considered as induction hypothesis. After that, we make use of second-order bi-abductive mechanism to infer a set of constraints that guarantee the validity of the entailment checks. Lastly, we recursively invoke the transformation to derive shape predicate definition.

Our normalization mechanism comprises a set of three operations called: useless parameter elimination, predicate splitting, and predicate reuse. We illustrate these operations (*predicate splitting*, useless parameter elimination and *predicate reuse*) through the normalization of the predicate `G`.

A root parameter is a parameter that is allocated or is equal to `NULL` in every branch of predicate definition. (See section 2.2 for a formal definition of root parameter). In the inferred definition of `G`, since two parameters `x` and `res` are roots, the synthesized predicate `G` is not normalized. We are going to transform that predicate into normal form through lemma synthesis. Based on the violation of the restriction above, we propose to *split* the predicate `G` using the following *weakening lemma*:

$$\texttt{lemma\_infer}\ [\texttt{U}_1;\texttt{U}_2]\ \texttt{G(x,res)} \rightarrow \texttt{U}_1\texttt{(x,res)} * \texttt{U}_2\texttt{(res,x)}$$

where $\texttt{U}_1$ and $\texttt{U}_2$ are *unknown* predicates that need to be inferred.

Using shape analysis (e.g. in the previous section), we can infer the following definition:

$$\begin{aligned} \mathtt{U}_1(\mathtt{x},\mathtt{s}) &\equiv \mathtt{emp} \wedge \mathtt{x}{=}\mathtt{s} \ \vee \ \mathtt{x}{\mapsto}\mathtt{node(q)}{*}\mathtt{U}_1(\mathtt{q},\mathtt{s}) \\ \mathtt{U}_2(\mathtt{y},\mathtt{z}) &\equiv \mathtt{y}{\mapsto}\mathtt{node(NULL)} \end{aligned}$$

After that, the second parameter of the predicate $\mathtt{U}_2$ is detected as a *useless* parameter. This parameter is semantically eliminated through the following lemma:

$$\mathtt{lemma\_infer}\ [\mathtt{U}_3]\ \mathtt{U}_2(\mathtt{y},\mathtt{z}) \leftrightarrow \mathtt{U}_3(\mathtt{y})$$

whereby the predicate $\mathtt{U}_3$ is another unknown predicate and inferred as:

$$\mathtt{U}_3(\mathtt{y}) \ \equiv \ \mathtt{y}{\mapsto}\mathtt{node(NULL)}$$

Finally, the predicate $\mathtt{G}$ is decomposed into two predicate $\mathtt{U}_1$ and $\mathtt{U}_3$. This split is described in the synthesized lemma as follows:

$$\text{lemma}\ \ \mathtt{G}(\mathtt{x},\mathtt{res}) \rightarrow \mathtt{U}_1(\mathtt{x},\mathtt{res}) * \mathtt{U}_3(\mathtt{res})$$

Furthermore, to obtain an equivalent transformation, the following *strengthening* lemma shall be proven to be valid:

$$\mathtt{lemma\_check}\ \mathtt{G}(\mathtt{x},\mathtt{res}) \leftarrow \mathtt{U}_1(\mathtt{x},\mathtt{res}) * \mathtt{U}_3(\mathtt{res})$$

Once inferred, the definitions of $\mathtt{U}_1$ and $\mathtt{U}_3$ can be used to prove the validity of the above strengthening lemma. With this result, the following equivalent lemma can be confirmed:

$$\text{lemma}\ \ \mathtt{G}(\mathtt{x},\mathtt{res}) \leftrightarrow \mathtt{U}_1(\mathtt{x},\mathtt{res}) * \mathtt{U}_3(\mathtt{res})$$

Consequently, if the shape predicate `lseg` in section 2.4.1 is given in advance (as library predicates), the *reuse* operation will match inferred `U` predicate in section

2.4.2 and inferred $\mathtt{U_1}$ predicate to form the following lemmas:

$$\mathtt{lemma\ U(x)} \leftrightarrow \mathtt{lseg(x,NULL)}$$

$$\mathtt{lemma\ U_1(x,s)} \leftrightarrow \mathtt{lseg(x,s)}$$

Finally, we generate the following specification for the `get_last` method:

$$\begin{array}{ll} \mathtt{requires} & \mathtt{x{\mapsto}node(q)} * \mathtt{lseg(x,NULL)} \\ \mathtt{ensures} & \mathtt{lseg(x,last)} * \mathtt{last{\mapsto}node(NULL)} \wedge \mathtt{res{=}last}; \end{array}$$

With this, let us now return to the entailment (4). Using the synthesized lemma above, the predicate `G` of the entailment would be transformed as follows:

$$\mathtt{lseg(x,last)} * \mathtt{last{\mapsto}node(NULL)} \vdash \exists q \cdot \mathtt{last{\mapsto}node(q)}$$

The procedures of [13, 114, 90] can now process the above entailment and thus verify memory safety property of the method `append`.

In the next three chapters, we present our main technical contributions in order for enhancing the HIP/SLEEK system to an automated verification system of *complete* specifications. Lastly, we conclude the thesis followed up by a future work discussion.

Chapter 3

Verifying Complete Specification

We present an error calculus to support a novel specification mechanism for sound and/or complete safety properties that are to be given by users. With such specifications, our calculus can form a foundation for both proving program safety (nothing bad will happen) and/or discovering real errors (something bad will happen). The basis of our calculus is an algebra with *a lattice domain* of four abstract statuses (namely *unreachability*, *validity*, *must-error* and *may-error*) on possible program states and *four operators* for this domain to calculate suitable program status. We show how *proof search* and *error localization* can be supported by our calculus. Our calculus can also be extended to *separation logic* with support for user-defined predicates and lemmas. We have implemented our calculus in an automated verification tool for pointer-based programs. Initial experiments have confirmed that it can achieve the dual objectives, namely of safety proving and bug finding, with modest overheads.

3.1 Complete Specifications

Traditionally, program specifications are given primarily for safety scenarios, i.e. proving functional correctness and memory safety of programs, and are used to describe the states under which program execution would occur safely. When successfully verified, such specifications are said to be *sound* for their specified input scenarios. That is, a specification is said to be *sound* if it has identified input scenarios (or preconditions) that are guaranteed to lead to safe program execution. However, we are also interested in *complete* specifications that will additionally verify the remaining input scenarios (that lead to bad execution, execution failure) are invalid ones. Informally, a specification is said to be *complete* if it has unambiguously identified both input scenarios that lead to safe code execution, and input scenarios that lead to code execution failure.

Such complete specifications for programs are helpful for two reasons. Firstly, they can be used to specify precisely (through weakest precondition[1]) when inputs can be handled correctly by programs. Conversely, we are also able to precisely identify when programs would fail to work properly (or safely). Secondly, the specifications on erroneous inputs can be used to help pinpoint actual software *bugs* in programs as they could be used to indicate where each given error occurs.

Though useful, the task of capturing complete specifications is very challenging, and may not always be possible since the input scenarios under which failures could occur may not be unambiguously specified and verified. In this paper, we shall provide the basic mechanisms that can help specify complete specifications, where possible. To achieve this goal, we propose *a lattice domain* of four abstract status (namely *unreachability*, *validity*, *must-error* and *may-error*) and make use of the validity (must-error) status for specifying safe (unsafe, resp.) execution scenarios. Furthermore, when the complete requirements are hard (or impossible) to specify, we have also provided approximation mechanisms that can help us specify *near-complete* specifications through the use of *may-error* as opposed to *must-error*

[1]While it may be desirable to have weakest precondition that guarantee safety or correctness, we also allow flexibility for users to specify a wider range of specifications that include those with either stronger preconditions and/or weaker postconditions. Though weaker specifications give fewer guarantees, often they can be verified more easily and may be enough to ensure reliability.

classification in weakened postcondition.

Our motivation for developing complete specifications for programs was further heightened by the VSTTE competition [1] that was held in November 2011. Out of five problems that the participants were asked to verify for safety and correctness, there were two problems (problem 4 and problem 5) where more complex specifications that satisfy *completeness* were requested. As complete specifications must additionally address erroneous scenarios, we have recently developed a comprehensive verification framework that could just as easily deal with input scenarios that invoke errors, as it would with input scenarios that led to safe program execution.

To efficiently support complex data structures in pointer-manipulating programs, we make use of separation logic (see section 2.2). Our verification system employs the entailment procedure presented in [114]. As presented in section 2.3, this procedure follows `UNFOLD/FOLD` and `MATCH` paradigm to search a proof for entailment check. To support such proof search, we extend the calculus to a set of sound structural rules. Another difficulty is how to define negation for inductive predicates on complex data structures for the completeness proving. We provide a machinery for this by detecting contradiction at the predicate level and through user-supplied lemmas. These lemmas, in turn, can be automatically proven and applied.

Yet another benefit from our calculus is the capability of localizing program statements relevant to errors. Existing approaches for error localization, e.g. [65, 83], are typically separate from program verification system. Thus the error localization component cannot exploit information, e.g. error trace conditions, from program verifiers. Hence, such information may be constructed twice, one for verification and another for error localization. In contrast, our system can support a combination of program verification and error localization by first l

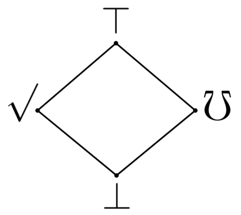

Figure 3-1: Status on Program States.

extending the four-point lattice with error messages and then enhancing the structural rules to support the localization. Indeed, our system can provide concise localizations for both *must* and *may* errors, and thus help the user to comprehend the reason for verification failures.

3.2 Motivation and Overview

3.2.1 An Algebra on Status of Program States

$\sqcap$	$\top$	$\mho$	$\surd$	$\bot$
$\top$	$\top$	$\mho$	$\surd$	$\bot$
$\mho$	$\mho$	$\mho$	$\bot$	$\bot$
$\surd$	$\surd$	$\bot$	$\surd$	$\bot$
$\bot$	$\bot$	$\bot$	$\bot$	$\bot$

$\sqcup$	$\top$	$\mho$	$\surd$	$\bot$
$\top$	$\top$	$\top$	$\top$	$\top$
$\mho$	$\top$	$\mho$	$\top$	$\mho$
$\surd$	$\top$	$\top$	$\surd$	$\surd$
$\bot$	$\top$	$\mho$	$\surd$	$\bot$

$\otimes$	$\top$	$\mho$	$\surd$	$\bot$
$\top$	$\top$	$\mho$	$\top$	$\bot$
$\mho$	$\mho$	$\mho$	$\mho$	$\bot$
$\surd$	$\top$	$\mho$	$\surd$	$\bot$
$\bot$	$\bot$	$\bot$	$\bot$	$\bot$

$\oplus$	$\top$	$\mho$	$\surd$	$\bot$
$\top$	$\top$	$\top$	$\surd$	$\bot$
$\mho$	$\top$	$\mho$	$\surd$	$\bot$
$\surd$	$\surd$	$\surd$	$\surd$	$\bot$
$\bot$	$\bot$	$\bot$	$\bot$	$\bot$

Figure 3-2: An Algebra on Status of Program States.

The basis of our proposal is the identification of an algebra $(\mathcal{E}, \mathcal{F})$ in which $\mathcal{E}$ is a lattice domain (see Fig. 3-1) with four points used to capture the status of each program state, while $\mathcal{F}$ is a set of four binary operators (meet ($\sqcap$), join ($\sqcup$), compose ($\otimes$) and search ($\oplus$)) to combine the status of program states.

The four points that are used for program status are as follows:

- $\bot$: denotes an unreachable state.
- $\surd$: denotes a valid program state from normal program execution.
- $\mho$: denotes a state that corresponds to a must (or definite) error scenario.
- $\top$: denotes a state that corresponds to a may error or an unknown scenario. That is, it could either be $\bot$, or $\surd$ or $\mho$.

Note that the must error status ($\mho$) subsumes the unreachable $\bot$ status. Where possible, we would also like to classify a *satisfiable* must error status that explicitly excludes the $\bot$ status. This is to help us identify input scenarios for *real bugs* that are reachable, and can be achieved by confirming that the state at that program point is provably satisfiable. The may error status ($\top$) comes from imprecision or from dependency on some unknown input. In our system, potential sources of imprecision include imprecise specifications, imprecise invariants of complex data structures and incomplete domains. Although we could separately identify those kinds of imprecision, for simplicity we uniformly specify them with the $\top$ status value. In the implementation, we distinguish them through different messages with status (see Sec. 3.4.3).

Let $\preceq$ be a partial ordering relation on status whereby $\tau_1 \preceq \tau_2$ means status τ_1 is more precise than status τ_2. The $\sqcup$ and $\sqcap$ operators denote the least upper bound and the greatest lower bound, respectively, over the lattice domain. The domain $\mathcal{E}$ and two operations $\sqcap$, $\sqcup$ form a complete lattice $\mathcal{D} = \langle \mathcal{E}, \preceq, \sqcup, \sqcap, \bot, \top \rangle$ organized as shown in Fig. 3-1. This lattice forms a core part of the underlying abstract semantics for our system. Furthermore, $\bot$ is *zero* element of $\otimes$ and $\oplus$ operations; it means $\mathtt{x} \oplus \bot = \bot$ and $\bot \otimes \mathtt{x} = \bot$ for any values $\mathtt{x}$. The remaining calculations of $\otimes$ and $\oplus$ are illustrated in Fig. 3-2. The $\otimes$ operator is meant to support conjunctive proving, and searches for *failures* from $\mho$ and $\top$ status . The $\oplus$ operator is meant to support *proof search*, and searches for $\surd$ status to succeed in proving. Thus the priority order of the $\otimes$ operator is $\mho$, $\top$ and lastly $\surd$, and the priority order of the $\oplus$ operator is $\surd$, $\top$ and lastly $\mho$. Contrast this with the $\sqcup$ operator which doesn't have any priority between $\surd$ and $\mho$. So it would simply yield $\top$ when the two status are combined together.

3.2.2 Mechanism for Sound and Complete Specifications

To illustrate our new specification mechanism, we consider a method that returns the data which its input points to, as shown below

```
int get_data(node x)
  case{ x≠NULL →  requires x↦node⟨d,p⟩ ensures (res=d) √;
         x=NULL → ensures (true ) ℧;  }
```

where `res` is a reserved identifier denoting the method's result and the data structure `node` is declared as: `data node { int val; node next }`.

We would like to remind that each method is specified by pre- and post-conditions (through separation logic formulas), denoted by `requires` and `ensures` keyword, respectively. In the specification above, we also use structured specifications [60] where disjoint conditions are expressed using case construct for expressing both sound (with $x{\neq}$`NULL` condition) and complete (with $x{=}$`NULL` condition) requirements, as can be seen for the above specification of `get_data` (with the $\surd$ and $\mho$ status in postconditions, resp.). In comparison, if we are only interested in sound specification, we would just use the following instead:

```
int get_data(node x)
  requires x↦node⟨d,p⟩ ensures (res=d) √;
```

Occasionally, it may be possible to automatically generate complete specification by negating the input conditions of sound specification. However, this may not always be feasible. Firstly, negation computation may be hard to implement in complex domains. For example, it is unclear how to compute negation in separation logic (which our system relies on). Secondly, not all methods have clearly delineated boundary between sound and complete conditions, as an example, consider the interactive schedule (`ischedule`) method in Fig. 3-3. With *prio*=0 condition, this method's status depends on the user input which is unknown at verification time. Therefore, there exists a gap between soundness and completeness that cannot be derived using just the negation operation.

For this example, we can instead provide a *near-complete* specification, as shown in Fig. 3-3. Informally, a specification is said to be *near-complete* if it captures all possible input conditions but contains either a $\top$ program status or an ambiguous disjunction, comprising of both $\surd$ and $\mho$ status, in its postconditions.

We note that our approach for proving the completeness of program is based on the assumption that the user-supplied specification is complete; namely that it covers all values of the input domain and that each error program state denotes an input scenario where no valid output state is possible. Checking the completeness of specifications is a challenging research direction that could be investigated in the future.

```
int ischedule(int prio){
  if (prio>0)/*run it */ return 0;
  else if (prio<0)  abort();
  else{
   printf("Allow this task to run? y or n");
   char c=getc();
   if (c ='y')/*run it */ return 0;
   else  abort(); } }
Sound Specification:
  int ischedule(int prio)
    requires prio>0  ensures (res=0)√;
Near-Complete Specification:
  int ischedule(int prio)
   case { prio>0 → ensures (res=0)√;
     prio<0 → ensures (true )℧;
     prio=0 → ensures (true )⊤; }
```

Figure 3-3: Code and Specification of `ischedule` Method.

3.2.3 Essence of Error Calculus

To highlight how our calculus can be used to verify programs, consider the method `foo` in Fig. 3-4. We shall verify the code of `foo` in a forward manner, and would compute a program state for each of its program point. Each program state, Φ, is a formula on the state of variables and heap. Each program state can be combined with a status and is represented by (Φ, τ) where τ denotes a status value from our lattice. As part of compositional verification, the precondition of each callee is checked against the current calling context and the postcondition is checked at the exit of the method's body. In the example, we can identify four program states of interests that correspond to four exits (L1, L2, L3 and L4) of the method. The following illustrates how the statuses are decided at exits through proof obligations discharged for postcondition checking with the help of the entailment procedure $\vdash$ that conforms to our error calculus. Given a program state π_a and a *post-condition* π_c, we can determine the status s for such checking with the help of the following lt

```
int foo(int x, int y)
 requires x≥0
 ensures (res>0)√; {
  if (x<0) return −1; /*L1*/
  else{
    if (y>1) return 1; /*L2*/
    else if (y<0) return −1; /*L3*/
    else return y; /*L4*/
  }}
```

Figure 3-4: Verifying *foo* Method with Error Calculus.

judgment: $\pi_a \vdash_p \pi_c \rightsquigarrow s$. The resulting statuses generated by the entailment procedure are as follows:

$$
\begin{array}{lll}
\texttt{L1}: & x{\geq}0 \wedge x{<}0 \wedge \texttt{res}{=}{-1} & \vdash \texttt{res}{>}0 \rightsquigarrow \bot \\
\texttt{L2}: & x{\geq}0 \wedge \neg(x{<}0) \wedge y{>}1 \wedge \texttt{res}{=}1 & \vdash \texttt{res}{>}0 \rightsquigarrow \surd \\
\texttt{L3}: & x{\geq}0 \wedge \neg(x{<}0) \wedge \neg(y{>}1) \wedge y{<}0 \wedge \texttt{res}{=}{-1} & \vdash \texttt{res}{>}0 \rightsquigarrow \mho \\
\texttt{L4}: & x{\geq}0 \wedge \neg(x{<}0) \wedge \neg(y{>}1) \wedge \neg(y{<}0) \wedge \texttt{res}{=}y & \vdash \texttt{res}{>}0 \rightsquigarrow \top
\end{array}
$$

Each of the above proofs yields a status based on the outcome of its entailment. This status can be added to program state for each of these program points. At L1, the antecedent is unsatisfiable which corresponds to an unreachable scenario (either infinite loop[2] or dead code) that can be captured by $(\texttt{false}, \bot)$ with `false` denoting contradiction at that program point. At L2, the consequent can be directly proven using the antecedent. This yields a valid program state that can be represented by $(x{\geq}0{\wedge}\neg(x{<}0){\wedge}y{>}1{\wedge}\texttt{res}{=}1, \surd)$. This program state indicates that the method will exit safely at this location with $\texttt{res}{=}1$. At L3, the negation of the consequent can be proven from its antecedent. The program state at L3 can be computed to be a must error as $(x{\geq}0{\wedge}\neg(x{<}0){\wedge}y{<}0, \mho)$. The sub-formula on

[2]Although we provide a mechanism to specify infinite loop, proving termination is beyond the scope of this paper.

result `res`$=-1$ is dropped since we have a must error outcome where the output state is unimportant. At L4, the antecedent can neither prove the consequent nor its negation. Hence, we would need to classify this program point as a may error whose state is $(x{\geq}0{\wedge}\neg(x{<}0) \wedge \neg(y{>}1) \wedge \neg(y{<}0) \wedge \texttt{res}{=}y, \top)$. A formula on result `res`$=y$ is still captured since the $\top$ status includes possibly safe output.

When an entailment checking fails, an error message is generated with relevant information to help debugging process. For example, the error message at $\mathtt{L_3}$ is:

```
Verify method foo.  Proving postcondition fails:
Failure (must):
```
$(x{\geq}0, 2) \wedge (\neg(x{<}0), 5) \wedge (\neg(y{>}1), 6) \wedge (y{<}0, 7) \wedge (\texttt{res}{=}{-1}, 7) \vdash (\texttt{res}{>}0, 3)$

where irrelevant formulas are sliced away and failures are localized by pairs of the relevant failing formulas and their corresponding statement code or specification line numbers.

3.3 Complete Specification Mechanism

$$\begin{array}{lcl}
\Psi & ::= & \{\Phi_1, \ldots, \Phi_n\} \\
\Re & ::= & \{\Phi'_1, \ldots, \Phi'_n\} \\
\Phi' & ::= & \bigvee\ (\exists w^* {\cdot} (\kappa{\wedge}\pi)\tau)^* \\
\tau & ::= & \bot \mid \mho \mid \surd \mid \top
\end{array}$$

Figure 3-5: Complete Specification Language

In our extended specification, we add a status value to separation formula Φ' (as defined in Fig.3-5) to help specify if we are expecting either valid or error scenario. The status captured by (τ) can be from the 4-point lattice domain introduced in section 3.2.1. This logic provided can be used to write Hoare-style (pre/post) specifications for each method of a program. Figure 3-6 provides a syntactic description of our extended specification, an extension of the structured specification presented [60]. Y denotes the (pre/post) specification while Φ denotes the formulas that may be used for pre/post specifications, as well as for predicate definitions. The *requires* keyword introduces a precondition. The postcondition is captured after each *ensures* keyword, which must appear as a terminating branch (or leaf) for the tree-like specification format. The postcondition may then include explicit error scenarios represented by the formula Φ'.

$$\begin{array}{lll} \mathsf{Y} & ::= \texttt{case}\{\pi_1 \Rightarrow \mathsf{Y}_1; \ldots; \pi_n \Rightarrow \mathsf{Y}_n\} & \text{case analysis} \\ & \mid \texttt{requires}\ \Phi\ \mathsf{Y} & \text{precond} \\ & \mid \texttt{ensures}\ \ \Phi' & \text{postcond} \end{array}$$

Figure 3-6: Complete Pre/Post Specifications

The extended specification mechanism will be further clarified through the examples as follows.

We shall now look at how (and why) specifications with explicit error scenarios can be written for pointer-based programs. Consider a simple example to sum up some values from two pointer locations.

```
int hoo(node x, node y)
{ return x.val + y.val; }
```

One simple specification for this method is the following:

$$\begin{array}{l} \texttt{requires}\ \mathtt{x}{\mapsto}\texttt{node}(\mathtt{a},\_) * \mathtt{y}{\mapsto}\texttt{node}(\mathtt{b},\_) \\ \texttt{ensures}\ (\mathtt{res}{=}\mathtt{a}+\mathtt{b})\surd; \end{array}$$

Though sound, this safety specification cannot be used to help the error calculus evaluate must/may bugs; the given precondition does not satisfy the dichotomy property, since there is another scenario with heap state $\mathtt{x}{\mapsto}\mathtt{node}(\mathtt{a}, \_){\wedge}\mathtt{x}{=}\mathtt{y}$ that also lead to a valid $\surd$ outcome. If we wish to obtain a complete safety precondition, we would have to add this extra valid scenario into our specification, as follows:

```
case {
  x=y  →  requires x↦node(a, _)
          ensures (res=2 * a)√;
  x≠y  →  requires x↦node(a, _)*y↦node(b, _)
          ensures (res=a + b)√ ;
}
```

With this more complete specification, all potential valid states are now captured by its precondition. Any contradiction with the precondition of this specification now represents a must error. However, specification with complete safety precondition can only be proven, if there is a way (either automatic or manual) to derive the negation of its precondition. As the negation operator are undecidable in many domains, including separation logic, there is not a general method for verifying that the code satisfies any given complete safety specifications. Our solution to this problem is to encourage users to write a more comprehensive specifications with explicit error scenarios. For our example, we could provide the following specification that fully captures all valid and error scenarios as shown in Figure 3-7.

3.4 A Calculus on Errors

In this section, we initially formalize the calculus with pure (without heap) formulas π. The extension of the calculus to heap formulas will be presented in

```
case {x=NULL∨y=NULL  →  ensures (true)℧;
      x≠NULL∧y≠NULL →
      case {
          x=y  →  requires x↦node(a,_)
                  ensures (res=2 ∗ a)√;
          x≠y  →  requires x↦node(a,_)∗y↦node(b,_)
                  ensures (res=a + b)√ ;
      }
}
```

Figure 3-7: Complete Specification Example.

the next section.

3.4.1 The Entailment Procedure

Given a program state π_a and a *postcondition state* π_c, we can determine the status s for such checking with the help of the following judgment: $\pi_a \vdash_p \pi_c \leadsto s$. The basic machinery for the judgment $\pi_a \vdash_p \pi_c \leadsto s$ is captured by the following four rules. We use underlying theorem solvers for answering sastifiability. Note that $\textbf{unsat}(\pi)$ denotes that π is unsatisfiable and $\textbf{sat}(\pi)$ denotes that π is satisfiable .

$$\underline{[\textbf{EC}-[\textbf{BOTTOM}]]} \qquad\qquad \underline{[\textbf{EC}-[\textbf{OK}]]}$$

$$\frac{\textbf{unsat}(\pi_{\mathbf{a}})}{\pi_a \vdash_p \pi_c \leadsto \bot} \qquad\qquad \frac{\textbf{unsat}(\pi_{\mathbf{a}} \wedge \neg\pi_{\mathbf{c}})}{\pi_a \vdash_p \pi_c \leadsto \surd}$$

$$\underline{[\textbf{EC}-[\textbf{MUST}-\textbf{ERROR}]]} \qquad\qquad \underline{[\textbf{EC}-[\textbf{MAY}-\textbf{ERROR}]]}$$

$$\frac{\textbf{unsat}(\pi_{\mathbf{a}} \wedge \pi_{\mathbf{c}})}{\pi_a \vdash_p \pi_c \leadsto \mho} \qquad\qquad \frac{\textbf{sat}(\pi_{\mathbf{a}} \wedge \neg\pi_{\mathbf{c}}) \quad \textbf{sat}(\pi_{\mathbf{a}} \wedge \pi_{\mathbf{c}})}{\pi_a \vdash_p \pi_c \leadsto \top}$$

Two rules at the first line check the success of the entailment and classify it as unreachable ($\bot$) or valid ($\surd$) as usual (checking $\textbf{unsat}(\pi_{\mathbf{a}} \wedge \neg\pi_{\mathbf{c}})$ is equivalent to checking $\pi_a \Longrightarrow \pi_c$). Next two rules at the second line check and classify the must/may error scenarios. In the first rule, a must error (invalid) is identified when $\pi_a \Longrightarrow \neg\pi_c$ is provable. [3] Due to the imprecision, entailments which has not been proven (as either valid or invalid) are marked with unknown status through the second rule.

To illustrate this entailment procedure, let us consider a postcondition check, $x{\geq}0$, under four different program states, as shown below.

$$\begin{array}{llll@{\qquad}llll} x{\leq}{-1}{\wedge}x{=}0 & \vdash_p & x{\geq}0 \rightsquigarrow & \bot & x{\leq}{-1} & \vdash_p & x{\geq}0 \rightsquigarrow & \mho \\ x{>}0 & \vdash_p & x{\geq}0 \rightsquigarrow & \surd & \texttt{true} & \vdash_p & x{\geq}0 \rightsquigarrow & \top \end{array}$$

3.4.2 Structural Rules

We provide sound structural rules that would carry out the entailment proving process. These rules support error localization, separation entailment procedure and modular verification.

$$\frac{\begin{array}{c}[\underline{\textbf{PEC}-[\sqcup\ \textbf{JOIN}]}]\\ \pi_1 \vdash_p \pi \rightsquigarrow \tau_1 \\ \pi_2 \vdash_p \pi \rightsquigarrow \tau_2\end{array}}{\pi_1{\vee}\pi_2 \vdash_p \pi \rightsquigarrow \tau_1{\sqcup}\tau_2} \qquad \frac{\begin{array}{c}[\underline{\textbf{PEC}-[\otimes\ \textbf{COMPOSE}]}]\\ \pi \vdash_p \pi_1 \rightsquigarrow \tau_1 \\ \pi \vdash_p \pi_2 \rightsquigarrow \tau_2\end{array}}{\pi \vdash_p \pi_1{\wedge}\pi_2 \rightsquigarrow \tau_1{\otimes}\tau_2} \qquad \frac{\begin{array}{c}[\underline{\textbf{PEC}-[\oplus\ \textbf{SEARCH}]}]\\ \pi \vdash_p \pi_1 \rightsquigarrow \tau_1 \\ \pi \vdash_p \pi_2 \rightsquigarrow \tau_2\end{array}}{\pi \vdash_p \pi_1{\vee}\pi_2 \rightsquigarrow \tau_1{\oplus}\tau_2}$$

These rules use the algebraic operations presented in Sec. 3.2 to combine the results. The first rule decomposes disjunction on the antecedent, while the second rule decomposes conjunction on the consequent. Both these rules can be implemented without any loss of information. The third rule performs a search over a disjunction in the consequent. This search returns a set of possible proofs for the entailment. According to the $\oplus$ operator, if at least one $\surd$ status is found in this solution set, the entailment succeeds.

Theorem 1 (Soundness of the Structural Rules)**.** *Given an entailment $\pi_1{\vdash_p}\pi_2$. If the application of the structural rules* $[\underline{\textbf{PEC}-[...]}]$ *on the given antecedent π_1 and*

[3]Without loss of generality, we assume that the $[\underline{\textbf{EC}-[\textbf{BOTTOM}]}]$ rule has always been checked before $[\underline{\textbf{EC}-[\textbf{MUST}-\textbf{ERROR}]}]$. Thus, when the latter rule is checked, π_a should be satisfiable.

consequent π_2 *returns the result* τ*, then the application of the* $[\underline{\textbf{EC}-[...]}]$*rules on the given antecedent* π_1 *and consequent* π_2 *returns the same result* τ*, namely* $\pi_1 \vdash_p \pi_2 \rightsquigarrow \tau$*.*

Proof The proof is by an induction on structural rules $[\underline{\textbf{PEC}-[...]}]$ and a case analysis on the returned result τ. As an illustration, suppose that the structural rule $[\underline{\textbf{PEC}-[\otimes\ \textbf{COMPOSE}]}]$ using entailment $\vdash$ is applied on the antecedent π_1 and the consequent $\pi'_2 \wedge \pi''_2$ returning $\mho$. According to the $\otimes$ operator rules, we can have either $\pi_1 \vdash_p \pi'_2 \rightsquigarrow \mho$ or $\pi_1 \vdash_p \pi''_2 \rightsquigarrow \mho$ or both are $\mho$. Assuming the former case is taken, and then using the rule $[\underline{\textbf{EC}-[\textbf{MUST}-\textbf{ERROR}]}]$ we have $(\textbf{unsat}(\pi_1 \wedge \pi'_2))$. Hence we can deduce that $\textbf{unsat}(\pi_1 \wedge \pi'_2 \wedge \pi''_2)$ is valid, and then using $[\underline{\textbf{EC}-[\textbf{MUST}-\textbf{ERROR}]}]$ we get $\pi_1 \vdash_p \pi'_2 \wedge \pi''_2 \rightsquigarrow \mho$. □

We present full proof of this theorem in Appendix .1.

3.4.3 Error Localization Extension to Calculus

$$
\begin{array}{lll}
\tau[m] & ::= & \bot[\emptyset] \mid \mho[m] \mid \surd[m] \mid \top[m] \\
m & ::= & bm \mid m_1 \sqcup m_2 \mid m_1 \otimes m_2 \mid m_1 \oplus m_2 \\
bm & ::= & \pi_1 \Longrightarrow \pi_2 \ (\texttt{valid}) \\
 & \mid & \pi_1 \Longrightarrow \overline{\pi_2} \ (\texttt{must error}) \\
 & \mid & \pi_1 \Longrightarrow \underline{\pi_2} \ (\texttt{may error})
\end{array}
\qquad
\begin{array}{rcl}
\tau_1[m_1] \diamond \tau_2[m_2] & \Rightarrow & (\tau_1 \diamond \tau_2)[m_1 \diamond m_2] \\
m \diamond \emptyset & \Rightarrow & m \\
\emptyset \diamond m & \Rightarrow & m \\
\bot[m] & \Rightarrow & \bot[\emptyset]
\end{array}
$$

Figure 3-8: Program State: Status and Message

To provide support for error localization, we must extend the four-point lattice with messages that capture the reason for each success or failure (see the left of Figure 3-8).

Status $\bot$ does not carry any message which is denoted by $\emptyset$. When faced with a message with error from $m_1 \sqcup m_2$ and $m_1 \otimes m_2$, both of the two smaller messages (with possible errors), denoted by m_1 and m_2, must be resolved, before the main message is said to be resolved. When faced with a message with error of the form $m_1 \oplus m_2$, only one of the messages with errors from either m_1 or m_2 needs to be resolved, before the main message $m_1 \oplus m_2$ is resolved. We may now modify the three operators $\sqcup$, $\otimes$ and $\oplus$, to propagate messages capturing the localizations for successes and failures. Let us denote this using a generic name $\diamond$ for the three operators. We propagate every message, where possible, as shown to the right of Fig. 3-8. In case empty message $\emptyset$ is generated, we remove it from the main message as shown in the second and third rules. In case the resulting status from $\tau_1 \diamond \tau_2$ is $\bot$, we remove its messages, as shown in the last rule.

3.5 Error Calculus for Separation Logic

While we have formulated a must/may error calculus for pure logic, we shall now extend it to separation logic. In this section, we show how our calculus can be used to support the reasoning of pointer-based programs via separation logic [116]. Separation logic with user-defined predicates can provide concise and precise notations for verifying programs with complex data structures. We show how our calculus can be used to support the reasoning of pointer-based programs via the fragment of separation logic presented in Chapter 2.

3.5.1 Separation Entailment with Proof Search

To support proof search, we enhance the entailment procedure for separation logic presented in section 2.3 as follows:

$$\Phi_1 \vdash^{\kappa}_{V} \Phi_2 \rightsquigarrow (\Psi, \tau)$$

whereby Ψ captures a set of residual program states with status information. We use a set of program states (Ψ) since our entailment procedure may have to conduct a proof search with the help of lemmas. The variable κ captures the current set of heap nodes (and predicates) that have been consumed (or accounted) by the

$$\frac{XPure(\kappa_1{*}\kappa){\wedge}\pi_1 \vdash_p \exists\, V{\cdot}\pi_2 \leadsto \tau}{\kappa_1{\wedge}\pi_1 \vdash^{\kappa}_{V} \pi_2 \leadsto (\kappa_1{*}\kappa{\wedge}\pi_1, \tau)}\;[\underline{\textbf{EENT-EMP}}]$$

$$\frac{\Phi_1 \vdash^{\kappa}_{V} \Phi_2 \leadsto (\Phi_0, \tau_0) \quad \Phi^i_{1/2} = \texttt{APP}(\texttt{LEM}^{\texttt{i}}, \Phi_{1/2})\;\; \Phi^i_1 \vdash^{\kappa}_{V} \Phi^i_2 \leadsto (\Phi^i, \tau^i)}{\Phi_1 \vdash^{\kappa}_{V} \Phi_2 \leadsto (\oplus\{\Phi_0;\Phi^i\}, \oplus\{\tau_0;\tau^i\})}\;[\underline{\textbf{EENT-LEMMA}}]$$

$$\frac{\Phi_i \vdash^{\kappa}_{V} \Phi_3 \leadsto (\Psi_i, \tau_i)\;\; i \in \{1,2\}}{\Phi_1{\vee}\Phi_2 \vdash^{\kappa}_{V} \Phi_3 \leadsto (\bigvee \Psi_i, \sqcup\tau_i)}\;[\underline{\textbf{EENT-LHS-OR}}]
\qquad
\frac{\Phi_1 \vdash^{\kappa}_{V} \Phi_i \leadsto (\Psi_i, \tau_i)\;\; i \in \{2,3\}}{\Phi_1 \vdash^{\kappa}_{V} (\Phi_2{\vee}\Phi_3) \leadsto (\bigcup \Psi_i, \oplus\tau_i)}\;[\underline{\textbf{EENT-RHS-OR}}]$$

$$\frac{[w/v]\Phi_1 \vdash^{\kappa}_{V} \Phi_2 \leadsto (\Phi_i, \tau) \quad \textit{fresh } w}{\exists v \cdot \Phi_1 \vdash^{\kappa}_{V} \Phi_2 \leadsto (\Phi_i, \tau)}\;[\underline{\textbf{EENT-LHS-EX}}]
\qquad
\frac{\Phi_1 \vdash^{\kappa}_{V\cup\{w\}} ([w/v]\Phi_2) \leadsto (\Phi_i, \tau) \quad \textit{fresh } w \quad \Phi_j{=}\exists\, w \cdot \Phi_i}{\Phi_1 \vdash^{\kappa}_{V} (\exists\, v \cdot \Phi_2) \leadsto (\Phi_j, \tau)}\;[\underline{\textbf{EENT-RHS-EX}}]$$

Figure 3-9: Separation Entailment with Set Outcomes

entailment procedure, while V captures existential variables from the consequent (that may be instantiated). If the antecedent semantically entails the consequent, the entailment succeeds and we expect status τ to be set to $\surd$. Otherwise, the entailment fails and we expect τ to be set to either $\mho$ or $\top$.

To simplify our presentation, only the five structural rules and a base case (or non-recursive) rule, are shown in Figure 3-9. In addition, our algorithm also performs matching of heap nodes (and predicates), unfolding (of a predicate in the antecedent), or folding (of a predicate in the consequent with support for variable instantiation). These steps are automatically applied, and are now standard features for entailment procedures that are designed to support automated program verification with user-defined predicates.

The final step prior to a successful entailment is the $[\underline{\textbf{EENT-EMP}}]$-rule that is only triggered when the consequent is just a pure formula. To support error calculus, we must modify (or implement) this rule as described.Reduce entailment checking on separation logic to implication checking on the first order-logic with $[\underline{\textbf{EENT-EMP}}]$ rule. When the consequent remains empty heap, e.g. $\texttt{emp} \wedge \pi_c$, the procedure employs [$\underline{\textbf{EENT−EMP}}$] inference rule to decide the entailment result. Firstly, this rules make use of the **eXPure** reduction to transform the combination of remain heaps in the antecedent and footprints into the first order-logic formula on the combination of pure domains, e.g. π_a. Then it checks the implication $\pi_a \vdash_p \pi_c$.

Note that $\oplus$ is overloaded. When it is applied for a set of the status, it is defined as $\oplus\{\tau_1, \tau_2, \ldots, \tau_n\} = (\ldots((\tau_1 \oplus \tau_2) \oplus \tau_3) \ldots \tau_n)$;[4] when it is applied for a set of residual states, it will search for all successful states (and will remove all the failed states). Finally, $\bigvee$ is a lifted disjunction operation and is defined as $\Psi_1 \vee \Psi_2 = \{\Phi_1 \vee \Phi_2 | \Phi_1 \in \Psi_1 \wedge \Phi_2 \in \Psi_2\}$.

Furthermore, we must extend our entailment procedure in the following ways:

- The rule [$\underline{\textbf{EENT−LEMMA}}$] illustrate how the calculus supports lemma application [113]. This rule expresses the possibility of lemma application in LHS and RHS. The status values of all possible lemma applications are combined by the union operator ($\oplus$) where $\mho$ takes priority over $\top$. Hence, if a proof search attempt fails, we return a $\mho$ status, rather than a $\top$ status since the latter prevents a $\mho$ failure from being reported, even if they can be confirmed by a different proof search.

- When our entailment procedure becomes stuck with a non-empty consequent (comprising some heap predicates) we shall firstly determine a pure approximation of the consequent for both heap and pure data. We may then determine if there is any contradiction with the antecedent to decide whether must or may failure is going to be reported.

[4] $\oplus$ is the search operator as described in Sec. 3.2.1.

3.5.2 Examples on Separation Entailment

Let us examine four simple examples to better understand how status outcome is being determined by the entailment procedure of separation logic.

Example 1

$$x{\mapsto}\texttt{node}(\_,q) * q{\mapsto}\texttt{node}(\_,\texttt{NULL}) \vdash x{\mapsto}\texttt{node}(\_,p)$$
$$\leadsto (q{\mapsto}\texttt{node}(\_,\texttt{NULL}) \wedge p{=}q \wedge x{\neq}\texttt{NULL}, \surd)$$

This entailment yields a residue $q{\mapsto}\texttt{node}(\_,\texttt{NULL})$ and an instantiation $p{=}q$ from (implicit) existential variable p. It also carries a pure formula $x{\neq}\texttt{NULL}$ from the antecedent.

Example 2

$$x{\mapsto}\texttt{node}(\_,q) * q{\mapsto}\texttt{node}(\_,\texttt{NULL}) \vdash x{\mapsto}\texttt{node}(\_,\texttt{NULL})$$
$$\leadsto (\texttt{true}\,, \mho)$$

This entailment yields a must failure, denoted by $\mho$. The consequent expects $q{=}\texttt{NULL}$, but the antecedent had $q{\mapsto}\texttt{node}(\_,\texttt{NULL})$. This contradiction has caused a $\mho$ failure to be raised. The residue captures the state when failure was detected.

Entailment 3

$$x{\mapsto}\texttt{node}(\_,q) * q{\mapsto}\texttt{node}(\_,\texttt{NULL}) \vdash x{\mapsto}\texttt{node}(3,p)$$
$$\leadsto (\texttt{true}\,, \top)$$

This entailment yields a may failure, denoted by $\top$. The consequent expects value 3 to be proven as the data field of x. However, the antecedent has no information on that field position. Hence, a $\top$ failure was raised.

Entailment 4

$$x{\mapsto}\texttt{node}(\_,\_) * y{\mapsto}\texttt{node}(\_,\_) \vdash x{=}y$$
$$\leadsto (\texttt{true}\,, \mho)$$

This entailment yields a must failure. The consequent expects value $x{=}y$ to be proven. However, the antecedent has x and y pointing to disjoint memory. This contradiction has caused a $\mho$ failure to be raised.

To handle linked-list of arbitrary length, we could use the following list segment predicate:

$$\begin{array}{l}\texttt{pred lseg}\langle\texttt{root},\texttt{n},\texttt{p}\rangle\equiv\texttt{root}{=}\texttt{p}\wedge\texttt{n}{=}0\\ \quad\vee\ \exists\,\texttt{d},\texttt{q}\cdot(\texttt{root}{\mapsto}\texttt{node}\langle\texttt{d},\texttt{q}\rangle{*}\ \texttt{lseg}\langle\texttt{q},\texttt{n}{-}1,\texttt{p}\rangle)\\ \quad\textbf{inv}\,\texttt{n}{\geq}0\end{array}$$

In addition, we declare `plseg` predicate that defines a list segment with only positive integers:

$$\begin{array}{l}\texttt{pred plseg}\langle\texttt{root},\texttt{n},\texttt{p}\rangle\equiv\\ \qquad\exists\,\texttt{d}\cdot(\texttt{root}{\mapsto}\texttt{node}\langle\texttt{d},\texttt{p}\rangle\wedge\texttt{n}{=}1\wedge\texttt{d}{>}0)\\ \quad\vee\ \exists\,\texttt{d},\texttt{q}\cdot(\texttt{root}{\mapsto}\texttt{node}\langle\texttt{d},\texttt{q}\rangle{*}\ \texttt{plseg}\langle\texttt{q},\texttt{n}{-}1,\texttt{p}\rangle\wedge\texttt{d}{>}0)\\ \quad\textbf{inv}\,\texttt{n}{\geq}1\end{array}$$

We now examine the two following separation entailments involving user-defined recursive predicates.

Example 5:

$$\begin{array}{l}\texttt{plseg}\langle\texttt{x},2,\texttt{NULL}\rangle\\ \qquad\vdash\ \texttt{x}{\mapsto}\texttt{node}(\texttt{a},\texttt{y})\ *\ \texttt{y}{\mapsto}\texttt{node}(\texttt{b},\texttt{q})\wedge\texttt{a}{+}\texttt{b}{<}0\end{array}$$

After unfolding on the antecedent followed by a matching, the entailment is:

$$\begin{array}{l}\exists a',z\cdot\texttt{plseg}\langle\texttt{z},1,\texttt{NULL}\rangle\wedge x\neq\texttt{NULL}\wedge a'=a\wedge z{=}y\wedge a'{>}0\\ \qquad\vdash\ \texttt{y}{\mapsto}\texttt{node}(\texttt{b},\texttt{q})\wedge\texttt{a}{+}\texttt{b}{<}0\end{array}$$

Again we do unfold on the antecedent followed by a matching, causing a consequent with no heap. We then apply the [**EENT−EMP**] rule, the following pure entailment is checked (for simplicity, we have omitted superfluous equalities):

$$\texttt{x}{\neq}\texttt{NULL}{\wedge}\texttt{a}{>}\texttt{0} \wedge \texttt{z}{=}\texttt{y}{\wedge}\texttt{y}{\neq}\texttt{NULL}{\wedge}\texttt{b}{>}\texttt{0}{\wedge}\texttt{q}{=}\texttt{NULL} \vdash \texttt{a}{+}\texttt{b}{<}\texttt{0}$$

Obtain pure over-approximation and applying the calculus on pure entailment, the ℧ error is returned as follow:

$$\texttt{a}{>}\texttt{0}{\wedge}\texttt{b}{>}\texttt{0} \vdash \texttt{a}{+}\texttt{b}{<}\texttt{0} \ \leadsto \ \mho$$

<u>Example 6:</u>

$$\texttt{plseg}\langle \texttt{x}, 2, \texttt{NULL}\rangle \vdash \texttt{lseg}\langle \texttt{x}, 2, \texttt{q}\rangle$$

This entailment fails with mismatch error ($\tau_0{=}\mho$). To improve our analysis, user may provide the following lemma:

$$\texttt{lemma}\ \text{“}w_1\text{”}\ \texttt{plseg}\langle \texttt{root}, \texttt{n}, \texttt{p}\rangle \rightarrow \texttt{lseg}\langle \texttt{root}, \texttt{n}, \texttt{p}\rangle$$

The procedure then applies the lemma w_1 to weaken a `plseg` (to become a `lseg`). With the help of the lemma $\texttt{w}_1$, another possible solution for proof search is to first apply the weakening lemma on `x`. It is then easy to prove ($\tau^1{=}\surd$) by applying a matching ([**ENT−PRED−MATCH**] in Section 2.3) followed by the [**EENT−EMP**] rule. Finally, we apply the [**EENT−LEMMA**] rule to combine the program status and residue by the search ($\oplus$) operator which results in program status: $\tau{=}\tau_0{\oplus}\tau^1{=}\mho{\oplus}\surd{=}\surd$.

3.5.3 Entailment with Contradiction Lemma

In separation logic, it is not clear how to define negation for inductive predicates of complex data structures. In this section, we address this by providing a new mechanism to support the detection of must errors at the predicate level: we propose a new scheme for entailment with contradiction lemmas. We allow users to declare contradiction lemmas of the following form:

$$\texttt{LHS} \rightarrow (\texttt{RHS})\mho$$

These lemmas can help detect contradiction at the predicate level. Semantically, if the above contradiction lemma can be proven, we can confirm $\texttt{LHS} \Longrightarrow \neg(\texttt{RHS})$. Operationally, the contradiction lemma is proven through the following separation logic entailment:

$$\texttt{LHS} \vdash \texttt{RHS} \leadsto (\Phi)\mho.$$

Application of this contradiction lemma causes a must error to be detected at the predicate level. Our lemmas must be supplied by users, but will be automatically proven and applied by our system.

For illustration, consider the `plseg` defined in the previous subsection, and its negation `nplseg`, which contains at least one non-positive integer.

$$\begin{array}{l}
\texttt{pred}\ \texttt{nplseg}\langle\texttt{root},\texttt{n},\texttt{p}\rangle\equiv\ \texttt{root}{=}\texttt{p}\wedge\texttt{n}{=}\texttt{0} \\
\quad\vee\ \exists\,\texttt{d},\texttt{q}\cdot(\texttt{root}{\mapsto}\texttt{node}\langle\texttt{d},\texttt{q}\rangle{*}\ \texttt{lseg}\langle\texttt{q},\texttt{n}{-}\texttt{1},\texttt{p}\rangle{\wedge}\texttt{d}{\leq}\texttt{0}) \\
\quad\vee\ \exists\,\texttt{d},\texttt{q}\cdot(\texttt{root}{\mapsto}\texttt{node}\langle\texttt{d},\texttt{q}\rangle{*}\ \texttt{nplseg}\langle\texttt{q},\texttt{n}{-}\texttt{1},\texttt{p}\rangle) \\
\quad\textbf{inv}\ \texttt{n}{\geq}\texttt{0}
\end{array}$$

To capture their relation, we provide the following contradiction lemma:

$$\texttt{lemma}\ \text{“}w_2\text{”}\ \texttt{nplseg}\langle\texttt{root},\texttt{n},\texttt{p}\rangle\ {\rightarrow}\texttt{plseg}\langle\texttt{root},\texttt{n},\texttt{p}\rangle\mho$$

When invoked, each application of such lemmas causes a must error to be triggered for the particular branch of the entailment proof. For example, consider the `list_sqrt_aux` procedure shown in Figure 3-10. The method `list_sqrt_aux` is an auxilary method which helps to compute square roots of numbers from a list. It computes square root of the current node `y`, if applicable, by function `sqrt` at line 6 and inserts the result at the beginning of the processed positive list segment `x`. We provide a specification to capture its

```
1   node list_sqrt_aux(node x, node y)
2     requires plseg⟨x, n, y⟩ ∗ y↦node⟨v, p⟩
3     case {
4         v>0 → ensures (plseg⟨res, n + 1, p⟩) √;
5         v≤0 → ensures (true ) ℧;
      }
    {
6     if (y.val > 0)    y.val = sqrt_int(y.val);
7     node tmp=y.next;
8     y.next=x;
9     x.last().next=tmp;
10    return y;
    }
```

Figure 3-10: Code of `list_sqrt_aux` Method.

correctness. Line 4 is for soundness of the method, while line 5 is for its completeness, so as to capture the remaining cases. Intuitively, the two scenarios in the specification correspond to two path traces of the *conditional* statement. We focus on the (implicit) *else* path trace which leads to an error scenario. At line 10, the condition of this trace is transformed to be a program state which is checked against the specification. Its proof obligation is generated as follows:

$$\mathtt{y}\mapsto\mathtt{node}\langle\mathtt{v},\mathtt{x}\rangle * \mathtt{plseg}\langle\mathtt{x},\mathtt{n},\mathtt{p}\rangle\wedge\neg(\mathtt{v}{>}0) \vdash\ \mathtt{plseg}\langle\mathtt{y},\mathtt{n}+1,\mathtt{p}\rangle$$

Since $\mathtt{v}\leq 0$, `y` cannot be matched in both sides and this entailment proof fails as a must error. Therefore, the specification at line 5 cannot be properly proven.

To improve our analysis, we now apply the lemma $\mathtt{w}_2$. With the help of the lemma $\mathtt{w}_2$, another possible solution for proof search is to first apply the weakening lemma on `x`. After that, `y` can now be folded to become a `nplseg` based on the definition of the `nplseg` predicate. Lastly, applying the contradiction lemma on `y` makes this entailment proof return a must error as (ideally) expected.

Nguyen et. al. [113] proposed a lemma mechanism to relate user-defined predicates. While they focus on subsumed and equivalent relations, we focus on disjoint relations (contradiction). Nevertheless, both enhance the completeness of program verification.

3.6 Modular Verification with Error Calculus

Code verification is typically formalised using Hoare triples of the form $\{pre\}c\{post\}$, where *pre*, *post* are the initial and final states of program code c. We would like to remind that to incorporate status into our program state, we shall use disjunctive program state of form $\bigvee(\Phi, \tau)$, giving us a new Hoare triple of the form $\{\bigvee(\Phi_1, \tau_1)\}\ c\ \{\bigvee(\Phi_1, \tau_1)\}$. To simplify our presentation, we shall use (Φ, τ) instead of the more general disjunctive program state $\bigvee(\Phi, \tau)$ that had been implemented.

Method Verification Rule. The verification requirement for methods can be affected by progressively accumulating the preconditions from the structured specification, prior to the verification of its method body. The forward verification rules are presented in Figure 3-11. The verification for each method is done by $[\underline{\textbf{FV}-[\textbf{METH}]}]$ rule. This rule uses a initial program state $\bigwedge(v' = v)^*$ that the current values of program variables are the same as their original parameters' values. In the $[\underline{\textbf{FV}-[\textbf{M}-\textbf{CASE}]}]$ rule, each branch is processed separately and their results are combined by join operations: $\bigvee$ (as defined in previous section) for program states and $\sqcup$ for status of program states. Finally, verification process is terminated by $[\underline{\textbf{FV}-[\textbf{M}-\textbf{ENSURES}]}]$ rule. In this rule, the status returned from the entailment proving of postcondition will be composed $(\otimes)$ with the status of code verification to become τ_4. τ_4 is then matched (by *inv* function) with τ. The function *inv* is implemented as follows: (1) if this is a safety proving ($\tau = \surd$), then *inv* returns τ_4; (2) if this is an error verified ($\tau = \mho$ or $\tau = \top$), then *inv* returns $\surd$ in the cases of $\tau_4 = \mho$ (or $\tau_4 = \top$, respectively), otherwise it returns $\mho$.

Method Call Rule. The abstract semantics of each method call is captured by

$$
\begin{array}{cc}
\begin{array}{c}
\left[\underline{\mathbf{FV{-}[METH]}}\right] \\
\dfrac{\begin{array}{c}\Phi_1 = \bigwedge(v' = v)^* \\ \vdash \{(\Phi_1, \surd)\}\ \{\mathsf{Y}\ c\}\ \{\Phi_2, \tau_2\}\end{array}}{\vdash t_0\ mn\ ([\mathit{ref}]\ t\ v)^*\ \mathsf{Y}\ \{c\}(\Phi_2, \tau_2)}
\end{array}
&
\begin{array}{c}
\left[\underline{\mathbf{FV{-}[M{-}ENSURE]}}\right] \\
\dfrac{\begin{array}{c}\vdash \{(\Phi_1, \tau_1)\}\ \{c\}\ \{\Phi_2, \tau_2\} \\ \Phi_2 \vdash^{\kappa}_{V} \Phi \rightsquigarrow (\Phi_3, \tau_3) \quad \tau_4 = \tau_2 \otimes \tau_3\end{array}}{\vdash \{(\Phi_1, \tau_1)\}\{\texttt{ensures}\ (\Phi, \tau)\ c\}(\Phi_3, \mathit{inv}(\tau, \tau_4))}
\end{array}
\\[2em]
\begin{array}{c}
\left[\underline{\mathbf{FV{-}[M{-}REQUIRES]}}\right] \\
\dfrac{\vdash \{(\Phi_1 * \Phi, \tau_1)\}\ \{\mathsf{Y}\ c\}\ \{\Phi_2, \tau_2\}}{\vdash \{(\Phi_1, \tau_1)\}\{\texttt{requires}\ \Phi\ \mathsf{Y} c\}(\Phi_2, \tau_2)}
\end{array}
&
\begin{array}{c}
\left[\underline{\mathbf{FV{-}[M{-}CASE]}}\right] \\
\dfrac{\vdash \{(\Phi_1 \wedge \pi_i, \tau_1)\}\ \{\mathsf{Y}\ c\}\ \{\Phi_i, \tau_i\}\ i = 1 \ldots n}{\vdash \{(\Phi_1, \tau_1)\}\{\texttt{case}\{\pi_i \Rightarrow \mathsf{Y}_i\}^*\}(\bigvee \Phi_i, \sqcup \tau_i)}
\end{array}
\end{array}
$$

Figure 3-11: Forward Verification Rules.

its specifications. We encode its verification with the rule $\left[\underline{\mathbf{FV{-}[CALL]}}\right]$. Note that $(\texttt{t v})^*$ and $(\texttt{ref t u})^*$ denote *pass-by-value* and *pass-by-reference* parameters, respectively. Each method call $\texttt{mn}(\texttt{v}^*, \texttt{u}^*)$ in our core language has only variables as arguments. To avoid the need for argument substitutions, we assume that each method declaration from `Program` has been suitably renamed so that actual arguments are identical to the formal arguments.

$$
\begin{array}{c}
\left[\underline{\mathbf{FV{-}[CALL]}}\right] \\
\dfrac{\begin{array}{c}
t_0\ mn\ ((t\ v)^*, (\mathit{ref}\ t\ u)^*)\ \mathsf{Y}\ \{c\} \in \texttt{Program} \\
\Phi_1 \vdash \mathsf{Y} \rightsquigarrow (\Phi_2, \tau_2) \\
\Phi_R = \texttt{if}\ \tau_1{=}\surd\ \texttt{then}\ (\exists v'^* \cdot \Phi_2)\ \texttt{else}\ \Phi_1
\end{array}}{\{(\Phi_1, \tau_1)\}\ mn\ ((t\ v)^*, (\mathit{ref}\ t\ u)^*)\{(\Phi_R, \tau_1 \otimes \tau_2)\}}
\end{array}
$$

The proof obligations are generated and verified at the second line, provided that the incoming status τ_1 is $\surd$. Furthermore, output states from proving entailment are composed with status from pre-state at the third line. By default, if the caller context contains errors, such errors are simply propagated to the next instruction in a similar manner as exceptions. However, unlike exceptions, error states are never caught. To generate proof obligations for the extended specification, we propose to extend the entailment procedure to handle specification with separation formulas. The revised judgement has the form $\Phi_1 \vdash \mathsf{Y} \leadsto (\Phi_2, \tau_2)$, where Φ_1 is the current state, Y is the specification and (Φ_2, τ_2) is the residual state and its status. Three syntax-directed rules are extended. They are used to prove each precondition and assume its respective postcondition for the callee, as shown below:

$$\left[\underline{\textbf{FV}-[\textbf{C}-\textbf{REQUIRES}]}\right]$$
$$\frac{\Phi_1 \vdash \Phi \leadsto (\Phi_2, \tau_2) \qquad (\Phi_2) \vdash \mathsf{Y} \leadsto (\Phi_3, \tau_3)}{\Phi_1 \vdash \texttt{requires}\ \Phi\ \mathsf{Y} \leadsto (\Phi_3, \tau_2 \otimes \tau_3)}$$

$$\left[\underline{\textbf{FV}-[\textbf{C}-\textbf{CASE}]}\right] \qquad \left[\underline{\textbf{FV}-[\textbf{C}-\textbf{ENSURES}]}\right]$$
$$\frac{\Phi \wedge \pi_i \vdash \mathsf{Y}_i \leadsto (\Phi_i, \tau_i)\ i = 1 \ldots n}{\Phi \vdash \texttt{case}\{\pi_i \Rightarrow \mathsf{Y}_i\}^* \leadsto (\bigvee \Phi_i, \sqcup \tau_i)} \qquad \frac{\Phi_1 \vdash_p \texttt{true} \leadsto (\Phi, \tau_1)}{\Phi_1 \vdash \texttt{ensures}\ (\Phi_2)\tau_2 \leadsto (\Phi_1 * \Phi_2, \tau_1 \otimes \tau_2)}$$

3.7 Implementation and Experiments

We have implemented our error calculus inside a program verification system for separation logic, called S2. We use S2 to verify C-based programs against user-given specifications. The verification is performed compositionally for each method, and loops are transformed to recursive methods. S2 eventually translates separation logic proof obligations to pure formulas that can be discharged by different theorem provers. Our system uses Omega [125], MONA [85], Redlog [55] and Z3 [45] as underlying theorem provers for answering the satisfiability and simplification queries. When program code is not successfully verified against safety properties, S2 not only further classifies the failures into the must or may errors but also localizes program statements and specifications relevant to the errors.

Programs	Size		#P	Time(sec.)		Invo.(#)	
	LOC	LOS	#	wo	w	wo	w
Linked list	327	50	26	0.44	0.46	2738	3202
Linked list	157	27	13	0.58	0.6	1520	1724
Sorted llist	98	11	6	0.46	0.49	955	1060
Doubly llist	186	23	13	0.34	0.34	1864	2083
Doubly llist	91	13	5	0.5	0.5	1309	1429
CompleteT	106	12	5	0.87	0.94	2149	2533
Heap trees	179	13	5	1.9	1.91	4540	4954
AVL	313	27	12	3.44	3.59	7863	8585
AVL2	152	37	7	2.83	3	6959	7876
BST	177	18	9	0.35	0.37	1883	2192
BST	153	12	6	0.3	0.31	1581	1836
RBT	508	48	19	3.32	3.38	13069	16687
Bubble sort	75	9	4	0.21	0.21	1092	1254
Quick sort	82	10	4	0.27	0.28	778	832
Merge sort	109	11	6	0.47	0.5	1035	1074
Quick sort - queue	127	4	2	4.25	5.27	13218	21139
Total	2840	325	142	20.53	22.15	62553	78460

Table 3.1: Verification Performance with (*w*) and without (*wo*) Error Calculus

3.7.1 Calculus Performance for Heap-Based Programs

To evaluate the overheads of error calculus, we executed our system S2 twice, once *with* error calculus and a second time *without*, on a suite of bug-free pointer-based programs. We stress that although the sizes of these programs are fairly small, they deal with fairly complex heap-based data structures, such as linked lists, doubly-linked lists and AVL-trees. Therefore, these programs can be used to *fully* evaluate the performance of our calculus which has been embedded inside a separation logic prover. The results are summarized in Table 3.1. The first column contains the list of the verified programs and their proven properties while the second, third and fourth columns describe number of lines of code (LOC), number of lines of specification (LOS) and number of procedures in each program (#P). On average, LOS is around 12% of LOC and specifications are complicated enough to demonstrate the performance of our calculus. The fifth and sixth columns show the total verification time (in seconds) for the system S2 without and with error

Programs	LOC	LOS	#P.	#V.	#F.	$\mho$	$\top^1$	$\top^2$	LOE	time(s)
tcas	173	48	9	41	48	31	14	3	3.48	3.06
schedule2	374	108	16	10	10	5	0	3	3	8.25
schedule1a	412	50	18	10	16	15	0	1	4.38	18.13
schedule1b	413	50	18	9	8	7	0	1	4.25	32.29
replace	564	73	21	24	24	18	0	6	4.21	17.89
print_tokens2	570	64	19	10	10	7	0	1	4.88	20.42
print_tokens	726	87	18	7	9	8	0	1	3.67	6.73
Total/(Avr.)	3232	480	119	111	125	91	14	16	(3.98)	(15.25)

Table 3.2: Bugs finding & localizing with programs in the Siemens Test Suite

calculus, respectively. The last two columns capture the number of satisfiability and simplification queries sent to the provers for each experiment.

In Table 3.1, the results show that the total overhead introduced by our error calculus is around 1.62 seconds (8%). This overhead is proportional to the number of extra satisfiability and simplification queries shown in the last two columns. These experimental results have shown that must/may error calculus with messages can be supported with modest overhead.

3.7.2 Calculus Usability

In order to show the usability of our error calculus on bugs finding and localizing, we evaluated our system on the Siemens test suite [52] of programs. The test suite contains programs with complex data structures (e.g. linked lists, queues), arrays and loops. Each program in the suite has #P number of procedures, has one non-faulty version, v_0, and a number of seeded faulty versions (#V). column in Table 3.2) from v_1 to v_n. Each of these faulty versions has one or more (seeded) faults. Total number of faults is captured in #F column. These faulty versions are suitable for checking the ability of tools in finding bugs and localizing errors (as used in [83]).We provide specifications for each program such that S2 (1) successfully verifies safety (sound or complete requirements) in the non-faulty versions, and (2) captures potential must-bug errors that are complementary to the

safety scenarios. We emphasize that these specifications were designed primarily to verify safety scenarios *without* considering the faulty versions of each program. Nevertheless, S2 is able to utilize the same specification to find and explain the presence of bugs in the faulty versions, as elaborated below.
Table 3.2 shows the result of running our system on six programs from the suite. The properties our tool proved include: (i) memory safety (all), (ii) size of data structures (schedule1a, schedule1b and schedule2 program), (iii) array-related properties (tcas, print_token, print_token2 and replace program), (iv) functional arithmetic constraints between input and output (all). We are interested in finding out all the errors in the programs and classifying them as must ($\mho$), disjunctive may ($\top^1$) or may ($\top^2$) errors. For instance, from 48 faults of program tcas, S2 was able to detect all the errors in the program, and classified 31 of them as must ($\mho$) errors, 14 as disjunctive may ($\top^1$) errors and 3 as may ($\top^2$) errors. In summary, S2 detected 97% of real bugs including 73%, 11% and 13% of $\mho$, $\top^1$ and $\top^2$ errors, respectively.

However, a few errors were not detected by our system, e.g. $\texttt{v}_4$, $\texttt{v}_9$ of schedule2 and $\texttt{v}_1$, $\texttt{v}_2$ of print_tokens2 were verified successfully by S2. Upon careful examination, we found that the substituted statement in $\texttt{v}_9$ is semantically equivalent with the non-faulty one in $\texttt{v}_0$. Hence, we consider it as a bug in constructing the benchmark rather than a real program bug. For $\texttt{v}_1$, $\texttt{v}_2$ and $\texttt{v}_4$, there were omitted statements that are related to the I/O systems. For instance, the following statement is omitted in $\texttt{v}_1$:

```
if(ch == EOF) fprintf(stdout, “It can not get character”);
```

This was not picked up by our system since the specification of I/O operations were not being modelled. It would be interesting to see I/O operations being modelled in future.

Our calculus further supports debugging in localizing the errors. The LOE column shows the average number of lines of program code and specification relevant to the errors for each program. We are able to provide concise (between 3-5 lines) error locations for all the bugs in the suite. Such short but accurate localizations make it easier for users to comprehend the discovered errors. The last column shows the average time which S2 took for verifying a faulty version of each program.

Purely from the system point of view and on the assumption that specifications have already been provided, S2 took on average 16 seconds for safety proving, bug finding and error localization on one faulty version of each program.

3.8 Discussions

Recent work in first order relational logic [54, 80, 136] also addresses the problem of finding bugs in programs with pointers and linked data structures. The method is based on under approximation for loops and heap, thus it only finds the must bugs in the code. Furthermore since they consider only postcondition violation as a must error, they do not report on the more common bugs that are due to preconditions. The underlying assumption in [80] is that most bugs can be found in the program with small scope (loop unrolling) and small heap size. Our approach is more comprehensive since it treats error conditions as first class values that can be specified and we use predicates based on separation logic to represent the heap.

Static analysis based bug finding is not new and already exists for languages like Java [59]. As static analysis suffers from precision problem, there have been attempts to use dynamic or hybrid analysis for safety and bug finding. Check 'n' Crash [42] uses dynamic analysis techniques on the constraints produced by ESC/Java [59] to generate concrete test cases that can expose the true bugs. An approach based on dynamic analysis to infer likely invariants from code is implemented in [58]. Invariants discovered can be used as method annotations or assumptions, which can aid the static checker to detect bugs in the code [43]. This hybrid analysis uses a combination of under approximation and over approximation by doing different phases of analysis. In our approach we do not rely on dynamic analysis as our complete lattice can symbolically capture a richer set of possible program states. Our method integrates both bug-finding and safety proving within a single calculus, without prejudice to working with dynamic-based analyses for maybe error scenarios. Another approach based on dual analysis was presented in [46]. In this work, the may and must queries corresponds to safety and liveness properties. They require both over- and under-approximation and their conditions are computed with respect to a finite abstraction for each particular property. In comparison, the conditions for our must/may error are captured in terms of symbolic (infinite) domain that relies on *only* over-approximation mechanisms. Moreover, we have also shown how our error calculus can handle data structures accurately with must aliasing through a simple integration with separation logic. Lastly, the use of negation on postconditions to characterize errors have been explored in [83, 65]. Our proposal supports must bug checking for not only postconditions of method implementation but also preconditions of their method calls.

For explaining failure, while Bug Assist [83] uses maximum satisfiability query to localize the errors, authors in [65] use a bounded model checker for verifying *abnormal predicates* which are constructed from counterexamples and negated specifications. In the domain of program optimization, Daniel et. al. [137] describe how to explain failures in program analysis by reasons. The reasons computed by their system constitute necessary and sufficient conditions for the failure. In contrast, minimum satisfying assignments are used to compute abductions in [48] to support error diagnosis. They handle error report classification using proof obligations and failure witness derived from abductive inference. We use structural entailment rules to provide precise and concise error explanations.

Traditionally, the way to precisely derive the error conditions is via a backwards precondition calculation which was used by ESC/Java [26, 59, 72]. Snugglebug [26] has looked at ways to practically infer the potential error conditions in real programs by means of directed call graph construction to support weakest precondition calculation. However, they can only support may aliasing in their work which hampers the analysis of pointer-based programs. Our specification mechanism for must errors allows us to integrate the validation of bug with verification, within the separation logic framework, thus supporting a richer set of pointer-based programs. Similarly, to generate error conditions, Exorcise [72] is based on *weakest liberal precondition*. However, Exorcise only verifies must error conditions. Our calculus is more expressive (with verifying not only must error but safety and maybe error) with the help of new specification mechanism with explicit error scenarios.

Manually writing functional specifications for methods is costly, potentially inaccurate and time consuming. Thus, having a toolset to automatically develop and maintain program specifications is crucial to support a rigorous development paradigm for reliable software. In the context of heap-manipulating programs, specification inference relies on shape synthesis. In the next two chapters, we shall introduce a foundation of shape specifications inference via second-order bi-abduction.

Chapter 4

Towards Specification Inference

We present second-order bi-abduction mechanism in order to integrate inference capability into a modular-based verification system. In the context of pointer-based programs, we specialize our mechanism to inferring shape specifications. This forms a novel approach to shape analysis via the second-order bi-abductive mechanism in separation logic. [1]

In this chapter, we will present an overall discussion on the specification inference approach, but focus on shape domain. After that, we present a shape analysis mechanism via second-order bi-abduction. More concretely, we propose an entailment procedure of separation logic that is capable of abduction and frame inference. Finally, we provide an extension of the second-order bi-abduction to infer the complete specification. The output of the shape analysis will be further refined with a transformation in the next chapter.

4.1 From Shape Analysis to Shape Synthesis

An important challenge for automatic program verifiers lies in inferring shapes describing abstractions for data structures used by each method. In the context

[1]In separation logic, Calcagno et al. [20] introduced bi-abduction as a combination of abduction with frame inference.

of heap manipulating programs, determining the shape abstraction is crucial for proving memory safety and is a precursor to supporting functional correctness.

However, discovering shape abstractions can be rather challenging, as linked data structures span a wide variety of forms, from singly-linked lists, doubly-linked lists, circular lists, to tree-like data structures. Previous shape analysis proposals have made great progress in solving this problem. However, the prevailing approach relies on using a predefined vocabulary of shape definitions (typically limited to singly-linked list segments) and trying to determine if any of the pre-defined shapes fit the data structures used. This works well with programs that use simpler shapes, but would fail for programs which use more intricate data structures. An example is the method below (written in C and adapted from [76]) to build a tree whose leaf nodes are linked as a list.

```
struct tree { struct tree* parent; struct tree* l;
  struct tree* r; struct tree* next;
}
struct tree* tll(struct tree* x, struct tree* p, struct tree* t) {
  x->parent = p;
  if (x->r==NULL) {
    x->next=t;
    return x;
  }else{
    struct tree* lm = tll(x->r, x, t);
    return tll(x->l, x, lm);   }
}
```

Our approach to modular shape analysis would introduce an unknown pre-predicate H (as the pre-condition), and an unknown post-predicate G (as the post-condition), as shown below, where `res` is the method's result.

$$\texttt{requires}\quad \mathtt{H}(x,p,t) \qquad\quad \texttt{ensures}\quad \mathtt{G}(x,p,\texttt{res},t)$$

Using Hoare-style verification and a new second-order bi-abduction entailment procedure, we would derive a set of relational assumptions for the two unknown predicates. These derived assumptions are to ensure memory safety, and can be systematically transformed into concise predicate definitions for the unknown predicates, such as:

$$\begin{aligned}
\mathtt{H}(x,p,t) \equiv\ & x{\mapsto}\texttt{tree}(\mathcal{D}_p,\mathcal{D}_l,r,\mathcal{D}_n) \wedge \mathtt{r}{=}\mathtt{NULL}\\
& \vee\ x{\mapsto}\texttt{tree}(\mathcal{D}_p,l,r,\mathcal{D}_n){*}\mathtt{H}(l,x,lm){*}\mathtt{H}(r,x,t) \wedge \mathtt{r} \neq \mathtt{NULL}\\
\mathtt{G}(x,p,\texttt{res},t) \equiv\ & x{\mapsto}\texttt{tree}(p,\mathcal{D}_l,r,t) \wedge \texttt{res}{=}\mathtt{x}{\wedge}\mathtt{r}{=}\mathtt{NULL}\\
& \vee\ x{\mapsto}\texttt{tree}(p,l,r,\mathcal{D}_n){*}\mathtt{G}(l,x,\texttt{res},lm){*}\mathtt{G}(r,x,lm,t){\wedge}\mathtt{r}{\neq}\mathtt{NULL}
\end{aligned}$$

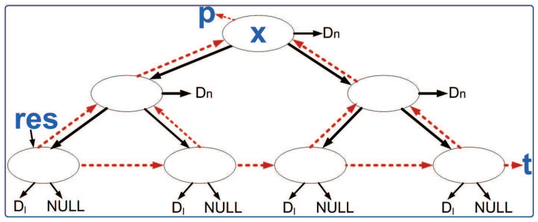

Figure 4-1: An example of G(x,p,res,t)

The derived pre-predicate H captures a binary tree-like shape that would be traversed by the method. $\mathtt{x}{\mapsto}\texttt{tree}(\mathcal{D}_\mathtt{p},\mathcal{D}_\mathtt{l},\mathtt{r},\mathcal{D}_\mathtt{n})$ denotes that `x` refers to a `tree` node with its `parent`, `l`, `r` and `next` fields being $\mathcal{D}_\mathtt{p}$, $\mathcal{D}_\mathtt{l}$, `r` and $\mathcal{D}_\mathtt{n}$, respectively. We use dangling references, such as $\mathcal{D}_\mathtt{l},\mathcal{D}_\mathtt{p},\mathcal{D}_\mathtt{n}$, as generic markers that denote field pointers that are not traversed by the method. Thus no assertion can be made on any of the $\mathcal{D}$ pointers. The post-predicate G, illustrated in Fig 4-1, adds `parent` field links for all nodes, and `next` field links for just the leaves. [2]

Current shape analysis mechanisms [13, 20, 57] are unable to infer pre/post specifications that ensure memory-safety for such complex examples. In this paper, we propose a fresh approach to shape analysis that can synthesize, from scratch, a set of shape abstractions that ensure memory safety. The central concept behind our proposal is the use of *unknown predicates* (or *second-order variables*) as place holders for shape predicates that are to be synthesized directly from proof obligations gathered by our verification process. Our proposal is based on a novel *bi-abductive entailment* that supports *second-order* variables. The core of the new entailment procedure generates a set of relational assumptions on unknown predicates to ensure memory safety. These assumptions are then refined into predicate definitions, by predicate *derivation* and *normalization* steps.

By building the generation of the required *relational assumptions* over unknown predicates directly into the new entailment checker, we were able to integrate our shape analysis into an existing program verifier with changes made only to the entailment process, rather than the program verification/analysis itself. Our proposed shape analysis thus applies an almost standard set of Hoare rules in constructing proof obligations which are discharged through the use of a new *second-order* bi-abductive entailment.

To sum up, we present a new modular shape analysis that can synthesize heap memory specification on a per method basis. We rely on a *second-order bi-abduction* mechanism that can give interpretations to unknown shape predicates by inferring *relational assumptions* on unknown predicates, before a predicate *derivation* and *normalization* process. There are several novel features in our shape analysis. Firstly, it is grounded on second-order abduction rather than

[2]Note that new links formed by the method are dashed and colored in red.

deduction. Secondly we provide a *heap guard* mechanism to support more precise preconditions for heap specification. Lastly, we provide a *instantiation* scheme to guide the entailment procedure. Our approach has been proven sound and been implemented on top of an existing automated verification system. We show its versatility in synthesizing a wide range of intricate shape specifications.

Organization. We shall present the shape fragment of separation logic used for the analysis in section 4.2. We introduce the second-order bi-abductive mechanism in section 4.3. After that we describe the instantiation scheme and formalism of the second-order bi-abduction in section 4.4. Section 4.5 presents Hoare rules and section 5.5 highlights the soundness. The two last sections show our implementation and a comparative remark.

4.2 Logic Syntax for Shape Specification

We revise the fragment of specification logic in section 2.2. We outline below the fragment underlying the proposed analysis:

Disj. formula	Φ	$::= \Delta \mid \Phi_1 \vee \Phi_2$
Guarded Disj.	Φ^g	$::= \Delta \mid (\Delta \,\texttt{@}\, (\kappa \wedge \pi)) \mid {\Phi^g}_1 \vee {\Phi^g}_2$
Conj. formula	Δ	$::= \exists \bar{v} \cdot (\kappa \wedge \pi)$
Spatial formula	κ	$::= \texttt{emp} \mid \top \mid v {\mapsto} c(\bar{v}) \mid \texttt{P}(\bar{v}) \mid \texttt{U}(\bar{v}) \mid \kappa_1 {*} \kappa_2$
Pure formula	π	$::= \alpha \mid \neg\alpha \mid \pi_1 \wedge \pi_2$
Var (Dis)Equality	α	$::= v \mid v_1{=}v_2 \mid v{=}\texttt{NULL} \mid v_1{\neq}v_2 \mid v{\neq}\texttt{NULL}$
Pred. Defn.	$\texttt{P}^{\texttt{def}}$	$::= \texttt{P}(\bar{v}) \equiv \Phi^g$
Pred. Dict.	Γ	$::= \{\texttt{P}_1^{\texttt{def}}, \ldots, \texttt{P}_n^{\texttt{def}}\}$

$\texttt{P} \in$ Known Predicates $\quad \texttt{U} \in$ Unknown Predicates

$c \in$ Data Nodes $\quad v \in$ Variables $\quad \bar{v} \equiv v_1 \ldots v_n$

We introduce $\Delta \,\texttt{@}\,(\kappa{\wedge}\pi)$, a special syntactic form called *guarded heap* that capture a heap context $\kappa{\wedge}\pi$ in which Δ holds. Thus, $\Delta \,\texttt{@}\,(\kappa{\wedge}\pi)$ holds for heap configurations that satisfy Δ and that can be extended such that they satisfy $\Delta * \kappa{\wedge}\pi$. In Sec.5.2 we will describe its use in allowing our shape inference to incorporate path sensitive information in the synthesized predicates. The assertion language is also extended with the following formula for describing heaps: `emp` denoting the empty heap; $\top$ denoting an arbitrary heap (pointed by dangling reference); points-to assertion, $x{\mapsto}c(\bar{v})$, specifying the heap in which x points to a data structure of type c whose fields contain the values $\bar{v}$; known predicate, $\texttt{P}(\bar{v})$, which holds for heaps in which the shape of the memory locations reachable from $\bar{v}$ can be described by the `P` predicate; unknown predicates, $\texttt{U}(\bar{v})$, with no prior given definitions. Separation conjunction $\kappa_1{*}\kappa_2$ holds for heaps that can be partitioned in two disjoint components satisfying κ_1 and κ_2, respectively. The pure formula captures only pointer equality and disequality. We allow a special constant `NULL` to denote a pointer which does not point to any heap location. Known predicates $\texttt{P}(\bar{v})$ are defined inductively through disjunctive formulas Φ^g. Their definitions are either user-given or synthesised by our analysis. We will use Γ to denote the repository (or set) of available predicate definitions. Through our analysis, we shall construct an inductive definition for each unknown predicate, where possible. Unknown predicates that have *not* been instantiated would not have any definition. They denote data fields that are not accessed by their methods, and would be marked as *dangling pointers.*

4.3 Overview of Shape Inference

Our approach on specification inference comprises three main steps: (i) inferring relational assumptions for unknown predicates via Hoare-style verification, (ii) deriving predicates from relational assumptions, (iii) normalizing predicates. In this chapter, we discuss the first step, the second-order bi-abduction mechanism. The remain steps, step (ii) and step (iii), will be discussed in the next chapter. For (i), a key machinery is the entailment procedure that must work with second-order variables (unknown predicates). Previous bi-abduction entailment proposals, pioneered by [20], would take an antecedent $\Delta_{\texttt{ante}}$ and a consequent $\Delta_{\texttt{conseq}}$ and return a frame residue $\Delta_{\texttt{frame}}$ and the precondition $\Delta_{\texttt{pre}}$, such that the following holds: $\Delta_{\texttt{pre}}{*}\Delta_{\texttt{ante}} \models \Delta_{\texttt{conseq}}{*}\Delta_{\texttt{frame}}$. Here, all four components use separation logic formulas based on known predicates with prior definitions.

Taking a different tact, we start with an existing entailment procedure for separation logic with user-defined predicates, and extend it to accept formulas with second-order variables such that given an antecedent $\Delta_{\texttt{ante}}$ and a consequent $\Delta_{\texttt{conseq}}$ the resulting entailment procedure infers both the frame residue $\Delta_{\texttt{frame}}$ and a set (or conjunction) of relational assumptions (on unknowns) of the form $\mathcal{R} = \bigwedge_{i=1}^{n}(\Delta_i \Rightarrow {\Phi^g}_i)$ such that:

$$\mathcal{R} \wedge \Delta_{\texttt{ante}} \models \Delta_{\texttt{conseq}}{*}\Delta_{\texttt{frame}}$$

The inferred $\mathcal{R}$ ensures the entailment's validity. We shall use the following notation $\Delta_{\texttt{ante}} \vdash \Delta_{\texttt{conseq}} \leadsto (\mathcal{R}, \Delta_{\texttt{frame}})$ for this second-order bi-abduction process.

There are two scenarios to consider for unknown predicates: (1) $\Delta_{\texttt{ante}}$ contains an *unknown* predicate instance that matched with a points-to or known predicate in $\Delta_{\texttt{conseq}}$; (2) $\Delta_{\texttt{conseq}}$ contains an *unknown* predicate instance. An example of the first scenario is:

$$\texttt{U(x)} \vdash \texttt{x}{\mapsto}\texttt{snode(n)} \leadsto (\texttt{U(x)}{\Rightarrow}\texttt{x}{\mapsto}\texttt{snode(n)}{*}\texttt{U}_0\texttt{(n)},\ \texttt{U}_0\texttt{(n)})$$

Here, we generated a relational assumption to denote an *unfolding* (or instantiation) for the unknown predicate `U` to a heap node `snode` followed by another unknown $\texttt{U}_0(\texttt{n})$ predicate. The data structure `snode` is defined as `struct snode { struct snode* next}`. A simple example of the second scenario is shown next.

$$\texttt{x}{\mapsto}\texttt{snode(NULL)}{*}\texttt{y}{\mapsto}\texttt{snode(NULL)} \vdash \texttt{U}_1(\texttt{x}) \leadsto (\texttt{x}{\mapsto}\texttt{snode(NULL)}{\Rightarrow}\texttt{U}_1(\texttt{x}), \texttt{y}{\mapsto}\texttt{snode(NULL)})$$

The generated relational assumption depicts a *folding* process for unknown $\texttt{U}_1(\texttt{x})$ which captures a heap state traversed from the pointer `x`. Both folding and unfolding of unknown predicates are crucial for second-order bi-abduction. To make it work properly for unknown predicates with multiple parameters, we shall later provide a novel #-annotation scheme to guide these processes. For the moment, we shall use this annotation scheme implicitly. Consider the following method which traverses a singly-linked list and converts it to a doubly-linked list (let us ignore the states $\alpha_1, .., \alpha_5$ for now):

```
struct node { struct node* prev; struct node* next}
void sll2dll(struct node* x, struct node* q)
{(α1) if (x==NULL)
    (α2) return;
  (α3) x->prev = q;
  (α4) sll2dll(x->next, x); (α5)
}
```

To synthesize the shape specification for this method, we introduce two unknown predicates, `H` for the pre-condition and `G` for the post-condition, as below.

$$\texttt{requires}\quad \texttt{H}(x,q) \qquad \texttt{ensures}\quad \texttt{G}(x,q)$$

We then apply code verification using these pre/post specifications with unknown predicates and attempt to collect a set of relational assumptions (over

$(\mathtt{A1}).\mathtt{H}(x,q) \wedge \mathtt{x{=}NULL} \Rightarrow \mathtt{G}(x,q)$

$(\mathtt{A2}).\mathtt{H}(x,q) \wedge \mathtt{x{\neq}NULL} \Rightarrow$
$\quad x{\mapsto}node(x_p,x_n){*}\mathtt{H_p}(x_p,q){*}\mathtt{H_n}(x_n,q)$

$(\mathtt{A3}).\mathtt{H_n}(x_n,q) \Rightarrow \mathtt{H}(x_n,x) \,\texttt{@}\, x{\mapsto}node(q,x_n)$

$(\mathtt{A4}).x{\mapsto}node(q,x_n){*}\mathtt{G}(x_n,x) \Rightarrow \mathtt{G}(x,q)$

(a)

(α_1). $\mathtt{H}(x,q)$

(α_2). $\mathtt{H}(x,q){\wedge}\mathtt{x{=}NULL}$

(α_3). $x{\mapsto}node(x_p,x_n){*}\mathtt{H_p}(x_p,q){*}\mathtt{H_n}(x_n,q){\wedge}\mathtt{x{\neq}NULL}$

(α_4). $x{\mapsto}node(q,x_n){*}\mathtt{H_p}(x_p,q){*}\mathtt{H_n}(x_n,q){\wedge}\mathtt{x{\neq}NULL}$

(α_5). $x{\mapsto}node(q,x_n){*}\mathtt{H_p}(x_p,q){*}\mathtt{G}(x_n,x){\wedge}\mathtt{x{\neq}NULL}$

(b)

Figure 4-2: Relational assumptions (a) and program states (b) for `sll2dll`

the unknown predicates) that must hold to ensure memory-safety. These assumptions would also ensure that the pre-condition of each method call is satisfied, and that the coresponding post-condition is ensured at the end of the method body. For example, our analysis can infer four relational assumptions for the `sll2dll` method as shown in Fig. 4-2(a).

These relational assumptions include two new unknown predicates, $\mathtt{H_p}$ and $\mathtt{H_n}$, created during the code verification process. All relational assumptions are of the form $\Delta_{\mathtt{lhs}}{\Rightarrow}\Delta_{\mathtt{rhs}}$, except for (A3) which has the form $\Delta_{\mathtt{lhs}}{\Rightarrow}\Delta_{\mathtt{rhs}} \,\texttt{@}\, \Delta_{\mathtt{g}}$ where $\Delta_{\mathtt{g}}$ denotes a heap guard condition. Such heap guard condition allows more precise pre-conditions to be synthesized (e.g. $\mathtt{H_n}$ in (A3)), and is shorthand for $\Delta_{\mathtt{lhs}}{*}\Delta_{\mathtt{g}}{\Rightarrow}\Delta_{\mathtt{rhs}}{*}\Delta_{\mathtt{g}}$.

Let us look at how relational assumptions are inferred. At the start of the method, we have (α_1), shown in Fig. 4-2 (b), as our program state. Upon exit from the then branch, the verification requires that the postcondition $\mathtt{G(x,q)}$ be established by the program state (α_2), generating the relational assumption (A1) via the following entailment:

$$(\alpha_2) \vdash \mathtt{G}(x,q) \leadsto (\mathtt{A1}, \mathtt{emp} \wedge \mathtt{x{=}NULL}) \tag{E1}$$

To get ready for the field access `x->prev`, the following entailment is invoked to unfold the unknown `H` predicate to a heap node, generating the relational assumption (`A2`):

$$\mathtt{H}(x,q) \wedge \mathtt{x{\neq}NULL} \vdash x{\mapsto}node(x_p,x_n) \leadsto (\mathtt{A2}, \mathtt{H_p}(x_p,q){*}\mathtt{H_n}(x_n,q) \wedge \mathtt{x{\neq}NULL}) \tag{E2}$$

Two new unknown predicates $\mathtt{H_p}$ and $\mathtt{H_n}$ are added to capture the `prev` ($\mathtt{x_p}$) and `next` ($\mathtt{x_n}$) fields of `x` (i.e. they represent heaps referred to by $\mathtt{x_p}$ and $\mathtt{x_n}$ respectively). After binding, the verification now reaches the state (α_3), which is then changed to (α_4) by the field update `x->prev = q`. Relational assumption (`A3`) is inferred from proving the precondition $\mathtt{H}(\mathtt{x_n},\mathtt{x})$ of the recursive call `sll2dll(x->next, x)` at the program state (α_4):

$$(\alpha_4) \vdash \mathtt{H}(x_n,x) \leadsto (\mathtt{A3}, x{\mapsto}node(q,x_n){*}\mathtt{H_p}(x_p,q){\wedge}\mathtt{x{\neq}NULL}) \tag{E3}$$

Note that the heap guard $x{\mapsto}node(q,x_n)$ from (α_4) is recorded in (`A3`), and is crucial for predicate derivation. The program state at the end of the recursive call, (α_5), is required to establish the post-condition `G(x, q)`, generating the relational assumption (`A4`):

$$(\alpha_5) \vdash \mathtt{G}(x,q) \leadsto (\mathtt{A4}, \mathtt{H_p}(x_p,q){\wedge}\mathtt{x{\neq}NULL}) \tag{E4}$$

These relational assumptions are automatically inferred symbolically during code verification.

$$\begin{array}{l}
(1).\mathtt{H}(x,p,t) \Rightarrow x{\mapsto}\mathtt{tree}(x_p,l,r,n)*\mathtt{H_p}(x_p,p,t)*\mathtt{H_l}(l,p,t)*\mathtt{H_r}(r,p,t)*\mathtt{H_n}(n,p,t)\\
(2).\mathtt{H_r}(r,p,t) \wedge \mathtt{r}{\neq}\mathtt{NULL}\ \mathtt{@}\ x{\mapsto}\mathtt{tree}(p,l,r,n) \Rightarrow \mathtt{H}(r,x,t)\\
(3).\mathtt{H_l}(l,p,t) \Rightarrow \mathtt{H}(l,x,lm)\ \mathtt{@}\ (x{\mapsto}\mathtt{tree}(p,l,r,n) \wedge \mathtt{r}{\neq}\mathtt{NULL})\\
(4).\mathtt{H_l}(l,p,t)*\mathtt{H_r}(r,p,t)*x{\mapsto}\mathtt{tree}(p,l,r,t){\wedge}\mathtt{r}{=}\mathtt{NULL}{\wedge}\mathtt{res}{=}\mathtt{x} \Rightarrow \mathtt{G}(x,p,\mathtt{res},t)\\
(5).\mathtt{H_n}(n,p,t) * x{\mapsto}\mathtt{tree}(p,l,r,n) * \mathtt{G}(r,x,lm,t) * \mathtt{G}(l,x,\mathtt{res},lm){\wedge}\\
\qquad\qquad \mathtt{r}{\neq}\mathtt{NULL} \Rightarrow \mathtt{G}(x,p,\mathtt{res},t)
\end{array}$$

Figure 4-3: Relational assumptions for `tll`

The `tll` example. Let us revisit the `tll` example shown in section 4.1. To synthesize the shape specification for this method, we introduce two unknown predicates, H for the pre-condition and G for the post-condition, as mentioned earlier.

$$\mathtt{requires}\ \ \mathtt{H}(x,p,t) \qquad \mathtt{ensures}\ \ \mathtt{G}(x,p,\mathtt{res},t)$$

We then apply code verification using these pre/post specifications with unknown predicates and attempt to collect a set of relational assumptions (over the unknown predicates) that must hold to ensure memory-safety. These assumptions would also ensure that the pre-condition of each method call is satisfied, and that the post-condition is ensured at the end of its method body.

For example, our analysis can infer the following five relational assumptions for the `tll` method as in Figure 4-3.

Our relational assumptions include four new unknown predicates, $\mathtt{H_p}, \mathtt{H_l}, \mathtt{H_r}$ and $\mathtt{H_n}$, that were created during the code verification process.

Let us look at how relational assumptions are inferred. For illustration, we annotate program states into the `tll` example as follows:

```
struct tree* tll(struct tree* x, struct tree* p, struct tree* t) {
 (S1)x->parent = p;
 (S2)if (x->r==NULL) {
    (S3) x->next=t; (S6)
        return x; }
      else{
   (S4)  struct tree* lm = tll(x->r, x, t);
    (S5) return tll(x->l, x, lm); (S7) }
 }
```

(S1). $\mathtt{H}(x,p,t)$

(S2). $x{\mapsto}\mathtt{tree}(p,l,r,n){*}\mathtt{H_l}(l,p,t){*}\mathtt{H_r}(r,p,t){*}\mathtt{H_n}(n,p,t){*}\mathtt{H_p}(x_P,p,t)$

(S3). $x{\mapsto}\mathtt{tree}(p,l,r,n){*}\mathtt{H_l}(l,p,t){*}\mathtt{H_r}(r,p,t){*}\mathtt{H_n}(n,p,t){\wedge}r{=}\mathtt{NULL}$

(S4). $x{\mapsto}\mathtt{tree}(p,l,r,n){*}\mathtt{H_l}(l,p,t){*}\mathtt{H_r}(r,p,t){*}\mathtt{H_n}(n,p,t){\wedge}r{\neq}\mathtt{NULL}$

(S5). $x{\mapsto}\mathtt{tree}(p,l,r,n){*}\mathtt{H_l}(l,p,t){*}\mathtt{G}(r,x,lm,t){*}\mathtt{H_n}(n,p,t){\wedge}r{\neq}\mathtt{NULL}$

(S6). $x{\mapsto}\mathtt{tree}(p,l,r,t){*}\mathtt{H_l}(l,p,t){*}\mathtt{H_r}(r,p,t){\wedge}\mathtt{r}{=}\mathtt{NULL}{\wedge}\mathtt{res}{=}\mathtt{x}$

(S7). $x{\mapsto}\mathtt{tree}(p,l,r,n){*}\mathtt{G}(l,x,\mathtt{res},lm){*}\mathtt{G}(r,x,lm,t){*}\mathtt{H_n}(n,p,t){\wedge}\mathtt{r}{\neq}\mathtt{NULL}$

At the start of the method, we have (S1), shown above, as our program state. Due to a field update, `x->parent`, relational assumption (1) was inferred which lead to (S2). The conditional evaluation led to (S3) and (S4), as the program states at the start of the then-branch and else-branch, respectively. Relational assumption (2) was then inferred from proving pre-condition `H(r, x, t)` of recursive call `tll(x->r, x, t)` under (S4), yielding program state (S5). Also, (3) was inferred from proving pre-condition `H(l, x, lm)` of the second recursive call `tll(x->l, x, lm)`. When inferring (3), heap guard $x{\mapsto}tree(p,l,r,n){\wedge}\mathtt{r}{\neq}\mathtt{NULL}$ from the program state (S5) was used, since $\mathtt{H}_1(l,p,t) \Rightarrow \mathtt{H}(l,x,lm)$, by itself, neither capture a connected context $\mathtt{r} \neq \mathtt{NULL}$ from the then-branch, nor properly instantiate the back (parent) pointer `x`. The program state (S6) at the end of then-branch was then used to prove post-condition $\mathtt{G}(x,p,\mathtt{res},t)$. This proving lead to relational assumption (4). Similarly, program state (S7) at the end of the else-branch, which assumed $\mathtt{G}(r,x,lm,t) * \mathtt{G}(l,x,\mathtt{res},lm)$ from the two recursive calls, would infer (5) when proving the post-condition of the method itself.

These relational assumptions are inferred symbolically during code verification with the help of second-order bi-abduction mechanism that we are proposing. They are also being *modularly* inferred on a per method basis, using automatically generated template pre/post conditions with unknown predicates.

Compared to some prior shape analyses, such as [57, 68], which requires the entire program to be available for analysis, our approach can perform this task modularly on a per method basis instead.

Our approach currently works only for shape abstractions of tree-like data structures with forward and back pointers. We are unable to infer specifications for graph-like or overlaid data structures yet. These abstractions are being inferred *modularly* on a per method basis. The inferred preconditions are typically the weakest ones that would ensure memory safety, and would be applicable to all contexts of use. We shall next elaborate and formalise on our second-order bi-abduction process.

4.4 Second-Order Bi-Abduction

We have seen the need for a bi-abductive entailment procedure to systematically handle unknown predicates. To cater to predicates with multiple parameters, we shall use an automatic *#-annotation* scheme to support both unfolding and folding of unknown predicates. Consider a predicate $U(v_1,..,v_n,w_1\#,..,w_m\#)$, where parameters $\mathtt{v_1,..,v_n}$ are unannotated and parameters $\mathtt{w_1,..,w_m}$ are #-annotated. From the perspective of unfolding, we permit each variable from $\mathtt{v_1,..,v_n}$ to be instantiated at most once (we call them *instantiatable*), while variables $\mathtt{w_1,..,w_m}$ are *disallowed* from instantiation (we call them *non-instantiatable*). This scheme ensures that each pointer is instantiated at most once, and avoids formulas, like $U_3(y,y)$ or $U_2(r,y){*}U_3(y,x\#)$, from being formed. Such formulas, where a variable may be repeatedly instantiated, may cause a trivial `false` pre-condition to be inferred. Though sound, it is imprecise. From the perspective of folding, we allow heap traversals to start from variables $\mathtt{v_1,..,v_n}$ and would stop whenever references to $\mathtt{w_1,..,w_m}$ are encountered. This allows us to properly infer segmented shape predicates and back pointers. Our annotation scheme is fully automated, as we would infer the #-annotation of pre-predicates based on which parameters could be field accessed; while parameters of post-predicates are left unannotated. For our running example, since `q` parameter is not field accessed (in its method's body), our automatic annotation scheme would start with the following pre/post specification:

$$\texttt{requires}\;\; \mathtt{H}(x,q\#) \qquad \texttt{ensures}\;\; \mathtt{G}(x,q)$$

Unfold. The entailment below results in an unfolding of the unknown `H` predicate. It is essentially (`E2`) in Sec 4.3, except that `q` is marked explicitly as non-instantiatable.

$$H(x,q\#){\wedge}\mathtt{x{\neq}NULL} \vdash x{\mapsto}node(x_p,x_n) \leadsto (\mathtt{A2},\Delta_1) \qquad (\mathtt{E2}')$$

With non-instantiatable variables explicitly annotated, the assumption (`A2`) becomes:

$$\mathtt{A2} \;\equiv\; H(x,q\#){\wedge}\mathtt{x{\neq}NULL} \Rightarrow x{\mapsto}node(x_p,x_n){*}H_p(x_p,q\#){*}H_n(x_n,q\#)$$

As mentioned earlier, we generated a new unknown predicate for each pointer field ($\mathtt{H_p}$ for $\mathtt{x_p}$, and $\mathtt{H_n}$ for $\mathtt{x_n}$), so as to allow the full recovery of the shape of the data structure being traversed or built. Note that each $\mathtt{x}, \mathtt{x_p}, \mathtt{x_n}$ appears only once in unannotated forms, while the annotated $\mathtt{q\#}$ remains annotated throughout to prevent the pointer from being instantiated. If we allow $\mathtt{q}$ to be instantiatable in ($\mathtt{E2'}$) above, we will instead obtain:

$$H(x,q){\wedge}\mathtt{x}{\neq}\mathtt{NULL} \vdash x{\mapsto}node(x_p,x_n) \rightsquigarrow (\mathtt{A2'}, \Delta_1')$$

We get

$$\mathtt{A2'} \;\equiv\; H(x,q){\wedge}\mathtt{x}{\neq}\mathtt{NULL} \Rightarrow x{\mapsto}node(x_p,x_n){*}H_p(x_p,q\#){*}H_n(x_n,q\#){*}\underline{U_2(q,x\#)},$$

where the unfolding process creates extra unknown predicate $\mathtt{U_2(q,x\#)}$ to capture shape for $\mathtt{q}$.

Our proposal for instantiating unknown predicates is also applicable when known predicates appear in the RHS. These known predicates may have parameters that act as *continuation fields* for the data structure. An example is the list segment $lseg(x,p)$ predicate where the parameter $\mathtt{p}$ is a continuation field.

$$ll(x) \;\equiv\; \mathtt{emp}{\wedge}\mathtt{x}{=}\mathtt{NULL} \vee x{\mapsto}snode(n) * ll(n)$$

$$lseg(x,p) \;\equiv\; \mathtt{emp}{\wedge}\mathtt{x}{=}\mathtt{p} \vee x{\mapsto}snode(n) * lseg(n,p)$$

Where `snode` (defined in the previous section) denotes singly-linked list node. Note that continuation fields play the same role as fields for data nodes. Therefore, for such parameters, we also generate new unknown parameters to capture the connected data structure that may have been traversed. We illustrate this with two examples:

$$U(x) \vdash ll(x) \rightsquigarrow (U(x){\Rightarrow}ll(x), \mathtt{emp})$$

$$U(x) \vdash lseg(x,p) \rightsquigarrow (U(x){\Rightarrow}lseg(x,q){*}U_2(q), U_2(p))$$

The first predicate $ll(x)$ did not have a continuation field. Hence, we did not generate any extra unknown predicate. The second predicate $lseg(x,p)$ did have a continuation field $\mathtt{p}$, and we generated an extra unknown predicate $U_2(p)$ to capture a possible extension of the data structure beyond this continuation field.

Fold. A second scenario that must be handled by second-order entailment involves unknown predicates in the consequent. For each unknown predicate $\mathtt{U_1}(\bar{\mathtt{v}},\bar{\mathtt{w}}\#)$ in the consequent, a corresponding assumption $\Delta{\Rightarrow}U_1(\bar{v},\bar{w}\#)\,\mathtt{@}\,\Delta_g$ is inferred where Δ contains unknown predicates with at least one instantiatable parameters from $\bar{\mathtt{v}}$, or heaps *reachable* from $\bar{\mathtt{v}}$ (via either any data fields or parameters of known predicates) but stopping at non-instantiatable variables $\bar{\mathtt{w}}\#$; a residual frame is also inferred from the antecedent (but added with pure approximation of footprint heaps [33]). For example, consider the following entailment:

$$x{\mapsto}snode(q){*}q{\mapsto}snode(\mathtt{NULL}){\wedge}\mathtt{q}{\neq}\mathtt{NULL} \vdash U_1(x,q\#) \leadsto (\mathtt{A_{f1}},\Delta_1)$$

The output of this entailment is:

$$\mathtt{A_{f1}} \equiv x{\mapsto}snode(q){\wedge}\mathtt{q}{\neq}\mathtt{NULL}{\Rightarrow}U_1(x,q\#) \quad \Delta_1 \equiv \mathtt{q}{\mapsto}\mathtt{snode}(\mathtt{NULL}){\wedge}\mathtt{x}{\neq}\mathtt{NULL}{\wedge}\mathtt{x}{\neq}\mathtt{q}$$

As a comparison, let us consider the scenario where $\mathtt{q}$ is unannotated, as follows:

$$x{\mapsto}snode(q){*}q{\mapsto}snode(\mathtt{NULL}){\wedge}\mathtt{q}{\neq}\mathtt{NULL} \vdash U_1(x,q) \leadsto (\mathtt{A_{f2}},\Delta_2)$$

In this case, the output of the entailment becomes:

$$\mathtt{A_{f2}} \equiv x{\mapsto}snode(q){*}q{\mapsto}snode(\mathtt{NULL}) \Rightarrow U_1(x,q) \quad \Delta_2 \equiv \mathtt{x}{\neq}\mathtt{NULL}{\wedge}\mathtt{q}{\neq}\mathtt{NULL}{\wedge}\mathtt{x}{\neq}\mathtt{q}$$

Moreover, the folding process also captures *known* heaps that are reachable from #-parameters as *heap guard conditions*, e.g. $\mathtt{x}{\mapsto}\mathtt{node}(\mathtt{q},\mathtt{x_n})$ in our running example

(E3):

$$x{\mapsto}node(q,x_n){*}H_p(x_p,q\#){*}H_n(x_n,q\#){\wedge}\texttt{x}{\neq}\texttt{NULL} \vdash H(x_n, x\#)$$
$$\rightsquigarrow (H_n(x_n, q\#) \Rightarrow H(x_n, x\#)\ \texttt{@}\ x{\mapsto}node(q, x_n), x{\mapsto}node(q,x_n){*}H_p(x_p,q\#){\wedge}\texttt{x}{\neq}\texttt{NULL})\ (\texttt{E3}')$$

Such heap guards help with capturing the relations of heap structures and recovering those relationships when necessary (e.g. back-pointer $\texttt{x\#}$).

Formalism. Bi-abductive unfold is formalized in Fig. 4-4. Here, $\texttt{project}(\bar{w}, \pi)$ is an auxiliary function that existentially quantifies in π all free variables that are not in the set $\bar{w}$. Thus it eliminates from π all subformulas not related to

$$\frac{\begin{array}{c}[\underline{\textbf{SO-ENTAIL-UNFOLD}}]\\ \kappa_s \equiv r{\mapsto}c(\bar{p}) \text{ or } \kappa_s \equiv \texttt{P}(r, \bar{p})\\ \kappa_f = *_{p_j \in \bar{p}}\ \texttt{U}_j(p_j, \bar{v}_i\#, \bar{v}_n\#) \quad \kappa_r =\ \texttt{U}_{rem}(\bar{v}_i, \bar{v}_n\#, r\#) \quad \text{where } \texttt{U}_r, \texttt{U}_j\text{: fresh preds}\\ \pi_a = \texttt{project}(\{r, \bar{v}_i, \bar{v}_n, \bar{p}\}, \pi_1) \quad \pi_c = \texttt{project}(\{\bar{p}\}, \pi_2)\\ \sigma \equiv (\texttt{U}(r, \bar{v}_i, \bar{v}_n\#) \wedge \pi_a \Rightarrow \kappa_s * \kappa_f * \kappa_r \wedge \pi_c)\\ \kappa_1 * \kappa_f * \kappa_{rem} \wedge \pi_1 \vdash \kappa_2 \wedge \pi_2 \rightsquigarrow (\mathcal{R}, \Delta_R)\end{array}}{\texttt{U}(r, \bar{v}_i, \bar{v}_n\#){*}\kappa_1{\wedge}\pi_1 \vdash \kappa_s{*}\kappa_2{\wedge}\pi_2 \rightsquigarrow (\sigma{\wedge}\mathcal{R}, \Delta_R)}$$

Figure 4-4: Bi-Abductive Unfolding.

$\bar{w}$ (*e.g.* $\texttt{project}(\{x, q\}, q{=}\texttt{NULL}{\wedge}y{>}3)$ returns $q{=}\texttt{NULL}$). In the first line, a RHS assertion, either a points-to assertion $\texttt{r}{\mapsto}\texttt{c}(\bar{\texttt{p}})$ or a known predicate instance $\texttt{P}(\texttt{r}, \bar{\texttt{p}})$ is paired through the parameter $\texttt{r}$ with the unknown predicate $\texttt{U}$. Second, the unknown predicates $\texttt{U}_\texttt{j}$ are generated for the data fields/parameters of $\kappa_\texttt{s}$. Third, the unknown predicate $\texttt{U}_\texttt{rem}$ is generated for the instantiatable parameters $\bar{\texttt{v}}_\texttt{i}$ of $\texttt{U}$. The fourth and fifth lines compute relevant pure formulas and generate the assumption, respectively. Finally, the unknown predicates $\kappa_\texttt{f}$ and $\kappa_\texttt{r}$ are combined in the residue of LHS to continue discharging the remaining formula in RHS.

Bi-abductive fold is formalized in Fig. 4-5. The function $\texttt{reach}(\bar{w}, \kappa_1{\wedge}\pi_1, \bar{z}\#)$ extracts portions from the antecedent heap (κ_1) that are (1) unknown predicates containing at least one instantiatable parameter from $\bar{w}$; or (2) point-to or known predicates reachable from $\bar{w}$, but not reachable from $\bar{z}$. In our running example (the entailment ($\mathtt{E3}'$) on last page), the function $\mathtt{reach}(\{x_n\}, x{\mapsto}\mathtt{node}(q, x_n) {*} \mathtt{H_p}(x_p, q\#) {*} \mathtt{H_n}(x_n, q\#) {\wedge} x{\neq}\mathtt{NULL}, \{x\#\})$ is used to obtain $\mathtt{H_n}(x_n, q\#)$. More detail on this function is in the report [89]. The $\mathit{heaps}(\Delta)$ function enumerates all known predicate instances (of the form $\mathtt{P}(\bar{v})$) and points-to instances (of the form $r{\mapsto}c(\bar{v})$)) in Δ. The function $\mathit{root}(\kappa)$ is defined as: $\mathit{root}(r{\mapsto}c(\bar{v}))=\{r\}$, $\mathit{root}(\mathtt{P}(r, \bar{v})) = \{r\}$. In the first line, heaps of LHS are separated into the assumption κ_{11} and the residue κ_{12}. Second, heap guards

$$\left[\underline{\textbf{SO-ENTAIL-FOLD}}\right]$$

$$\frac{\begin{array}{c}\kappa_{11}{=}\mathtt{reach}(\bar{w}, \kappa_1{\wedge}\pi_1, \bar{z}\#) \quad \exists\kappa_{12} \cdot \kappa_1{=}\kappa_{11}{*}\kappa_{12} \\ \kappa_g = {*}\{\kappa \mid \kappa{\in}\mathtt{heaps}(\kappa_{12}){\wedge}\mathit{root}(\kappa){\subseteq}\bar{z}\} \quad \bar{r}{=}\bigcup_{\kappa\in\kappa_g}\mathit{root}(\kappa) \\ \sigma \equiv (\kappa_{11}{\wedge}\mathtt{project}(\bar{w}, \pi_1) \Rightarrow \mathtt{U}_c(\bar{w}, \bar{z}\#)\,\mathtt{@}\,\kappa_g{\wedge}\mathtt{project}(\bar{r}, \pi_1)) \\ \kappa_{12} \wedge \pi_1 \vdash \kappa_2 \wedge \pi_2 \rightsquigarrow (\mathcal{R}, \Delta_R)\end{array}}{\kappa_1 \wedge \pi_1 \vdash \mathtt{U_c}(\bar{w}, \bar{z}\#) * \kappa_2 \wedge \pi_2 \rightsquigarrow (\sigma{\wedge}\mathcal{R}, \Delta_R)}$$

Figure 4-5: Bi-Abductive Folding.

(and their root pointers) are inferred based on κ_{12} and the #-annotated parameters $\bar{z}$. The assumption is generated in the third line and finally, the residual heap is used to discharge the remaining heaps of RHS.

4.5 Hoare Rules for Shape Inference

By supporting relational assumptions in the entailment prover, we were able to support shape analysis within an existing program verifier. We were able to leverage on the arithmetic reasoning capability of the verifier presented in [33] to move beyond purely shape analysis. Though we use C as our source language, we translate it into a core imperative language that supports a mutable heap-based data structures (*datat*) and a set of methods (*meth*). The language is presented in Figure 4-6. A method declaration includes a prototype,

$$
\begin{array}{lll}
\texttt{Program} & ::= & \mathit{datat}^* \ \mathit{meth}^* \\
\mathit{datat} & ::= & \texttt{data}\ c\ \{\ \mathit{field}^*\ \} \\
\mathit{field} & ::= & t\ v \quad t ::= \texttt{int} \mid \texttt{bool} \mid \texttt{void} \mid \texttt{c} \ \mid \ldots \\
\mathit{meth} & ::= & t\ mn\ (([\texttt{ref}]\ t\ v)^*) \quad \Phi_{pr} \ \ \Phi_{po};\ \{e\} \\
e & ::= & \texttt{NULL} \mid k^{\tau} \mid v \mid v.f \mid v{=}e \mid v.f{=}e \mid \texttt{new}\ c(v^*) \\
 & & \mid e_1; e_2 \mid t\ v;\ e \mid mn(v^*) \mid \texttt{if}\ v\ \texttt{then}\ e_1\ \texttt{else}\ e_2
\end{array}
$$

Figure 4-6: Core Imperative Language.

pre-/post-condition and its body code. Methods can have call-by-reference parameters (prefixed with `ref`). Loops are included by transforming to tail-recursive methods with `ref` parameters to capture mutable variables. Code verification is formulated as a validity proof of a Hoare-style triple $\vdash \{\Delta_{\texttt{pre}}\}\texttt{e}\{\Delta_{\texttt{post}}\}$, where Δ_{pre} denotes a formula abstracting pre-states from which program code `e` can safely run and Δ_{post} denotes a formula abstraction of all possible post states. To support shape analysis, we extend the formulation to:

$$\vdash \{\Delta_{\texttt{pre}}\}\ \texttt{e}\ \{\mathcal{R}, \Delta_{\texttt{post}}\}$$

where $\mathcal{R}$ accumulates the set of relational assumptions generated by the entailment procedure. To support inference, the specification may contain unknown predicates in preconditions, where they are classified as *pre-predicates*, or unknown predicates in postconditions, where they are classified as *post-predicates*. For clarity, we occasionally denote pre-predicates using $\texttt{U}_{\texttt{i}}^{\texttt{pre}}$ and post-predicates using $\texttt{U}_{\texttt{i}}^{\texttt{post}}$.

We list below the resulting verification rules for field access, method calls and method declaration as shown in Figure 4-7. Note that the primed variable (e.g. $\texttt{x}'$) denotes the latest value (of the program variable `x`). The formula $\Delta_1 *_{\bar{v}} \Delta_2$ denotes $\exists \bar{r} \cdot ([\bar{r}/\bar{v}']\Delta_1) * ([\bar{r}/\bar{v}]\Delta_2)$ ([33]).

The key outcome is that if a solution for the set of relational assumptions $\mathcal{R}$ can be found, the program is memory-safe and all the methods abide by their

$$\underline{\textbf{[SA-FLD-RD]}}$$
$$\dfrac{\begin{array}{c}\texttt{data}\ c\ \{t_1\ f_1, .., t_n\ f_n\} \in \texttt{Program} \quad \Delta_1 \vdash x'{\mapsto}c(v_1..v_n) \leadsto (\mathcal{R}, \Delta_3) \\ \Delta_4{=}\exists v_1..v_n \cdot (\Delta_3{*}x'{\mapsto}c(v_1..v_n){\wedge}\texttt{res}{=}v_i)\end{array}}{\vdash \{\Delta_1\}\ \texttt{x.f}_\texttt{i}\ \{\mathcal{R}, \Delta_4\}}$$

$$\underline{\textbf{[SA-METH]}}$$
$$\dfrac{\begin{array}{c}\vdash \{\Phi_{pr}{\wedge} \bigwedge(u'{=}u)^*\}\ e\ \{\mathcal{R}_1, \Delta_1\} \quad \Delta_1 \vdash \Phi_{po} \leadsto (\mathcal{R}_2, \Delta_2) \\ \Gamma = \texttt{solve}(\mathcal{R}_1{\cup}\mathcal{R}_2)\end{array}}{t_0\ mn\ ((t\ u)^*)\ \ \Phi_{pr}\ \Phi_{po}\ \{e\}}$$

$$\underline{\textbf{[SA-CALL]}}$$
$$\dfrac{\begin{array}{c}\texttt{t}_0\ mn\ ((\texttt{ref t}_\texttt{i}\ \texttt{v}_\texttt{i})_{\texttt{i=1}}^{\texttt{m-1}}, (\texttt{t}_\texttt{j}\ \texttt{v}_\texttt{j})_{\texttt{j=m}}^{\texttt{n}})\ \ \Phi_{pr}\ \ \Phi_{po};\ \{\texttt{e}\} \in \texttt{Program} \\ \rho{=}[v'_k/v_k]_{k=1}^{n} \quad \Phi'_{pr} = \rho(\Phi_{pr}) \quad W{=}\{v_1, .., v_{m-1}\} \quad V{=}\{v_m, .., v_n\} \\ \Delta \vdash \Phi'_{pr} \leadsto (\mathcal{R}, \Delta_2) \quad \Delta_3{=}(\Delta_2 \wedge \bigwedge_{i=m}^{n}(v'_i = v_i))\ \ *_{V \cup W}\ \Phi_{po}\end{array}}{\vdash \{\Delta\}\ mn(v_1, .., v_{m-1}, v_m, .., v_n)\ \{\mathcal{R}, \Delta_3\}}$$

Figure 4-7: Hoare Rules for Shape Inference.

specifications. Furthermore, we propose a bottom-up verification process which is able to incrementally build suitable predicate instantiations one method at a time by solving the collected relational assumptions $\mathcal{R}$ progressively. Our main procedure (`solve`) consists of two separate operations described in the next chapter: predicate synthesis, `pred_syn`, and predicate normalization, `pred_norm`. That is `solve`($\mathcal{R}$) = `pred_norm`(`pred_syn`($\mathcal{R}$)). After the method is successfully verified, the resulting predicate definitions Γ provide an interpretation for the unknown predicates appearing in the specifications such that memory safety is guaranteed. By returning Γ, the method verification allows the inferred definitions and specifications to be consistently reused in the verification of the remaining methods.

4.6 Soundness of Bi-Abductive Entailment

For brevity, we introduce the notation $\mathcal{R}(\Gamma)$ to denote a set of predicate instantiations Γ satisfying the set of assumptions $\mathcal{R}$. That is, for all assumptions

$\Delta \Rightarrow \Phi^g \in \mathcal{R}$, (i) Γ contains a predicate instantiation for each unknown predicate appearing in Δ and Φ^g; (ii) by interpreting all unknown predicates according to Γ, then it is provable that Δ implies Φ^g, written as $\Gamma : \Delta \vdash \Phi^g$

With respect to the abduction phase, soundness requires that if all the relational assumptions generated are satisfiable, then the entailment is valid.

Lemma 1. *Let $\mathcal{R}$ be the set of relational assumptions returned by our bi-abductive entailment checker:*

$$\Delta_{\mathtt{ante}} \vdash \Delta_{\mathtt{conseq}} \rightsquigarrow (\mathcal{R}, \Delta_{\mathtt{frame}})$$

If there exists $\Gamma{=}\{\mathtt{U}_1(\bar{\mathtt{v}}_1){\equiv}\Delta_1, ..\mathtt{U}_n(\bar{\mathtt{v}}_n){\equiv}\Delta_n\}$*, a set of instantiations for unknown predicates such that* $\mathcal{R}(\Gamma)$*, then the following entailment holds:* $\Gamma : \Delta_{\mathtt{ante}} \vdash \Delta_{\mathtt{conseq}} * \Delta_{\mathtt{frame}}$.

Proof See Appendix .2.1.

4.7 Implementation

We have implemented the proposed shape analysis within a separation logic verification system. The resulting verifier, called S2, uses the CIL infrastructure [112] to support heap-based C programs. Our translation converts each C-program to an expression-oriented core language with heap allocation. Stack-allocated data structures are mimicked as heap-allocations that are disposed at the exit of their declaration block. This allow us to guarantee memory safety for stack accesses too. Each loop is converted to a tail-recursive method, with pass-by-reference semantics for variables that are being updated. Our tool utilizes external provers, e.g. Omega [125] and Z3 [45], to discharge pure proof obligations. Our analysis modularly infers the pre/post specification for each method. It attempts to provide the weakest possible precondition to ensure memory safety (from null dereferencing and memory leaks), and the strongest possible post-condition on heap usage pattern, where possible.

We have explored the generality and efficiency of the proposed analysis through a number of small but challenging examples. We have evaluated programs which manipulate a wide range of data structures. The evaluation showed that our proposed shape analysis can handle recursive methods with an expressive fragment of shape predicate including lists (singly-, doubly-linked, nested and skip variants), trees (binary, rose and mcf) and combinations (e.g. `tll`: trees whose leaves are chained in a linked list).

4.8 Discussion

A significant body of research has been devoted to shape analysis. Most proposals are orthogonal to our work as they focus on determining shapes based on a fixed set of shape domains. For instance, the analysis in [106] can infer shape and certain numerical properties but is limited to the linked list domain. The analyses from [9, 13, 51, 66, 100, 138] are tailored to variants of lists and a fixed family of list interleavings. Likewise, Calcagno et al. [21] describe an analysis for determining lock invariants with only linked lists. Lee et al. [94] presents a shape analysis specifically tailored to overlaid data structures. In the matching logic framework, a set of predicates is typically assumed for program verification [131]. The work [5] extends this with specification inference. However, it currently does not deal with the inference of inductive data structure abstractions.

The proposal by Magill et al. [106, 135] is able to infer numerical properties. While the former is still parametric in the shape domain, the latter is not fully automatic. Similarly, the separation logic bi-abduction described in [20, 70] assumes a set of built-in or user-defined predicates. Xisa, a tool presented by Rival et. al. [28], works on programs with more varied shapes as long as structural invariant checkers, which play the role of shape definitions, are provided. A later extension [130] also considers shape summaries for procedures with the additional help of global analysis. Other similarly parameterized analysis includes [62]. In comparison, our approach is built upon the foundation of second-order bi-abductive entailment, and is able to infer unknown predicates from scratch or guided by user-supplied assertions. This set-up is therefore highly flexible, as we could support a mix of inference and verification, due to our integration into an existing verification system.

With respect to fully automatic analyses, there are [16], [68] and the Forester system [73]. Although very expressive in terms of the inferred shape classes, the analysis proposed by Guo et al. [68] relies on a heavy formalism and depends wholly on the shape construction patterns being present in the code. They describe a global analysis that requires program slicing techniques to shrink the analyzed code and to avoid noise on the analysis. Furthermore, the soundness of their inference could not be guaranteed; therefore a re-verification of the inferred invariants is required. Brotherston and Gorogiannis [16] propose a novel way to synthesize inductive predicates by ensuring both memory safety and termination. However, their proposal is currently limited to a simple imperative language without methods. A completely different approach is presented in the Forrester system [73] where a fully automated shape synthesis is described in terms of graph transformations over forest automata. Their approach is based on learning techniques that can discover suitable forest automata by incrementally constructing shape abstractions called boxes. However, their proposal is currently restricted both in terms of the analysed programs, e.g. recursion is not yet supported, and in terms of the inferred shapes, as recursive nested boxes (needed by `tll`) are not supported.

In the TVLA tradition, [129] describes an interprocedural shape analysis for cut-free programs. The approach explores the interaction between framing and the reachability-based representation. Other approaches to shape analysis include grammar-based inference, e.g. [95] which relies on inferred grammars to define the recursive backbone of the shape predicates. Although [95] is able to handle various types of structures, e.g. trees and dlls, it is limited to structures with only one argument for back pointers. [102] employs inductive logic programming (ILP) to infer recursive pure predicates. While, it might be possible to apply a similar approach to shape inference, there has not yet been any such effort. Furthermore, we believe a targeted approach would be able to easily cater for the more intricate shapes. Since ILP has been shown to effectively synthesize recursive predicates, it would be interesting to explore an integration of ILP with our proposal for inferring recursive predicates of both shape and pure properties. A recent work [64] that aims to automatically construct verification tools has implemented various proof rules for reachability and termination properties however it does not focus on the synthesis of shape abstractions. In an orthogonal direction, [47] presents an analysis for constructing precise and compact method summaries. Unfortunately, both these works lack the ability to handle recursive data structures.

Chapter 5

Derivation and Transformation of Shape Predicates

Given a set of relational assumption on *unknown* shape predicates, we propose a solver with a set of derivation rules to infer sound definitions to the shape predicates. To distinguish unknown *pre-predicates* in pre-conditions from unknown *post-predicates* in post-condition, we invoke a sound mechanism to split relational assumptions at shared base formulas. For a precise synthesis of pre-predicates, we presented a *heap guard* mechanism to capture spatial context while generating relational assumptions, we now show how to derive context of predicate definitions, and to employ such context of the definitions during performing inline. The proposed shape solver will be presented in section 5.2.

As shape analysis typically follows the structure of programs, it may produce overtly complicated result and occasionally can not be immediately used by verification systems. In our framework, the shape solver above may produce a set of inductive predicate definitions that are beyond the capability of entailment procedures. Especially, in a modular shape analysis, shape predicates inferred for specifications of callees may involve in proof obligations generated to prove the correctness of callers. Such obligations may contain different shape predicates.

However, reasoning with different shape predicates have not been well supported by existing entailment procedures of separation logic. A direct solution to handle such reasoning requires capability of *induction proving*, like cyclic proof presented in [17]. But the cyclic technique has not been extended to infer heap frame. And thus, it can not be deployed in the modular reasoning. Instead of enhancing decision procedure with such a hard requirement, we propose an indirect solution to match predicates which have different syntactical names but are semantically equivalent. More concretely, we propose a transformational approach to classify shape predicates into equivalent sets. This helps to minimize the requirement of proving among different predicates. The proposed approach has been implemented through two phases. In the first phase, it simplifies shape predicates by eliminating predicate without definition and eliminating useless parameters of predicates to increase the opportunity of predicate matching. In the second phase, it semantically matches predicates via a new lemma mechanism. As expected, inferred specifications are more concise with smaller number of equivalent sets of predicates. The proposed transformation will be presented in section 5.4.

5.1 Illustration

We would like to remind that our approach on specification inference comprises three main steps: (i) inferring relational assumptions for unknown predicates via Hoare-style verification, (ii) deriving predicates from relational assumptions, (iii) normalizing predicates. We discussed the first step, the second-order bi-abduction mechanism, in the previous chapter. In this chapter, we discuss step (ii) and step (iii), a transformation of the output from step (i). The transformation takes a set of relational assumptions over unknown predicates as input, then it will (ii) derive predicates from relational assumptions, (iii) normalize predicates.

5.1.1 The `sll2dll` Example

With the set of relational assumptions inferred by the shape analysis in Figure 4-2, we proceed to transform them as follows. For (ii), we employ a predicate derivation procedure to transform (by either equivalence-preserving or abductive steps) the set of relational assumptions into a set of predicate definitions. Sec. 5.2 gives more details on predicate derivation. For our `sll2dll` example, we initially derive the following predicate definitions (for `H` and `G`):

$$\begin{array}{lcl} \mathtt{H}(x,q) & \equiv & \mathtt{emp}\wedge x{=}\mathtt{NULL} \vee x{\mapsto}node(x_p,x_n) * \mathtt{H_p}(x_p,q) * \mathtt{H}(x_n,x) \\ \mathtt{G}(x,q) & \equiv & \mathtt{emp}\wedge x{=}\mathtt{NULL} \vee x{\mapsto}node(q,x_n) * \mathtt{G}(x_n,x) \end{array}$$

Although the definition of the shape predicate `H` is sound, it is not easy to realize that `H` is a full singly-linked list. Thus we proceed to the last step, step (iii), for a further simplification.

In the last step, we use a normalization procedure to simplify the definition of predicate `H`. Since $\mathtt{H_p}$ is discovered as a dangling predicate, the special variable $\mathcal{D}_\mathtt{p}$ corresponds to a *dangling reference* introduced:

$$\mathtt{H}(x,q) \equiv \mathtt{emp}\wedge x{=}\mathtt{NULL} \vee x{\mapsto}node(\mathcal{D}_\mathtt{p},x_n) * \mathtt{H}(x_n,x).$$

Furthermore, we can synthesize a more concise $\mathtt{H_2}$ from `H` by eliminating its useless `q` parameter:

$$\begin{array}{lcl} \mathtt{H}(x,q) & \equiv & \mathtt{H_2}(x) \\ \mathtt{H_2}(x) & \equiv & \mathtt{emp}\wedge x{=}\mathtt{NULL} \vee x{\mapsto}node(\mathcal{D}_\mathtt{p},x_n) * \mathtt{H_2}(x_n) \end{array}$$

As can be seen, the predicate $\mathtt{H_2}$ is a singly-linked list and `H` is equivalent to $\mathtt{H_2}$.

If singly-linked list (named sll) and doubly-linked list (named dll) predicates are provided in advance, the predicates `H`, $\mathtt{H_2}$ and `G` would be matched as:

$$\mathtt{H}(x,q) \equiv \mathtt{sll}(x) \qquad \mathtt{H_2}(x) \equiv \mathtt{sll}(x) \qquad \mathtt{G}(x,q) \equiv \mathtt{dll}(x,q)$$

5.1.2 The `tll` Example

Similarly, with the set of relational assumptions inferred by the shape analysis in Figure 4-3, we proceed to transform them in this chapter. The `tll` example will be used as a running example to highlight our derivation and normalization in this chapter. For a summary, at the end our transformation can synthesize the following predicate definitions (for `H` and `G`):

$$\begin{aligned}
&\mathtt{H}(x,p,t) \equiv x{\mapsto}tree(\mathcal{D}_p,\mathcal{D}_l,r,\mathcal{D}_n){\wedge}r{=}\mathtt{NULL}\\
&\qquad\qquad \vee\ x{\mapsto}tree(\mathcal{D}_p,l,r,\mathcal{D}_n){*}\mathtt{H}(l,x,lm){*}\mathtt{H}(r,x,t) \wedge r \neq \mathtt{NULL}\\
&\mathtt{G}(x,p,\mathtt{res},t) \equiv x{\mapsto}tree(p,\mathcal{D}_l,r,t) \wedge \mathtt{res}{=}x{\wedge}r{=}\mathtt{NULL}\\
&\qquad\qquad \vee\ x{\mapsto}tree(p,l,r,\mathcal{D}_n){*}\mathtt{G}(l,x,\mathtt{res},lm){*}\mathtt{G}(r,x,lm,t){\wedge}r{\neq}\mathtt{NULL}
\end{aligned}$$

The variables $\mathcal{D}_{\mathtt{l}}$, $\mathcal{D}_{\mathtt{p}}$ and $\mathcal{D}_{\mathtt{n}}$ correspond to *dangling predicates* that have not been field accessed. Note that for memory safety, the input tree x must contain at least one node and that x->l must be non-null when x->r is non-null. These requirements are captured by our synthesized pre-predicate.

5.2 Deriving Shape Predicates

Given a set of the relational assumptions, we proceed to split them into two sets: a set of relational assumptions of pre-predicates and another set of relational assumptions of post-predicates. We then apply a series of refinement steps in order to derive predicate definitions for each pre- and post-predicate.

5.2.1 Algorithm Outline

Fig. 5-1 outlines our strategy for applying the refinement steps. We use the $\boxed{\textbf{Syn-}*}$ notation for refinement rules that are described in later sub-sections. First, assumptions on pre-predicates from those on post-predicates are separated

```
function PRED_SYN( $\mathcal{R}$)
    $\Gamma \leftarrow \emptyset$
    $\mathcal{R} \leftarrow$ exhaustively apply $[\textbf{Syn-Base}]$ on $\mathcal{R}$
    $\mathcal{R}_{pre}, \mathcal{R}_{post} \leftarrow \textit{sort-group}(\mathcal{R})$
    while $\mathcal{R}_{pre} \neq \emptyset$ do
        $\mathtt{U^{pre}}, \sigma \leftarrow$ pick unknown and select related assumptions in $\mathcal{R}_{pre}$
        $\mathtt{U^{pre}_{def}} \leftarrow$ apply $[\textbf{Syn-Case}]$, $[\textbf{Syn-Group-Pre}]$,$[\textbf{Syn-Pre-Def}]$ on $\sigma$
        $\mathcal{R}_{pre}, \mathcal{R}_{post} \leftarrow$ inline $\mathtt{U^{pre}_{def}}$ in $(\mathcal{R}_{pre} \setminus \sigma)$ and $\mathcal{R}_{post}$
        $\Gamma \leftarrow \Gamma \cup \{\mathtt{U^{pre}_{def}}\}$
    end while
    while $\mathcal{R}_{post} \neq \emptyset$ do
        $\mathtt{U^{post}}, \sigma \leftarrow$ pick unknown and select related assumptions in $\mathcal{R}_{post}$
        $\mathtt{U^{post}_{def}} \leftarrow$ apply $[\textbf{Syn-Group-Post}]$, $[\textbf{Syn-Post-Def}]$ on $\sigma$
        discharge $\mathtt{U^{post}}$ obligations
        $\mathcal{R}_{post} \leftarrow \mathcal{R}_{post} \setminus \sigma$        $\Gamma \leftarrow \Gamma \cup \{\mathtt{U^{post}_{def}}\}$
    end while
    return $\Gamma$
end function
```

Figure 5-1: Shape Derivation Outline

by splitting base-case constraints in relational assumptions with both types of predicates.

5.2.2 Base Splitting of Pre/Post-Predicates

$$\frac{\begin{array}{c}[\textbf{Syn-base}]\\ \sigma:\ \mathtt{U^{pre}}(\bar{\mathtt{x}}){*}\kappa{\wedge}\pi \Rightarrow \mathtt{U^{post}}(\bar{\mathtt{y}}) \\ \sigma_1:\ \mathtt{U^{pre}}(\bar{\mathtt{x}}){\wedge}\mathtt{project}(\bar{\mathtt{x}},\pi){\Rightarrow}\mathtt{emp} \qquad \sigma_2:\ \kappa{\wedge}\pi \Rightarrow \mathtt{U^{post}}(\bar{\mathtt{y}}) \\ \kappa_{\mathtt{g}}{=}{*}\{\kappa_1 \mid \kappa_1{\in}\mathtt{heaps}(\kappa){\wedge}\mathit{pars}(\kappa_1){\cap}\bar{\mathtt{x}}{\neq}\emptyset\} \qquad \bar{\mathtt{w}}{=}\bigcup\{\mathit{pars}(\kappa_1) \mid \kappa_1{\in}\kappa_{\mathtt{g}}\} \\ \sigma_3:\ \mathtt{U^{pre}}(\bar{\mathtt{x}}){\Rightarrow}\mathtt{U^{fr}}(\bar{\mathtt{x}})\ \mathtt{@}\ \kappa_{\mathtt{g}}{\wedge}\mathtt{project}(\bar{\mathtt{x}}{\cup}\bar{\mathtt{w}},\pi) \qquad \sigma_4:\ \mathtt{U^{fr}}(\bar{\mathtt{x}}) \Rightarrow \top\end{array}}{\mathtt{if\ is\_base}(\bar{\mathtt{x}},\pi){=}\mathtt{true\ then}\ (\sigma_1{\wedge}\sigma_2)\ \mathtt{else}\ (\sigma{\wedge}\sigma_3{\wedge}\sigma_4)}$$

Figure 5-2: Shape Predicate Derivation: Base Splitting Rule

We first deal with relational assumptions of the form $\mathtt{U^{pre}}(\ldots){*}\Delta \Rightarrow \mathtt{U^{post}}(\ldots)$, which capture constraints on both a pre-predicate and a post-predicate. To allow greater flexibility in applying specialized techniques for pre-predicates or post-predicates, we split the assumption into two assumptions such that pre-predicate $\mathtt{U}^{\mathtt{pre}}$ is separated from post-predicate $\mathtt{U}^{\mathtt{post}}$. Base splitting can be formalized as in Figure 5-2. The premise contains an assumption (σ) which could be split. The conclusion captures the new relational assumptions. There are two scenarios:

(1) The first scenario takes place when the test $\mathtt{is\_base}(\bar{\mathtt{x}}, \pi)$ holds. It signifies that π contains a base case formula for some pointer(s) in $\bar{\mathtt{x}}$. Note that $\mathtt{is\_base}(\bar{\mathtt{x}}, \pi)$ holds if and only if ($\exists\ \mathtt{v}{\in}\bar{\mathtt{x}}.\ \pi \vdash \mathtt{v}{=}\mathtt{NULL}$) or ($\exists \mathtt{v}_1,\mathtt{v}_2{\in}\bar{\mathtt{x}}.\pi \vdash \mathtt{v}_1{=}\mathtt{v}_2$). In such a situation, the assumption σ is split into σ_1 and σ_2. This reflects the observation that a pre-predicate guard will likely constrain the pre-predicate to a base-case with empty heap. This scenario happens in our running example where the assumption (A1) is split to:

$$(\mathtt{A1a}).\ \mathtt{H}(x,q) \wedge \mathtt{x{=}NULL} \Rightarrow \mathtt{emp} \qquad (\mathtt{A1b}).\ \mathtt{emp} \wedge \mathtt{x{=}NULL} \Rightarrow \mathtt{G}(x,q)$$

(2) If the test $\mathtt{is\_base}(\bar{\mathtt{x}}, \pi)$ fails, there is no base case information available for us to instantiate $\mathtt{U}^{\mathtt{pre}}(\bar{\mathtt{x}})$. The assumption σ is not split and kept in the result. To have a more precise derivation, we would also record the fact that $\mathtt{U}^{\mathtt{pre}}(\bar{\mathtt{x}})$ has no instantiation under the current context. To do this, in the second line we record in $\kappa_{\mathtt{g}}$ such a heap context (related to $\bar{\mathtt{x}}$), extract in $\bar{\mathtt{w}}$ related pointers from the context, and introduce a fresh unknown predicate $\mathtt{U}^{\mathtt{fr}}$ as the instantiation for $\mathtt{U}^{\mathtt{pre}}$, as indicated by the assumption σ_3 in the third line. Note the heap guard specifies the context under which such an assumption holds. We also add σ_4 into the result, where the new predicate $\mathtt{U}^{\mathtt{fr}}$ is instantiated to the afore-mentioned memory locations (encapsulated by $\top$). Assumptions of the form $\mathtt{U}^{\mathtt{fr}}(\mathtt{p}) \Rightarrow \top$ are being used to denote dangling pointers. We also note that introducing the dangling predicate $\mathtt{U}^{\mathtt{fr}}$ into the guarded assumption σ_3 is essential to help relate non-traversed pointer fields between the pre-predicate $\mathtt{U}^{\mathtt{pre}}$ and the post-predicate $\mathtt{U}^{\mathtt{post}}$. The function $\mathit{pars}(\kappa)$ (the 2nd line) retrieves parameters: $\mathit{pars}(\mathtt{r}{\mapsto}\mathtt{c}(\bar{\mathtt{v}})) = \bar{\mathtt{v}}$, $\mathit{pars}(\mathtt{P}(\mathtt{r}, \bar{\mathtt{v}})) = \bar{\mathtt{v}}$. As an example, consider splitting

$$(\sigma_5):\ \mathtt{U}^{\mathtt{pre}}(p){*}x{\mapsto}node(p{,}n){\wedge}\mathtt{n}{=}\mathtt{NULL} \Rightarrow \mathtt{U}^{\mathtt{post}}(x).$$

The test $\mathtt{is\_base}(\{\mathtt{p}\},\mathtt{n}{=}\mathtt{NULL})$ fails. In addition to (σ_5), the splitting returns also

$$(\sigma_6):\mathtt{U}^{\mathtt{pre}}(p) \Rightarrow \mathtt{U}^{\mathtt{fr}}(p)\ @\ (x{\mapsto}node(p{,}n){\wedge}\mathtt{n}{=}\mathtt{NULL}) \qquad (\sigma_7):\mathtt{U}^{\mathtt{fr}}(p) \Rightarrow \top$$

For the `tll` example in Sec 4.3, the $[\underline{\textbf{Syn-base}}]$ transformation can be applied to assumption (4) yielding the following three new assumptions:

$(4a)$ $\mathtt{res}{\mapsto}tree(p{,}l{,}r{,}t){*}\mathtt{H}_{\mathtt{l}}(l{,}p{,}t){\wedge}\mathtt{r}{=}\mathtt{NULL}{\wedge}\mathtt{res}{=}\mathtt{x} \Rightarrow \mathtt{G}(x{,}p{,}\mathtt{res}{,}t)$

$(4b)$ $\mathtt{H}_{\mathtt{r}}(r,p,t) \wedge \mathtt{r}{=}\mathtt{NULL} \Rightarrow \mathtt{emp}$

$(4c)$ $\mathtt{H}_{\mathtt{l}}(l{,}p{,}t) \Rightarrow \mathtt{H}^{\mathtt{f}}_{\mathtt{l}}(l{,}p{,}t)\ @\ (\mathtt{res}{\mapsto}tree(p{,}l{,}r{,}t){\wedge}\mathtt{r}{=}\mathtt{NULL})$

$(4d)$ $\mathtt{H}^{\mathtt{f}}_{\mathtt{l}}(l{,}p{,}t) \Rightarrow \top$

Pre-predicate $\mathtt{H}_{\mathtt{r}}$ captures $r{=}\mathtt{NULL}$ as its base-case split. Pre-predicate $\mathtt{H}_{\mathtt{l}}$ uses a heap guard for its base context, and $\top$ to denote an un-accessed dangling heap residue encapsulated in $\mathtt{H}^{\mathtt{f}}_{\mathtt{l}}$.

5.2.3 Assumption Sorting and Partitioning

In order to allow for a convenient instantiation order, we sort and group together each set of relational assumptions pertaining to the same predicate, through *sort-group($\mathcal{R}$)*. Intuitively, the call to *sort-group($\mathcal{R}$)* will sort assumptions relevant to each predicate according to the following pattern:

1. $H(\ldots) \wedge \pi \Rightarrow H_2(\ldots)\ @\ \Delta$
2. $H(\ldots) \Rightarrow H_2(\ldots)$
3. $H(\ldots) * \Delta \Rightarrow H_2(\ldots)$
4. $H(\ldots) \Rightarrow \Delta$
5. $H(\ldots) \wedge \pi \Rightarrow \Delta$
6. $\Delta \Rightarrow G(\ldots)$

The first four forms could be used directly for in-lining if they are not self-recursive. Guarded assumptions (even if disjunctive but not self-recursive) are given high priority to facilitate their early removal by inlining. The fifth form leads to disjunctive recursive formula, and is not inlined for conciseness reason. As mentioned, post-predicates are only processed after all the pre-predicates have been synthesized. In order to decrease the number of assumptions that need to be considered during the derivation, we will try to partition the assumptions relevant to a predicate into assumptions that will used in the synthesise proces and assumptions that will be treated as proof obligations to be discharged after the predicate synthesis. As a simple example, the relational assumptions generated for the `tll` structure would be ordered and partitioned into: (i) assumptions to be used in pre-predicate synthesis $[(\mathtt{3}),(\mathtt{2}),(\mathtt{4b}),(\mathtt{4c}),(\mathtt{4d}),(\mathtt{1})]$, (ii) assumptions denoting outstanding proof obligations related to pre-predicates $\emptyset$, (iii) assumptions to be used in post-predicate synthesis $[(\mathtt{4a}),(\mathtt{5})]$, (iv) post-predicate obligations $\emptyset$. Note that we order pre-predicates before post-predicates, and would synthesize the simpler $\mathtt{H_l}$ and $\mathtt{H_r}$ predicates, before the $\mathtt{H}$ predicate. Also, the processing of post-obligation for a post-predicate is done after the synthesis of the respective post-predicate. This could generate extra assumptions for other un-sythesised predicates.

5.2.4 Deriving Pre-Predicates

$$\begin{array}{c}
\left[\textbf{Syn-Case}\right] \\
\dfrac{\begin{array}{c}\mathtt{U}(\bar{x}){\wedge}\pi_1{\Rightarrow}\Delta_1\ \mathtt{@}\ \Delta_{1g} \quad U(\bar{x}){\wedge}\pi_2{\Rightarrow}\Delta_2\ \mathtt{@}\ \Delta_{2g} \quad \pi_1{\wedge}\pi_2 \not\Rightarrow \mathtt{false} \\ \Delta_1{\wedge}\Delta_2{\Rightarrow}^{\bar{x}}_{\wedge}\Delta_3 \quad \Delta_{1g}{\wedge}\Delta_{2g}{\Rightarrow}^{\bar{x}}_{\wedge}\Delta_{3g} \quad \mathbf{sat}(\boldsymbol{\Delta_{3g}})\end{array}}{\begin{array}{c}\mathtt{U}(\bar{x}){\wedge}\pi_1{\wedge}\neg\pi_2{\Rightarrow}\Delta_1\ \mathtt{@}\ \Delta_{3g} \qquad \mathtt{U}(\bar{x}){\wedge}\pi_2{\wedge}\neg\pi_1{\Rightarrow}\Delta_2\ \mathtt{@}\ \Delta_{3g} \\ \mathtt{U}(\bar{x}){\wedge}\pi_1{\wedge}\pi_2{\Rightarrow}\Delta_3\ \mathtt{@}\ \Delta_{3g}\end{array}}
\end{array}$$

Figure 5-3: Shape Predicate Derivation: Case Split on Pre-Predicates Rule

Pre-predicates typically appear in relational assumptions under pure guards π, as illustrated below:

$$\mathtt{U}^{\mathtt{pre}}(\ldots)\wedge\pi \Rightarrow \Delta$$

In order to derive definitions for these pre-predicates, the first step is to transform the relational assumptions that overlap on their guards by forcing an explicit case analysis that generates a set of relational assumptions disjoint on their guard conditions as in Figure 5-3. For brevity, we assume a renaming of free variables to allow $\bar{\mathtt{x}}$ to be used as arguments in both assumptions. Furthermore, we use the $\Rightarrow^{\bar{\mathtt{x}}}_{\wedge}$ operator to denote a normalization of overlapping conjunction, $\Delta_1\wedge\Delta_2$ [127]. Informally, in order for $\Delta_1\wedge\Delta_2$ to hold, it is necessary that the shapes described by Δ_1 and Δ_2 agree when describing the same memory locations. Normalization thus determines the overlapping locations, Δ_c such that $\Delta_1{=}\Delta_c{*}\Delta'_1$ and $\Delta_2{=}\Delta_c{*}\Delta'_2$ and returns $\Delta_c{*}\Delta'_1{*}\Delta'_2$. We defer formal definition of $\Rightarrow^{\bar{\mathtt{x}}}_{\wedge}$ to Section 5.3.1.Once all the relational assumptions for a given pre-predicate have been transformed such that the pure guards do not overlap, we may proceed to combine them as follows:

$$\boxed{\textbf{Syn-Group-Pre}}$$
$$\frac{\mathtt{U}(\bar{x})\wedge\pi_1 \Rightarrow \Phi^g_1 \quad \mathtt{U}(\bar{x})\wedge\pi_2 \Rightarrow \Phi^g_2 \quad \pi_1\wedge\pi_2 \Rightarrow \mathtt{false}}{\mathtt{U}(\bar{x}) \wedge (\pi_1\vee\pi_2) \Rightarrow \Phi^g_1\wedge\pi_1 \vee \Phi^g_2\wedge\pi_2}$$

We shall perform this exhaustively until a single relational assumption for `U` is derived. If the assumption RHS is independent of any post-predicate it becomes the unknown pre-predicate definition.

$$\boxed{\textbf{Syn-Pre-Def}}$$
$$\frac{\mathtt{U}^{\mathtt{pre}}(\bar{x}){\Rightarrow}\Phi^g \qquad \mathtt{no\_post}(\Phi^g)}{\mathtt{U}^{\mathtt{pre}}(\bar{x}) \equiv \Phi^g}$$

$$\frac{\begin{array}{c}[\textbf{Syn-Inline}]\\ \mathtt{U^{pre}_d}(\bar{x}) \equiv \Delta_1 \,\texttt{@}\, (\kappa_1 \wedge \pi_1) \vee \ldots \vee \Delta_n \,\texttt{@}\, (\kappa_n \wedge \pi_n) \\ \mathtt{U^{pre}}(\bar{x}) \wedge \pi_a \Rightarrow (\mathtt{U^{pre}_d}(\bar{x}) * \kappa \wedge \pi) \,\texttt{@}\, (\kappa_g \wedge \pi_g) \\ \mathcal{S}_1 = \{\Delta_i * \Delta_r * \kappa_i \mid \kappa \wedge \pi \vdash \kappa_i \wedge \pi_i \leadsto (\emptyset, \Delta_r)\} \\ \mathcal{S}_2 = \{\kappa * \Delta_i \wedge \pi \,\texttt{@}\, (\kappa_i \wedge \pi_i) \mid \textbf{sat}(\kappa \wedge \kappa_\mathtt{i} \wedge \pi \wedge \pi_\mathtt{i}), \kappa \wedge \pi \not\vdash \kappa_\mathtt{i} \wedge \pi_\mathtt{i}\}\end{array}}{\mathtt{U^{pre}}(\bar{x}) \wedge \pi_a \Rightarrow \bigvee_{\Delta \in \mathcal{S}_1 \cup \mathcal{S}_2} (\Delta \,\texttt{@}\, (\kappa_g \wedge \pi_g))}$$

Figure 5-4: Shape Predicate Derivation: Inline Rule.

For the `tll` example, by the above rules, (2) and (4b) yield:

$$(6).\ \mathtt{H_r}(r,p,t) \equiv \mathtt{H}(r,x,t) \wedge \mathtt{r} \neq \mathtt{NULL} \,\texttt{@}\, x \mapsto \mathtt{tree}(p,l,r,n) \vee \mathtt{emp} \wedge \mathtt{r} = \mathtt{NULL}$$

Similarly, (3) and (4c) derives $\mathtt{H_l}$:.

$$(7).\ \mathtt{H_l}(l,p,t) \equiv \mathtt{H}(l,x,lm) \,\texttt{@}\, x \mapsto \mathtt{tree}(p,l,r,n) \wedge \mathtt{r} \neq \mathtt{NULL}$$
$$\vee\ \mathtt{H^f_l}(l,p,t) \,\texttt{@}\, x \mapsto \mathtt{tree}(p,l,r,n) \wedge \mathtt{r} = \mathtt{NULL}$$

This conversion of each derived relational assumption into a definition for its pre-predicate is done without any weakening. As the derived predicate may contain heap guards, we may remove them by inlining predicate occurrences with the relevant heap context. For example, relational assumption (1) for unknown predicate $\mathtt{H}$ contains a heap context that would allow $\mathtt{H_r}$ and $\mathtt{H_l}$ to be safely inlined, giving:

$$\mathtt{H}(x,p,t) \equiv x \mapsto \mathtt{tree}(x_p,l,r,x_n) * \mathtt{H}(l,x,lm) * \mathtt{H}(r,x,t) * \mathtt{H_p}(x_p,p,t) * \mathtt{H_n}(x_n,p,t) \wedge \mathtt{r} \neq \mathtt{NULL}$$
$$\vee\ x \mapsto \mathtt{tree}(x_p,l,r,x_n) * \mathtt{H^f_l}(l,x,t) * \mathtt{H_p}(x_p,p,t) * \mathtt{H_n}(x_n,p,t) \wedge \mathtt{r} = \mathtt{NULL}$$

Formally, such predicate inlining can be carried out as shown in Figure 5-4. Inlining serves two purposes: (i) allow instantiation of back pointers with the use of heap guards; (ii) minimize the number of predicates a definition relies on. As an eager optimization, the inlining discards infeasible disjuncts in which the context contradicts the guard and also drops the guard where it is already satisfied.

5.2.5 Deriving Post-Predicates

We start the derivation for a post-predicate after all pre-predicates have been derived. We can incrementally group each pair of relational assumptions on a post-predicate, as follows:

$$\boxed{\textbf{Syn-Group-Post}}$$

$$\frac{\Delta_a \Rightarrow \mathtt{U}^{\mathtt{post}}(\bar{x}) \qquad \Delta_b \Rightarrow \mathtt{U}^{post}(\bar{x})}{\Delta_a \vee \Delta_b \Rightarrow \mathtt{U}^{\mathtt{post}}(\bar{x})}$$

By exhaustively applying the above reduction all assumptions relating to predicate $\mathtt{U}^{\mathtt{post}}$ get condensed into an assumption of the form: $\Delta_1 \vee \ldots \vee \Delta_n \Rightarrow \mathtt{U}^{\mathtt{post}}(\bar{x})$. This may then be used to confirm the post-predicate by generating the following predicate definition:

$$\boxed{\textbf{Syn-Post-Def}}$$

$$\frac{\Delta_1 \vee \ldots \vee \Delta_n \Rightarrow \mathtt{U}^{\mathtt{post}}(\bar{x})}{\mathtt{U}^{\mathtt{post}}(\bar{x}) \;\equiv\; \Delta_1 \vee \ldots \vee \Delta_n}$$

Using this, we can combine (4a) and (5) to give:

$$\begin{array}{l}\mathtt{G}(x,p,\mathtt{res},t) \equiv x{\mapsto}\mathtt{tree}(p,l,r,t){*}\mathtt{H}_1(l,x_h,t_h) \wedge \mathtt{res{=}x{\wedge}r{=}NULL} \\ \qquad\qquad \vee x{\mapsto}\mathtt{tree}(p,l,r,n){*}\mathtt{G}(r,x,l_m,t){*}\mathtt{G}(l,x,\mathtt{res},l_m){*}\mathtt{H_n}(n,x_h,t_h){\wedge}\mathtt{r{\neq}NULL}\end{array}$$

5.2.6 Obligation for Post-Predicates

Memory locations abstracted by post-predicates may be further accessed after a recursive call. This may lead to relational assumptions of the following form.

$$\mathtt{U}^{\mathtt{post}}(\ldots) \wedge \pi \Rightarrow \Delta$$

We regard this as an obligation that has to be proven, and at the same time it could also be used to infer the definition of unknown post-predicates that were generated. As an example, consider the following post-predicate relational obligation:

$$\mathtt{G}(r,x){\wedge}x{\neq}\mathtt{NULL} \Rightarrow r{\mapsto}\mathtt{node}(r_v,r_n) * \mathtt{G_r}(r_n,x\#){*}\mathtt{G_x}(x,r\#)$$

This obligation introduces two extra unknown post-predicates $\mathtt{G_r}$ and $\mathtt{G_x}$. In order to synthesize definitions for them, it is possible to leverage on earlier synthesized definitions for $\mathtt{G}$. Thus, if such a definition exists, say:

$$\begin{aligned}\mathtt{G}(r,x) \ &\equiv x{\mapsto}\mathtt{node}(v,\mathtt{NULL}) \wedge \mathtt{r} = \mathtt{x}\\ &\vee \ x{\mapsto}\mathtt{node}(x_v,x_n) * \mathtt{G_x}(x_n,r) * r{\mapsto}\mathtt{node}(\_,\mathtt{NULL})\end{aligned}$$

We can re-use our second-order entailment to prove the earlier post-predicate obligation, which generates the following set of relational assumptions on $\mathtt{G_r}$ and $\mathtt{G_x}$.

$$\begin{array}{lcl} r{=}\mathtt{NULL} & \Rightarrow & \mathtt{G_r}(r,\_)\\ x{=}r \wedge x{\neq}\mathtt{NULL} & \Rightarrow & \mathtt{G_x}(x,r)\\ x{\mapsto}\mathtt{node}(\_,x_n) * \mathtt{G_x}(x_n,r) & \Rightarrow & \mathtt{G_x}(x,r)\end{array}$$

Subjecting them to synthesis for post-predicate yields:

$$\begin{array}{lcl} \mathtt{G_r}(r,\_) & \equiv & r{=}\mathtt{NULL}\\ \mathtt{G_x}(x,r) & \equiv & x{=}r \wedge x{\neq}\mathtt{NULL} \ \vee \ x{\mapsto}\mathtt{node}(\_,x_n) * \mathtt{G_x}(x_n,r)\end{array}$$

More formally,

$$\frac{\begin{array}{c}[\underline{\textbf{Syn-Post-Obl}}]\\ \mathtt{U^{post}}(\bar{v}){\wedge}\pi \Rightarrow \Delta \quad (\mathtt{U^{post}}(\bar{v}){\equiv}\Delta_{\mathtt{U^{post}}}) \in \Gamma \quad \Delta_{\mathtt{U^{post}}}{\wedge}\pi \vdash \Delta \rightsquigarrow (\mathcal{R},\Delta_{\mathtt{f}})\end{array}}{\mathcal{R}}$$

$$\pi \wedge \Delta \Rightarrow^{\bar{v}}_{\wedge} (\Delta \wedge \pi, \{\ \}) \qquad \Delta \wedge \pi \Rightarrow^{\bar{v}}_{\wedge} (\Delta \wedge \pi, \{\ \})$$

$$\frac{\Delta_1 \wedge \Delta_3 \Rightarrow^{\bar{v}}_{\wedge} (\Delta_5, \mathcal{S}_1) \qquad \Delta_2 \wedge \Delta_4 \Rightarrow^{\bar{v}}_{\wedge} (\Delta_6, \mathcal{S}_2)}{(\Delta_1 \vee \Delta_2) \wedge (\Delta_3 \vee \Delta_4) \Rightarrow^{\bar{v}}_{\wedge} (\Delta_5 \vee \Delta_6, \mathcal{S}_1 \cup \mathcal{S}_2)}$$

$$\frac{\begin{array}{c}\texttt{P is known} \quad x{\in}\bar{v} \quad \bar{y} \cap \bar{v} = \emptyset \quad \rho{=}[\bar{y} \mapsto \bar{z}] \\ \Delta_1 \wedge \rho\ \Delta_2 \Rightarrow^{\bar{v}\cup\bar{z}}_{\wedge} (\Delta_3, \mathcal{S})\end{array}}{\texttt{P}(x, \bar{z}) * \Delta_1 \wedge \texttt{P}(x, \bar{y}) * \Delta_2 \Rightarrow^{\bar{v}}_{\wedge} (\texttt{P}(x, \bar{z}) * \Delta_3, \mathcal{S})}$$

$$\frac{\begin{array}{c}\texttt{U}_1, \texttt{U}_2\ \texttt{are dangling} \quad x \in \bar{v} \quad \rho = [\bar{y} \mapsto \bar{z}] \quad \bar{y} \cap \bar{v} = \emptyset \\ \Delta_1 \wedge \rho\ \Delta_2 \Rightarrow^{\bar{v}\cup\bar{z}}_{\wedge} (\Delta_3, \mathcal{S}) \qquad \mathcal{S}_1 = \mathcal{S} {\cup} \{\texttt{U}_1 {\equiv} \texttt{U}_2\}\end{array}}{\texttt{U}_1(x, \bar{z}) {*} \Delta_1 \wedge \texttt{U}_2(x, \bar{y}) {*} \Delta_2 \Rightarrow^{\bar{v}}_{\wedge} (\texttt{U}_1(x, \bar{z}) {*} \Delta_3, \mathcal{S}_1)}$$

$$\frac{x{\in}\bar{v} \quad \rho = [\bar{y} \mapsto \bar{z}] \quad \bar{y}{\cap}\bar{v}{=}\emptyset \quad \Delta_1 \wedge \rho\ \Delta_2 \Rightarrow^{\bar{v}\cup\bar{z}}_{\wedge} (\Delta_3, \mathcal{S})}{x{\mapsto}c(\bar{z}) * \Delta_1 \wedge x{\mapsto}c(\bar{y}) * \Delta_2 \Rightarrow^{\bar{v}}_{\wedge} (x{\mapsto}c(\bar{z}) * \Delta_3, \mathcal{S})}$$

Figure 5-5: Conjunctive Unification Rules.

5.3 Unification

5.3.1 Conjunctive Unification

When describing the pre-predicate derivation, we observed that there is a need for a normalization operation for formulas $\Delta_1 \wedge \Delta_2$ to ensure the result is within the logic fragment described in Sec. 4.2. We obtain this normalization through a *conjunctive unification* step, $\Rightarrow^{\bar{v}}_{\wedge}$. Informally, in order for $\Delta_1 \wedge \Delta_2$ to be satisfiable, to describe at least one feasible heap, it is necessary that the shapes described by Δ_1 and Δ_2 agree when describing the same memory locations. Based on this observation, it is possible to construct a possibly stronger approximation for $\Delta_1 \wedge \Delta_2$ expressed in our logic fragment by unifying the common heap locations as presented in Figure 5-5. To streamline the unification process, the $\Rightarrow_{\wedge}$ operation is parameterized with a set of variables $\bar{v}$ which denotes the set of possibly common memory locations. We seed this parameter initially with the set of arguments of the predicate under construction.

In the process of unifying predicate instances or heap nodes, the transformation on one hand modifies the formulas by applying a substitution of the arguments, and on the other constructs extra constraints on dangling predicates, thus strengthening the result. We observe however that this does not affect the soundness of the result as this strengthening is restricted to pre-condition predicate definitions. By imposing an equality constraint on dangling predicates the resulting definitions become more concise, with fewer extra predicates being synthesized.

Naturally, this strengthening *may* lead to a contradiction which we will consider as a failure of shape analysis, due to contradictory scenarios. An example is illustrated below:

$$\texttt{x}{=}\texttt{NULL} \wedge \texttt{x}{\mapsto}\texttt{node}(\texttt{p},\texttt{n}) \Rightarrow^{\texttt{x}}_{\wedge} (\texttt{false}\,, \{\ \})$$

The formula above is contradicted since it is required that the pointer x both equals to `NULL` and points to an allocated object.

Conjunctive unification may also lead to a satisfiable program state, but it is allowed to impose some equivalent constraints on the unknown predicates. This is helpful for obtaining more concise specification with fewer synthesized predicates.

5.3.2 Disjunctive Unification

We propose to apply *disjunctive unification* to derive more concise definitions for post-predicate. The aim here is to factor out common constraints on disjunctive branches of a given post-predicate:

$$x{\mapsto}c(\bar{y}) * \Delta_1 \vee x{\mapsto}c(\bar{y}) * \Delta_2 \Rightarrow \mathtt{U}(x,\bar{a})$$

$$\Rightarrow_{\vee} \left\{ \begin{array}{l} x{\mapsto}c(\bar{y}){*}\mathtt{R}(x,\bar{a},\bar{y}) \Rightarrow \mathtt{U}(x,\bar{a}) \\ \Delta_1 \vee \Delta_2 \Rightarrow \mathtt{R}(x,\bar{a},\bar{y}) \end{array} \right.$$

so that common heap terms in disjunct can be abstracted.

5.4 Normalizing Shape Predicates

In this chapter, we propose a preliminary investigation into normalizing inferred shape predicates. Our primary aims is to reduce the complexity of the inferred predicates.

Shape predicates are either provided by programmers or derived by shape analysis (like in Chapter 4). Before feeding those to program verification, we propose to apply normalization to transform each predicate definition to its most concise form. Given a set of shape predicates Γ, our current method uses four key steps:

```
function PRED_NORM($\Gamma$)
    $\Gamma_1 \leftarrow$ process-dangling-and-unused-preds $\Gamma$
    $\Gamma_2 \leftarrow$ eliminate-useless-parameters $\Gamma_1$
    $\Gamma_3 \leftarrow$ perform-predicate splitting on $\Gamma_2$
    return reuse-predicates $\Gamma_3$
end function
```

5.4.1 Detecting and Eliminating Dangling Predicates

We have seen how relational assumptions are soundly transformed into predicate definitions. However, it is still possible for some pre-predicates not to have any definition. As mentioned in Section, 4.3, these *dangling* predicates denote fields that were not accessed. Though it is safe to drop such predicates (by frame rule), we keep them to capture linking information between pre- and post-conditions.

In this predicate normalization step, we associate each dangling predicate $\mathtt{U}(\mathtt{x},..)$ encapsulating a pointer that is instantiated and not an argument of the current method, with a logical variable $\mathcal{D}_{\mathtt{U}}$ denoting such a predicate instance. With this extra notation, in effect, we are making explicit that the addresses pointed to by such fields have neither been read nor written to during the execution of its method. Thus, these logical variables appear in the precondition, and after the execution, may also appear in the postcondition. For example, in Section 5.2 we showed how definitions for $\mathtt{H}$, $\mathtt{G}$, $\mathtt{H_n}$ and $\mathtt{H_p}$ can be obtained. Since $\mathtt{H^f_l}$, $\mathtt{H_n}$ and $\mathtt{H_p}$ are dangling predicates, we can further refine the definitions for $\mathtt{H}$, $\mathtt{G}$ as follows:

$$\begin{array}{l}\mathtt{H}(x,p,t) \equiv x{\mapsto}tree(\mathcal{D}_p,\mathcal{D}_l,r,\mathcal{D}_n) \wedge \mathtt{r{=}NULL} \\ \quad \vee\ x{\mapsto}tree(\mathcal{D}_p,l,r,\mathcal{D}_n){*}\mathtt{H}(l,x,lm){*}\mathtt{H}(r,x,t){\wedge}\mathtt{r{\neq}NULL}\end{array}$$

$$\begin{array}{l}\mathtt{G}(x,p,\mathtt{res},t) \equiv x{\mapsto}tree(p,\mathcal{D}_l,r,t){\wedge}\mathtt{res{=}x}{\wedge}\mathtt{r{=}NULL} \\ \quad \vee x{\mapsto}tree(p,l,r,\mathcal{D}_n){*}\mathtt{G}(l,x,\mathtt{res},l_m){*}\mathtt{G}(r,x,l_m,t){\wedge}\mathtt{r{\neq}NULL}\end{array}$$

Notice that the marking of pointers as dangling is guided by the context and thus reflects the access patterns. For example the left field in both $\mathtt{H}$ and $\mathtt{G}$ predicates, in the branch with `NULL` right subtree, are marked as dangling since the method does not accessed any of these left fields, thus imposing no constraints over these subtrees. Similarly, in the post predicate $\mathtt{G}$ the `next` field is dangling for nodes that are not leaves as those fields have not been accessed. We can formalize these steps as follows:

$$\frac{\begin{array}{c}\mathtt{U}(\bar{y}) \equiv (\mathtt{U}_d(x,\bar{v}){*}\kappa{\wedge}\pi) \,\mathtt{@}\, (\kappa_g{\wedge}\pi_g) \ \vee\ \Phi^g \quad \mathtt{x} \notin \bar{\mathtt{y}} \quad \mathtt{U_d}(\mathtt{x},\bar{\mathtt{v}}) \equiv \top \\ \textit{fresh}\ \mathcal{D}_{\mathtt{U_d}} \qquad \delta{=}\{(\mathcal{D}_{\mathtt{U_d}}/\mathtt{x})\}\end{array}}{\mathtt{U}(\bar{y}) \equiv ((\kappa{\wedge}\pi)[\delta]) \,\mathtt{@}\, (\kappa_g{\wedge}\pi_g) \vee \Phi^g}$$

This step consists of (a) identifying each unknown predicate with no definition (i.e. $\mathtt{U}_d(x,\bar{v}) \equiv \top$); (b) associating the field with a unique logical variable denoting the set of all possible values for that field (i.e. fresh $\mathcal{D}_{\mathtt{U}_d}$); (c) substituting each predicate instance with the logical variable. These steps denote equivalent-preserving transformations.

5.4.2 Eliminating Useless Parameters

We observe that there are cases in which predicate arguments are not used in the synthesized definitions. For example, the last two parameters of our derived pre-predicate for `tll` are redundant.

In order to simplify the definitions and improve predicate reuse, we propose to detect and eliminate such arguments. For a given predicate definition, $\mathtt{P}(\bar{x}) \equiv \Delta$, we can discover if any constraint in the predicate body involves a parameter through a standard, but sound flow analysis. Once a set of candidate arguments, $\bar{z}$, has been identified, we construct a new unknown predicate $U_z(\bar{x}')$ where $\bar{x}' = \bar{x} \setminus \bar{z}$ which can then be instantiated by running the bi-abductive entailment check on the following entailment where $\mathtt{P}'(\bar{x}) \equiv \exists \bar{z}.\Delta$:

$$\mathtt{P}'(\bar{x}) \vdash \mathtt{U}_\mathtt{z}(\bar{x}')$$

This would gather the necessary set of assumptions on $\mathtt{U}_\mathtt{z}$ that can provide a definition for the new predicate without the useless argument(s). This could be followed by an equivalent check that with the inferred definition $\mathtt{U}_\mathtt{z}(\bar{\mathtt{x}}) \vdash \mathtt{P}'(\bar{\mathtt{x}})$. Thus, ideally, the resulting definitions should be:

$$\mathtt{P}(\bar{x}) \equiv \mathtt{U}_\mathtt{z}(\bar{x}') \qquad \mathtt{U}_\mathtt{z}(\bar{x}') \equiv \Delta'$$

Using this step, we can obtain a much simpler pre-predicate for `tll`:

$$\begin{aligned}
&\mathtt{H}(x,p,t) \equiv \mathtt{H}_\mathtt{f}(x)\\
&\mathtt{H}_\mathtt{f}(x) \equiv x \mapsto tree(\mathcal{D}_p,\mathcal{D}_l,r,\mathcal{D}_n) \wedge r{=}\mathtt{NULL}\\
&\qquad\qquad \vee\ x \mapsto tree(\mathcal{D}_p,l,r,\mathcal{D}_n) * \mathtt{H}_\mathtt{f}(l) * \mathtt{H}_\mathtt{f}(r) \wedge r \neq \mathtt{NULL}
\end{aligned}$$

5.4.3 Reusing Predicates

In order to derive more concise predicate definitions, we propose an equivalence detection step that would try to match a newly inferred definition with a predicate definition previously provided or inferred. We leverage on our second-order entailment prover to perform this task, but limit its folding steps to syntactically matching of predicates that are possibly equivalent. We also use an analysis to pre-determine those predicates that are unlikely to be equivalent, or have already been processed as such. For any two synthesized predicates $\texttt{U1}(\bar{v})$ and $\texttt{U2}(\bar{w})$, we first align their parameters, and then prove two entailments $\mathit{unfold}[\texttt{U1}(\bar{v})] \vdash \texttt{U2}(\bar{w})$ and $\mathit{unfold}[\texttt{U2}(\bar{w})] \vdash \texttt{U1}(\bar{v})$. (Each *unfold* replaces a predicate instance by its definition. It ensures that our inductive proof is well-founded.) If both entailments fail, we assert the pair of predicates to be disequal. If only one of the entailments succeeds, we assert that a *predicate subsumption* has been detected. If both succeeded, we may return a further set of possibly equivalence pairs, from with we must prove the equivalence. When no more pairs of possibly equivalent predicates are found, we assert $\texttt{U1}(\bar{v}) \leftrightarrow \texttt{U2}(\bar{v})$ to indicate the equivalence of the sets of pairs of predicates that we have just proven. For our running `tll` example, we can detect the following subsumption: $G(x,\texttt{res},t) \rightarrow H(x)$. While this is not an equivalence, the newly discovered implication could serve as a lemma that may be automatically utilized by the entailment process.

5.4.4 Predicate Splitting

We illustrate our proposal through the `zip` method presented in Figure 5-6. The `zip` example is to sum up the corresponding integers from two lists. The two lists are of the same length, otherwise the program `aborts`.

The `zip` method has three return points at lines *4*, *5*, and *9*. While the

```
1.   struc node { int val; struc node *next};
2.   struc node* zip (struc node *x, struc node *y) {
3.     if (x==NULL)
4.       if (y!=NULL) abort;
5.       else return x;
6.     else {
7.       x->val = x->val+y->val;
8.       x->next = zip(x->next, y->next);
9.       return x;
10.    }
11.  }
```

Figure 5-6: Split Predicates: Code of `zip` Method.

returns at lines 5 and 9 describe normal execution, the return at line 4 captures an abnormal execution.

We classify the former execution as good scenarios and the latter execution as bad scenario such that users want to specify their requirements and verify its implementation against the specification. For the good scenarios, the shape analysis presented in section 5.2 can infer the following shape specification that guarantees memory safety for the method:

$$\texttt{requires twosame(x,y)} \qquad \texttt{ensures twosame(x,y)}{\wedge}\texttt{res=x}$$

with the following predicate for the safety of the above program:

$$\begin{aligned} \mathit{twosame}(x,y) \ \equiv \ & \texttt{x=NULL} \wedge \texttt{y=NULL} \\ & \vee\ x{\mapsto}\mathit{node}(\_,q){*}y{\mapsto}\mathit{node}(\_,r){*}\mathit{twosame}(q,r) \end{aligned}$$

Inspired by a decidable fragment of separation logic with inductive predicates presented in [76], we restrict that each predicate expresses heap region accessible via *one root* pointer. In the example, the pointers x and y in synthesized predicate `twosame` are both roots and thus `twosame` is not normalized. We are going to transform that predicate into normal form through a mechanism of lemma synthesis. Our lemma notations are adopted from [113].

With the predicate `twosame` given above , we illustrate a *split-predicate* tactic to derive, where possible, lemmas of the following form:

$$\texttt{lemma\_infer}\ [\texttt{U}_1;\texttt{U}_2]\ \texttt{twosame(x,y)} \rightarrow \texttt{U}_1\texttt{(x)}{*}\texttt{U}_2\texttt{(y)}$$

with $\texttt{U}_1$ and $\texttt{U}_2$ are *unknown* predicates that need to be inferred. The lemma above denotes known facts about valid implication over heap formula that can be used by the entailment checker. To explore such splitting, we can follow the example of useless parameter elimination where our second-order bi-abductive entailment is again used to infer definitions for unknown predicates, $\texttt{U}_1$ and $\texttt{U}_2$. Once these definitions are derived, we can even use the same entailment check to determine if the converse implication $twosame(x,y) \leftarrow U_1(x){*}U_2(y)$ holds. For this example, Using shape analysis (e.g. in the previous section), we can infer the following definition:

$$\begin{array}{lcl} twosame(x,y) & \rightarrow & U_1(x){*}\texttt{U}_2(y) \\ \texttt{U}_1(x) & \equiv & \texttt{x=NULL} \vee x{\mapsto}node(\_,n) * \texttt{U}_1(n) \\ \texttt{U}_2(x) & \equiv & \texttt{x=NULL} \vee x{\mapsto}node(\_,n) * \texttt{U}_2(n) \end{array}$$

Furthermore, to obtain an equivalent transformation, the *strengthening* lemma has to be valid:

$$\texttt{lemma\_check}\ \texttt{twosame(x,y)} \leftarrow \texttt{U}_1\texttt{(x)}{*}\texttt{U}_2\texttt{(y)}$$

However, those inferred definitions of $\texttt{HP}_1$ and $\texttt{HP}_2$ above can not be used to prove the validity of the strengthening lemma.

We only derive a weakening lemma that can be applied to post-condition, but not to pre-condition. For safely splitting pre-condition of `twosame` which captures two lists of the same length, we will need to extend our inference to capture size properties on lists.

To assure its validity, we must first detect that size property of the predicates should be captured. We extend $\mathtt{U}_1$ and $\mathtt{U}_2$ predicates with size property by using predicate extension mechanism presented in [135]

$$\begin{array}{lcl}\mathtt{U}_1(x,n) & \equiv & (\mathtt{x{=}NULL} \wedge \mathtt{n{=}0}) \vee (x{\mapsto}node(\_,p) * \mathtt{U}_1(p,n{-}1)) \\ \mathtt{U}_2(x,n) & \equiv & (\mathtt{x{=}NULL} \wedge \mathtt{n{=}0}) \vee (x{\mapsto}node(\_,p) * \mathtt{U}_2(p,n{-}1))\end{array}$$

After that, we invoke the following *strengthening* lemma synthesis:

$$\mathtt{lemma\_infer}\ [\mathtt{P}]\ \mathtt{H(x,y)} \leftarrow \mathtt{U}_1\mathtt{(x,m)} * \mathtt{U}_2\mathtt{(y,n)} \wedge \mathtt{P(m,n)}$$

with P is *unknown* predicate on pure properties that needs to be inferred

Using pure bi-abduction mechanism [135], the lemma above is valid when the following set of constraints has a solution:

$$\begin{array}{l}\mathtt{P(m,n)} \Rightarrow \mathtt{m{=}0} \wedge \mathtt{n{=}0} \\ \mathtt{P(m,n)} \Rightarrow \mathtt{m}_1\mathtt{{=}m{-}1} \wedge \mathtt{n}_1\mathtt{{=}n{-}1} \wedge \mathtt{P(m}_1\mathtt{,n}_1\mathtt{)}\end{array}$$

By invoking a fixpoint computation (e.g. [124]), we obtain

$$\mathtt{P(m,n)} \ \equiv\ \mathtt{m{=}n}$$

Finally, we synthesize the following lemma:

$$\mathtt{lemma\ H(root,y)} \leftrightarrow \mathtt{U}_1\mathtt{(root,m)} * \mathtt{U}_1\mathtt{(y,n)} \wedge \mathtt{m{=}n}$$

Using these newly synthesized predicates $\mathtt{U}_1$ and $\mathtt{U}_2$, we have more opportunity to match/reuse them with given library predicates. For example, using technique in section 5.4.3, we can match $\mathtt{U}_1$ and $\mathtt{U}_2$ with the `lln`, a linked list with length property. Finally, we generate the following specification for the `zip` method:

```
requires lln(x,m)∗ lln(y,n)∧m=n      ensures lln(x,m)∗ lln(y,n)∧m=n∧res=x
```

5.5 Soundness of Derivation and Normalization

Here we briefly outline several key soundness results, and leave some of the proof details to Appendix .2. For brevity, we introduce the notation $\mathcal{R}(\Gamma)$ to denote a set of predicate instantiations Γ satisfying the set of assumptions $\mathcal{R}$. That is, for all assumptions $\Delta \Rightarrow \Phi^g \in \mathcal{R}$, (i) Γ contains a predicate instantiation for each unknown predicate appearing in Δ and Φ^g; (ii) by interpreting all unknown predicates according to Γ, then it is provable that Δ implies Φ^g, written as $\Gamma : \Delta \vdash \Phi^g$

In order to prove the synthesis step sound we need to prove that if a set of predicate definitions is constructed then those definitions satisfy the initial set of assumptions. Since the synthesis consists of assumption refinement, predicate generation and predicate normalization, we will argue (i) assumption refinement does not introduce spurious satisfying instantiations, (ii) the generated predicates satisfy the refined assumptions, and (iii) normalization is meaning preserving.

Lemma 2. *Given a set of relational assumptions $\mathcal{R}$, let $\mathcal{R}'$ be the assumption set obtained by applying any of the refinement steps* [**Syn-Base**], [**Syn-Case**], [**Syn-Group-Pre**], [**Syn-Inline**], *or* [**Syn-Group-Post**], *then for any Γ such that $\mathcal{R}'(\Gamma)$, we have $\mathcal{R}(\Gamma)$.*

Proof See Appendix .2.2.

Lemma 3. *Given a set of relational assumptions $\mathcal{R}$ with only one pre-assumption, $\mathtt{U^{pre}}(\bar{\mathtt{v}}) \Rightarrow \Phi^g$, on a predicate $\mathtt{U^{pre}}$, if a final solution Γ is produced by our algorithm, then $(\mathtt{U^{pre}}(\bar{\mathtt{v}}) \equiv \Phi^g) \in \Gamma$. Similarly, given a set of relational assumptions $\mathcal{R}$ with* *only one post-assumption on* $\mathtt{U^{post}}{\Rightarrow}\Phi^g$*, if a final solution* Γ *is produced, then* $(\mathtt{U^{post}}(\bar{\mathtt{v}}){\equiv}\Phi^g){\in}\,\Gamma$*.*

Proof See Appendix .2.3.

While the above lemmas state that the constructed definitions satisfy the constraints used to construct them, it must also be shown that all assumptions have been considered.

Lemma 4. *Given a set of assumptions* $\mathcal{R}$*, if the* $\mathtt{pred\_syn}(\mathcal{R})$ *produces a set of instantiations* Γ *then* $\mathcal{R}(\Gamma)$*.*

Proof Sketch : By case analysis on the assumption structure we show that all assumptions have either been proven or incorporated into predicate definitions. The full proof is presented in Appendix .2.4. □

Lastly, it is needed to show that for a given set of assumptions, $\mathcal{R}$, and a set of instantiations Γ that satisfies $\mathcal{R}$, the normalization of Γ produces a set of instantiations that still satisfies $\mathcal{R}$.

Lemma 5. *Given a set of assumptions* $\mathcal{R}$ *and a predicate instantiation* Γ *such that* $\mathcal{R}(\Gamma)$*, let* Γ' *be the set of predicate instantiations such that* $\Gamma' = \mathtt{pred\_norm}(\Gamma)$*, then* $\mathcal{R}(\Gamma')$*.*

Proof Sketch : We show that by construction the normalization steps are meaning preserving. See Appendix .2.5 for the full proof. □

Theorem 5.5.1 (Soundness)**.** *Given two separation logic formulas with unknown predicates* $\Delta_{\mathtt{ante}}$ *and* $\Delta_{\mathtt{conseq}}$*, if*

$$\Delta_{\mathtt{ante}} \vdash \Delta_{\mathtt{conseq}} \rightsquigarrow (\mathcal{R}, \Delta_{\mathtt{frame}}) \quad \mathtt{and} \quad \Gamma{=}\mathtt{pred\_norm}(\mathtt{pred\_syn}(\mathcal{R}))$$

then $\Gamma : \Delta_{\mathtt{ante}} \vdash \Delta_{\mathtt{conseq}} * \Delta_{\mathtt{frame}}$

Proof It follows from the structure of $\mathtt{pred\_syn}$, $\mathtt{pred\_norm}$ and Lemmas 1 to 5.

5.6 Towards Complete Specification Inference

We are beginning to work towards complete specification inference. In this section, we give a preliminary discussion on extending the shape analysis to this research direction.

5.6.1 Enhancing Second-order Bi-Abduction

The [**SO-ENTAIL-UNFOLD**] rule, presented in section 4.4, infers missing information that is safe to establish pointer field accesses. To support specification inference for both safe and erroneous scenarios, we need to revise this rule to infer additional information that causes memory errors on those field accesses. More concretely, we need to enhance the abduction to infer missing *guards* that is either consistent with heaps of RHS (for safety) or inconsistent with heaps of RHS (for error). As a result, the revised [**SO-ENTAIL-UNFOLD**] is able to generate relational assumptions for both safety and errors.

Computing the guards above relies on a reduction, called **eXPure**, from a separation logic formula to a first-order logic formula. To implement this reduction, we make use of the decision procedure presented in [18] to compute shape predicate invariants. That procedure provides an algorithm to compute fix points (as *exact* pure formula) of a separation logic formula with inductive shape predicates. With the fragment of separation logic in section 4.2 which consists of heaps and only (dis)equality pure constraints, this computation is sound, complete and terminating. We illustrate the reduction through the two following examples:

$$\textbf{eXPure}(x{\mapsto}snode(\texttt{NULL}) \wedge z{=}\texttt{NULL}) \rightsquigarrow x{\neq}\texttt{NULL} \wedge z{=}\texttt{NULL}$$
$$\textbf{eXPure}(x{\mapsto}snode(y){*}y{\mapsto}snode(\texttt{NULL})) \rightsquigarrow x{\neq}\texttt{NULL}{\wedge}y{\neq}\texttt{NULL}{\wedge}x{\neq}y$$

With this **eXPure** reduction, we now give two entailment checks to illustrate the extension of the [**SO-ENTAIL-FOLD**] as discussed above.

Example 1.

$$\mathtt{H}(x) \vdash x{\mapsto}\mathtt{snode}(p)$$

In this entailment, we propose to perform bi-abduction to get either a valid or an invalid result. For a valid check, we infer a guard condition that is necessary for the unknown predicate H to establish the RHS. We make use of error status values presented in Chapter 3 to explicitly express status of states $\surd$ for safety and $\mho$ for errors. For example, from $x{\mapsto}\mathtt{snode}(p)$ of RHS, we can infer a guard that is consistent with the RHS as follows: $\pi_g \equiv \mathbf{eXPure}(x{\mapsto}\mathtt{snode}(p))$. That yields $\pi_g{\equiv}x{\neq}\mathtt{NULL}$. Like the present mechanism, we can infer the following relational assumption:

$$\sigma_{\mathtt{s}} \ \equiv \ \mathtt{H}(x) \Rightarrow x{\mapsto}\mathtt{snode}(p){*}\mathtt{H_p}(p) \ \ \surd$$

and the residue is $\Delta_s{=}x{\mapsto}\mathtt{snode}(p){*}\mathtt{H_p}(p){\wedge}x{\neq}\mathtt{NULL} \ \ \surd$.

For an invalid check, we infer a guard condition that is unsatisfiable for the unknown predicate H to establish the RHS. For example, from $x{\mapsto}\mathtt{snode}(p)$ of RHS, we can infer a guard as the negation of π_g that is inconsistent with the RHS. With this guard, we can infer the following relational assumption:

$$\sigma_{\mathtt{e}} \ \equiv \ \mathtt{H}(x) \Rightarrow \mathtt{emp}{\wedge}\mathtt{x}{=}\mathtt{NULL} \ \ \surd$$

and the residue is $\Delta_e{=}\mathtt{emp} \wedge x{=}\mathtt{NULL} \ \ \mho$.

Finally, the output of the entailment check is the set of the two above states, as follows: $\{(\Delta_{\mathtt{s}}, \sigma_{\mathtt{s}}), (\Delta_{\mathtt{e}}, \sigma_{\mathtt{e}})\}$.

Example 2.

$$\mathtt{H}(x){\wedge}\mathtt{x}{\neq}\mathtt{NULL} \vdash x{\mapsto}\mathtt{snode}(p)$$

Compared to the entailment 1, this entailment already has the pure constraint x≠NULL in its LHS. This constraint contradicts with the guard condition of the invalid case (x=NULL) and prevents to generate the relational assumption for invalid result. More concrete, the result for invalid scenario is as follows:

$$\sigma_{\mathtt{e}} \;\equiv\; \mathtt{H}(x){\wedge}\mathtt{x}{\neq}\mathtt{NULL} \Rightarrow \mathtt{emp}{\wedge}\mathtt{x}{=}\mathtt{NULL} \;\; \surd$$

and the residue is $\Delta_e{=}\mathtt{emp}{\wedge}\mathtt{x}{\neq}\mathtt{NULL}{\wedge}x{=}\mathtt{NULL}$ ℧. The state Δ_e is unreachable and will be eliminated. Therefore, the output of this entailment check contains only the state of the valid scenario: $\{(\Delta_s, \sigma'_s)\}$, whereby

$$\sigma'_{\mathtt{s}} \;\equiv\; \mathtt{H}(x){\wedge}\mathtt{x}{\neq}\mathtt{NULL} \Rightarrow x{\mapsto}\mathtt{snode}(p){*}\mathtt{H_p}(p) \;\; \surd$$

5.6.2 Enhancing Transformation

For complete specification inference, we extend the shape analysis to collect set of states generated by the new second-order bi-abductive entailment procedure. This set includes states with $\surd$ status for safety specification as well as states with ℧ status for error specification. From these states, we collect their relational assumptions to construct multiple specifications. We illustrate this enhancing shape analysis over the following examples.

foo Example.

```
struct snode* foo(struct snode* x){
  return x->next;
}
```

The *foo* example returns the *next* pointer of the input *x*. At the starting point, our analysis will generate the following specification with two unknown predicates $\mathtt{H_1}$ and $\mathtt{G_1}$ that are need to be inferred:

$$\mathtt{requires}\ \mathtt{H_1}(x) \qquad \mathtt{ensures}\ \mathtt{G_1}(x, res);$$

At the field access, our analysis generates the following proof obligation:

$$\mathtt{H_1}(x) \vdash x{\mapsto}\mathtt{snode}(p)$$

Similar to the Example 1, the enhancing second-order bi-abduction produces a set of two states $\{(\Delta_s, \sigma_s), (\Delta_e, \sigma_e)\}$. Assumptions of these states are collected for specification inference. Only safe states (i.e. $\{(\Delta_s, \sigma_s))$ will be forwarded to analyse the rest of program. At the exit point, our shape analysis will generate the following proof obligations:

1. $x{\mapsto}\mathtt{snode}(p){*}\mathtt{H_p}(p){\wedge}x{\neq}\mathtt{NULL}{\wedge}\mathtt{res}{=}p\ \ \surd \vdash \mathtt{G_1}(x, res)$
2. $\mathtt{emp} \wedge x{=}\mathtt{NULL}\ \ \mho \vdash \mathtt{G_1}(x, res)$

The enhancing second-order bi-abduction produces one state for each entailment check above. Their assumptions are as follows:

$$\begin{array}{lcl}
\sigma_{\mathtt{s2}} & \equiv & x{\mapsto}\mathtt{snode}(p){*}\mathtt{H_p}(p){\wedge}x{\neq}\mathtt{NULL}{\wedge}\mathtt{res}{=}p\ \ \surd \Rightarrow \mathtt{G_1}(x, res) \\
\sigma_{\mathtt{e2}} & \equiv & \mathtt{emp} \wedge x{=}\mathtt{NULL}\ \ \mho \Rightarrow \mathtt{G_1}(x, res)
\end{array}$$

Finally, our shape analysis collects these four relational assumptions, and hands them to the transformation engine. This engine will generate the definition for $\mathtt{H_1}$ and $\mathtt{G_1}$ as follows:

$$\begin{array}{ll}
\mathtt{H_1(x)} \equiv & \mathtt{emp}{\wedge}x{=}\mathtt{NULL}\surd\ \ \vee\ \ x{\mapsto}snode(\mathtt{DP})\ \surd \\
\mathtt{G_1(x, res)} \equiv & \mathtt{emp}{\wedge}x{=}\mathtt{NULL}\ \mho\ \ \vee\ \ x{\mapsto}snode(\mathtt{DP}){\wedge}\mathtt{res}{=}\mathtt{DP}\surd
\end{array}$$

Based on these predicate definitions, the analysis will generate the following multiple specifications:

$$\begin{array}{l}
\mathtt{requires}\ x{=}\mathtt{NULL}\quad \mathtt{ensures}\ (\mathtt{true}\,)\ \mho \\
\mathtt{requires}\ \mathtt{x}{\mapsto}\mathtt{snode(DP)}\quad \mathtt{ensures}\ \mathtt{x}{\mapsto}\mathtt{snode(DP)} \wedge \mathtt{res}{=}\mathtt{DP}\ \surd;
\end{array}$$

Finally, we may need to enhance the normalization to transform these multiple specifications into complete specification. The expected complete specification we would like to infer is:

```
case { x=NULL →  ensures (true ) ℧;
  x≠NULL →
    requires  x↦snode(DP)
    ensures   x↦snode(DP) ∧ res=DP √;
 }
```

goo Example.

```
struct snode∗ goo(struct snode∗ x){
  return x->next->next;
}
```

The *goo* example returns the pointer at the second *next* fields of the input pointer. Our analysis will generate the following specification with two unknown predicates $\mathtt{H}_2$ and $\mathtt{G}_2$ that are need to be inferred:

$$\texttt{requires}\ \mathtt{H}_2(x) \qquad \texttt{ensures}\ \mathtt{G}_2(x, res);$$

The shape analysis of the first field access is similar to the one in example *foo*. For example, it returns a set of two states: one for safety and another for error.While the latter will not be forwarded, the former will be used to analyzed the second field access. Similarly, this analysis also generates two more states, one for safety another for error. Therefore, there are three states at the exit point, two for errors and one for safety. The proving for postcondition is similar to the *foo* example.

Our approach would infer the following definitions for $\mathtt{H}_2$ and $\mathtt{G}_2$

$$\mathtt{H}_2(\mathtt{x}) \equiv \mathtt{emp}{\wedge}x{=}\mathtt{NULL}\surd \vee x{\mapsto}snode(\mathtt{NULL})\surd \vee x{\mapsto}snode(\mathtt{p}){*}p{\mapsto}snode(\mathtt{DP})\surd$$

$$\mathtt{G}_2(\mathtt{x},\mathtt{res}) \equiv \mathtt{emp}{\wedge}x{=}\mathtt{NULL}\mho \vee x{\mapsto}snode(\mathtt{NULL})\mho\vee$$
$$x{\mapsto}snode(p){*}p{\mapsto}snode(\mathtt{DP}){\wedge}\mathtt{res}{=}\mathtt{DP}\surd$$

Based on these predicate definitions, the analysis will generate the following multiple specifications:

1. $\mathtt{requires}\ x{=}\mathtt{NULL}\quad \mathtt{ensures}\ (\mathtt{true}\,)\ \mho$
2. $\mathtt{requires}\ x{\mapsto}snode(\mathtt{NULL})\quad \mathtt{ensures}\ \mathtt{true}\quad \mho;$
3. $\mathtt{requires}\ x{\mapsto}snode(\mathtt{p}) * p{\mapsto}snode(\mathtt{DP})$

   $\mathtt{ensures}\ x{\mapsto}snode(\mathtt{p}) * p{\mapsto}snode(\mathtt{DP}) \wedge \mathtt{res}{=}\mathtt{DP}\ \surd;$

Finally, the expected complete specification we would like to infer for the *goo* example is:

```
case {
  $x{=}NULL \rightarrow$  ensures (true ) $\mho$;
  $x{\neq}NULL \rightarrow$ requiresx$\mapsto$snode(p)
   case {
     $p{=}NULL \rightarrow$  ensures (true ) $\mho$;
     $p{\neq}NULL \rightarrow$
        requires  p$\mapsto$snode(DP)
        ensures   p$\mapsto$snode(DP) $\wedge$ res=DP $\surd$;
  }
```

The get_last Example.

Running shape analysis on the motivating example `get_last` in Figure 2-5, we obtains the set of relational assumptions as shown in Figure 5-7.

relational assumptions for safety
s1. $\texttt{H(x)} \Rightarrow \texttt{x}\mapsto\texttt{node(q)}*\texttt{U(q)}$ $\surd$
s2. $\texttt{U(q)}\wedge\texttt{q}\neq\texttt{NULL} \Rightarrow \texttt{H(q)}$ $\surd$
s3. $\texttt{x}\mapsto\texttt{node(q)}*\texttt{G(q,res)}\wedge\texttt{q}\neq\texttt{NULL} \Rightarrow \texttt{G(x,res)}$ $\surd$
s4. $\texttt{x}\mapsto\texttt{node(q)}*\texttt{U(q)}\wedge\texttt{q}=\texttt{NULL}\wedge\texttt{res}=x \Rightarrow \texttt{G(x,res)}$ $\surd$
relational assumptions for errors
e1. $\texttt{H(x)} \Rightarrow \texttt{emp}\wedge\texttt{x}=\texttt{NULL}$ $\surd$
e2. $\texttt{emp}\wedge\texttt{x}=\texttt{NULL}$ $\mho$ $\Rightarrow \texttt{G(x)}$

Figure 5-7: Relational Assumptions for Safety and Errors.

For this set of relational assumptions, we invoke the transformation steps (derivation and normalization) in this chapter to synthesize predicate definition for each unknown predicate. One possible result of the transformation would be as follows:

$$\texttt{H(x)} \equiv \quad \texttt{emp}\wedge x=\texttt{NULL}\surd \;\vee\; x\mapsto node(q)*\texttt{U(q)}\ \surd$$

$$\texttt{U(x)} \equiv \quad (\texttt{emp}\wedge x=\texttt{NULL} \vee x\mapsto node(q)*\texttt{U(q)})\surd$$

$$\texttt{G(x)} \equiv \quad \texttt{emp}\wedge x=\texttt{NULL}\ \mho \vee \texttt{U}_2\texttt{(x,q)}*q\mapsto node(\texttt{NULL})\surd$$

$$\texttt{U}_2\texttt{(x,s)} \equiv \quad (\texttt{emp}\wedge x=s \vee x\mapsto node(q)*\texttt{U}_2\texttt{(q,s)})\surd$$

with $\texttt{U}_2$ is a new predicate introduced during the normalization.

This extension enables us to infer the complete specification of the `get_last` method as in Figure 5-8.

```
case {
  x=NULL  →  ensures (true ) ℧;
  x≠NULL  →
     requires  x↦node(q) ∗ U(q)
     ensures   U₂(x,last) ∗ last↦node(NULL) ∧ res=last √;
 }
```

Figure 5-8: Complete Specification Inferred for `get_last` Method.

5.7 Implementation and Experimental Results

We have implemented the proposal as a separate features of S2: derivation and normalization. Those features can be used to refined the output of the shape analysis in chapter 4 or in [16]. In the rest of this chapter, we evaluated each feature on the output of shape analysis in chapter 4.

5.7.1 Two More Examples

We provide further illustrations of our proposal through two examples to highlight key features of our shape inference mechanism.

```
struct dnode { struct dnode∗ prev  struct dnode∗ next};
void append(struct dnode ∗ x, struct dnode ∗ y)
 requires H(x, y#) ensures G(x, y#)
{
  if (x->next)  append(x->next, y);
  else {
    x->next = y;
    y->prev = x;
  }
}
```

Figure 5-9: Code of append Method.

The append method shown in Figure 5-9 joins two doubly-linked lists. To guide the shape synthesis, the initial stub specification is pre-analysed with # annotations. Thus, by the same process described in the previous sections the following relational assumptions are inferred:

1 $\mathtt{H}(x{,}y\#) \Rightarrow x{\mapsto}dnode(x_p{,}x_n){*}\mathtt{H_p}(x_p{,}y\#){*}\mathtt{H_n}(x_n{,}y\#){*}\mathtt{H_y}(y{,}x\#)$

2 $\mathtt{H_n}(x_n{,}y\#){*}\mathtt{H_y}(y{,}x\#){\wedge}\mathtt{x_n}{\neq}\mathtt{NULL} \Rightarrow \mathtt{H}(x_n{,}y\#)$

3 $\mathtt{H_y}(y{,}x\#) \Rightarrow y{\mapsto}dnode(y_p{,}y_n){*}\mathtt{H_{yp}}(y_p{,}x\#){*}\mathtt{H_{yn}}(y_n{,}x\#)$

4 $\mathtt{H_n}(x_n{,}y\#){\wedge}\mathtt{x_n}{=}\mathtt{NULL} \Rightarrow \mathtt{emp}$

5 $\mathtt{H_p}(x_p{,}y\#){*}x{\mapsto}dnode(x_p{,}x_n){*}\mathtt{G}(x_n{,}y\#){\wedge}\mathtt{x_n}{\neq}\mathtt{NULL} \Rightarrow \mathtt{G}(x{,}y\#)$

6 $x{\mapsto}dnode(x_p{,}y){*}y{\mapsto}dnode(x{,}y_n){*}\mathtt{H_{yn}}(y_n{,}x\#){*}\mathtt{H_p}(x_p{,}y\#){\Rightarrow}\mathtt{G}(x{,}y\#)$

We can then synthesize the following predicate definitions:

$$\mathtt{H}(x, y){\equiv}x{\mapsto}dnode(\mathcal{D}_p, x_n){*}\mathtt{H_n}(x_n){*}y{\mapsto}dnode(\mathcal{D}_{yp}, \mathcal{D}_{yn})$$

$$\mathtt{H_n}(x_n){\equiv}\mathtt{emp}{\wedge}\mathtt{x_n}{=}\mathtt{NULL} \vee \; x_n{\mapsto}dnode(\mathcal{D}_p, x_{nn}){*}\mathtt{H_n}(x_{nn})$$

$$\mathtt{G}(x, y){\equiv}x{\mapsto}dnode(\mathcal{D}_p, y){*}y{\mapsto}dnode(x, \mathcal{D}_{yn})$$

$$\vee \; x{\mapsto}dnode(\mathcal{D}_p, x_n){*}\mathtt{G}(x_n, y){\wedge}\mathtt{x_n}{\neq}\mathtt{NULL}$$

Our shape inference mechanism manages to infer a precise (weak) pre-condition which only requires a singly-linked list for the first parameter, and a single node for the second parameter without enforcing unnecessary constraints on the rest of the locations reachable from the second parameter. Furthermore, the derived post-predicate describes a non-empty recursive list segment joined with the structure described by the second parameter. Through the use of dangling references, the derived specification permits cyclic data structures for the second parameter, and moreover guarantees that only its first node is being changed. This more precise pre/post specification subsumes the specification which uses two doubly-linked lists for the two parameters.

To illustrate a more complex data structure, consider the mutual-recursive methods shown in Figure 5-10. Those procedures validate a rose tree, whose children are linked via a doubly-linked list with parent pointers. The rose tree is defined as follows:

$$\texttt{struct mtree}\ \{\texttt{int val}; \texttt{struct mtree}^*\ \texttt{children}; \}$$

$$\texttt{struct mnode}\ \{\ \texttt{struct mtree}^*\ \texttt{child}; \texttt{struct mnode}^*\ \texttt{prev};$$
$$\texttt{struct mnode}^*\ \texttt{next}; \texttt{struct mnode}^*\ \texttt{parent}; \}$$

This checker code is special in that we are using it to validate some expected data

```
bool c_tree (struct mtree* t)
 requires H1(t) ensures G1(t) ∧ res;
{
  struct mnode* n=NULL;
  if (t->children == null) return true;
  else return c_child(t->children, NULL, t);
}

bool c_child(struct mnode* l, struct mnode* prv,struct mtree* par)
requires H2(l, prv, par) ensures G2(l, prv, par) ∧ res;
{
  if (l == null) return true;
  else
     if (l->parent == par && l->prev == prv)
        return c_child(l->next, l, par) && c_tree(l->child);
    else return false;
}
```

Figure 5-10: Example on trees on benchmark 181.mcf from SPEC2000.

structure. We use it here primarily for evaluating the precision of our synthesis method. Notice the use of mixed constraints e.g. $G_1(t) \wedge \texttt{res}$, which requires both inference and verification to work together. This comes naturally from our integration of *second-order bi-abduction* into an existing separation logic verifier with proving capability. Essentially, this code is checking that each tree node contains a pointer to a null-terminated doubly-linked list, with pointers to parent node. Our approach is able to derive the following precise and concise predicate definitions. We achieve this feat by using normalization techniques which unify

Example	size	Syn.	Veri.
SLL (delete)	9	0.23	0.2
SLL (reverse)	20	0.2	0.18
SLL (insert)	13	0.21	0.21
SLL (setTail)	7	0.18	0.18
SLL (get-last)	20	0.24	0.22
SLL-sorted (check)	8	0.26	0.23
SLL (bubblesort)	13	0.26	0.31
SLL (insertsort)	15	0.26	0.25
SLL (zip)	12	0.31	0.36
SLL-zip-leq	10	0.3	0.32
SLL + head (check)	9	0.23	0.2
SLL + tail (check)	9	0.25	0.23
skip-list$_2$ (check)	11	0.3	0.27
skip-list$_3$ (check)	17	0.45	0.45
SLL of 0/1 SLLs	8	0.24	0.25
CSLL (check)	8	0.23	0.22

Table 5.1: Experimental Results for Shape Analysis

disjuncts and semantically-equivalent predicates, where possible.

$\mathtt{H_1(t) \equiv t \mapsto mtree(v,c) * H_2(c,NULL,t)}$

$\mathtt{H_2(l,b,p) \equiv emp \wedge l=NULL}$

$\quad \mathtt{\vee\ H_2(n_1,l,p) * l \mapsto mnode(c_1,b,n_1,p) * c_1 \mapsto mtree(v,c) * H_2(c,NULL,c_1)}$

$\mathtt{G_1(t) \equiv H_1(t)}$

$\mathtt{G_2(l,b,p) \equiv H_2(l,b,p)}$

5.7.2 Expressivity

In this experiment, we evaluated the expressiveness of our inference system. For each method, we employed the shape analysis in chapter 4 to infer shape abstraction for memory safety and further derived definition for each *unknown* predicate in pre- and post-condition of the method. The experiments were performed on a machine with the Intel i7-960 (3.2GHz) processor and 16 GB of RAM. Table 5.1 and Table 5.2 presents our experimental results. For each test,

Example	size	Syn.	Veri.
CSLL (traverse)	8	0.23	0.33
CSLL of CSLLs (check)	18	0.31	0.31
SLL2DLL	8	0.19	0.19
DLL (check)	8	0.2	0.2
DLL (append)	23	0.18	0.19
CDLL (check)	9	0.25	0.23
CDLL of 5CSLLs	29	0.69	118
CDLL of CSLLs$_2$	33	0.44	0.51
tree (search)	11	0.25	0.24
tree-parent (traverse)	14	0.22	0.27
rose-tree (check)	9	0.22	0.23
swl (traverse)	19	0.39	1
mcf (check)	28	0.31	0.26
tll (traverse)	18	0.19	0.23
tll (check)	39	0.21	0.29
tll-parent (check)	19	0.24	0.29

Table 5.2: Experimental Results for Shape Analysis (cont.)

we list the name of the manipulated data structure and the effect of the verified code under the `Example` column. We use SLL for singly-linked lists, DLL for doubly-linked lists, CLL for cyclic-singly linked-lists, CDLL for cyclic-doubly linked-lists. SLL + head (tail) stands for a SLL where each element points to the head (tail) of the SLL. For nested lists, SLL of 0/1 SLLs stands for a SLL nested by a SLL of size 0 or 1, CSLL of CSLLs for CSLL nested by CSLL, CDLL of 5CSLLs for CDLL where each node is a source of five CSLL, and CDLL of CSLLs$_2$ for CDLL where each node is a nested CSLL. The skip lists subscript denotes the number of skip pointers. The swl procedure implements list traversal following the DeutschSchorr-Waite style. `rose-trees` are trees with nodes that are allowed to have variable number of children, typically stored as linked lists, and `mcf` trees [68] are rose-tree variants where children are stored in doubly-linked lists with sibling and parent pointers. The *size* column shows the number of conjuncts in the synthesized shapes. The `Syn.` column describes the synthesis times in seconds. In order to evaluate the performance of our shape synthesis, we re-verified the source programs against the inferred specifications. The verification time (in seconds) is captured in the `Veri`. column. All the inference times and most of the verification times are took under one second each. This was due to Omega taking a long time for several large proof obligations that were generated for that example. When we replace the underlying prover by Z3, the same example took about 1s to verify.

The experiments showed that our tool can handle fairly complex recursive methods, like the recursive method operating over a `tll` structure. It can synthesize shape abstractions for a large variety of data structures; from list and tree variants to combinations. Furthermore, the tool can infer shapes with mutual-recursive definitions, like the `rose-trees`, trees with nodes that are allowed to have variable number of children, typically stored in linked lists, and `mcf` trees which are rose-tree variants where children are stored in doubly-linked lists with sibling and parent pointers.

5.7.3 Experimental Results on Normalization

The normalization phase aims to reduce the complexity of inferred shape predicates. To evaluate its effectiveness, we further employ normalization feature to refine the result of the analysis in Section 5.7.2. Now, we have a synthesis on two scenarios: without (w/o) (in Section 5.7.2) and with (w) normalization. The number of conjuncts in the synthesized shapes is captured with *size* column in Table 5.3 and Table 5.4. This experiment showed that the normalization reduces 68% (169/533) the size of synthesized predicates with an overhead of 27% (8.37s/10.62s). It is able to detect and eliminate useless parameters, to split the predicate and to match with existing predicates.

Example	w/o norm.		w/ norm.		Veri.
	size	Syn.	size	Syn.	
SLL (delete)	9	0.23	2	0.23	0.29
SLL (reverse)	20	0.2	8	0.23	0.22
SLL (insert)	13	0.21	11	0.21	0.21
SLL (setTail)	7	0.18	2	0.19	0.18
SLL (get-last)	20	0.24	17	0.24	0.75
SLL-sorted (check)	8	0.26	2	0.27	0.27
SLL (bubblesort)	13	0.26	9	0.29	0.36
SLL (insertsort)	15	0.26	11	0.26	0.3
SLL (zip)	12	0.31	2	0.31	0.32
SLL-zip-leq	10	0.3	2	0.3	0.27
SLL + head (check)	9	0.23	2	0.25	0.2
SLL + tail (check)	9	0.25	2	0.26	0.18
skip-list$_2$ (check)	11	0.3	1	0.29	0.25
skip-list$_3$ (check)	17	0.45	1	0.46	0.3
SLL of 0/1 SLLs	8	0.24	1	0.24	0.23
CSLL (check)	8	0.23	2	0.2	0.21

Table 5.3: Experimental Results for Transformation Approach

5.7.4 Larger Experiments

We have evaluated S2 on real source code from the Glib open source library [2]. Glib is a cross-platform C library including non-GUI code from the GTK+ toolkit and the GNOME desktop environment. Due to our focus, we restrict our experiments to only those files which implemented heap data structures, such as SLL (gslist.c), DLL (glist.c), balanced binary trees (gtree.c) and N-ary trees (gnode.c). Our experimental results are presented in Table 5.5. LOC reports the number lines of code (exclusive of comment), #Proc (#Loop) captures the number of procedures (number of while/for loops) in each file. Also, #$\surd$ column reports the number of procedures and loops for which S2 inferred specifications that guarantee memory safety. S2 can infer specifications that guarantee memory safety for 90% of procedures and loops (200/223) [1]. To show limitation of our current proposal, we highlight a example i.e the `g_tree_insert_internal` presented

[1]Our current implementation does not support array data structures. Hence, procedures like `g_tree_insert_internal` cannot be verified too.

Example	w/o norm.		w/ norm.		Veri.
	size	Syn.	size	Syn.	
CSLL (traverse)	8	0.23	5	0.23	0.24
CSLL of CSLLs (check)	18	0.24	4	0.23	0.22
SLL2DLL	18	0.19	2	0.2	0.18
DLL (check)	8	0.21	2	0.23	0.19
DLL (append)	11	0.2	8	0.2	0.2
CDLL (check)	23	0.22	8	0.26	0.21
CDLL of 5CSLLs	28	0.39	4	0.66	1.3
CDLL of CSLLs$_2$	29	0.33	4	0.44	0.29
tree (search)	33	0.23	2	0.24	0.23
tree-parent (traverse)	11	0.23	2	0.29	0.24
rose-tree (check)	14	0.28	14	0.3	0.23
swl (traverse)	19	0.23	13	0.27	22
mcf (check)	19	0.26	17	0.28	0.26
tll (traverse)	21	0.23	2	0.25	0.21
tll (check)	21	0.29	2	0.32	0.19
tll-parent (check)	39	0.24	2	0.35	0.24

Table 5.4: Experimental Results for Transformation Approach (cont.)

	LOC	#Proc	#Loop	#$\surd$	Syn. (second)
gslist.c	863	44	19	59	3.45
glist.c	957	29	19	43	6.41
gtree.c	1334	36	14	43	5.26
gnode.c	1131	37	25	55	9.17

Table 5.5: Experimental Results on Glib Programs

in Figure 5-11, where we are currently unable to prove memory safety. In this example, the lists `l`$_1$ and `l`$_2$ are actually from overlapping heap memory since they are computed from the same list, `list`. Our current modular procedure infers two disjoint lists as pre-condition of the loop, with `l`$_1$ being longer than `l`$_2$ through a composite predicate (similar to the `zip` example). Precondition of this loop cannot be currently proven inside `g_slist_sort_real`. To analyse this example successfully, our current tool would have to be extended to infer immutability and size property of heap data structure, so that certain heap overlaps can be handled. In particular, we would need to infer the following more precise specification:

```
struct GSList*
 g_slist_sort_real(struct GSList* list, ...) {
  struct GSList *l1, *l2;
  if (!list) return NULL;
  if(!list->next) return list;
  l₁ = list; l₂=list->next;
  while ((l₂=l₂->next) ≠ NULL){
    if ((l₂=l₂->next) == NULL) break;
    l₁=l₁->next;} //failed precondition
 l₂=l₁->next;
 ...}
```

Figure 5-11: Code of `g_tree_insert_internal` Method (Glib).

$$\texttt{requires}\ (\texttt{lseg}(\texttt{l}_2,\texttt{NULL},\texttt{n})@\texttt{L} \wedge \texttt{lseg}(\texttt{l}_1,\texttt{q},\texttt{n}-1)@\texttt{L}) \wedge \texttt{n}>0$$

$$\texttt{ensures}\ \texttt{l}_2'=\texttt{NULL} \wedge \texttt{l}_1'=\texttt{q}$$

Here, list segments of $\texttt{l}_1$ and $\texttt{l}_2$ of lengths `n` and `n−1` are overlapping but accessed in read-mode via the `@L` annotation. $\texttt{l}_1'$ and $\texttt{l}_2'$ captures the updated variables at exit of loop. Furthermore, there are other examples which rely on both shape and pure properties (e.g. sortedness or size) for memory safety. These failures motivate us to explore S2 to infer immutability and pure properties in the near future. We discuss a preliminary step in this direction next.

5.7.5 Extension to numerical properties

While outside the focus of the current proposal, we confirm that our shape synthesis system can be extended to analyse data structures with deeper numerical invariant properties, such as sorted lists. We achieved this by a small extension that generates unknown predicates for numerical fields. As a simple example, consider the singly-linked list `node` with a numerical field `val`, with code snippet presented in Figure 5-12. In the verification process, our extension provides support for the following bi-abductive entailment:

$$\texttt{H}(\texttt{x},\texttt{v:int}) \vdash \texttt{x}{\mapsto}\texttt{node}(\texttt{a},\texttt{b}) \leadsto (\texttt{H}(\texttt{x},\texttt{v}) \Rightarrow$$
$$\texttt{x}{\mapsto}\texttt{node}(\texttt{n},\texttt{q}){*}\texttt{H}_2(\texttt{q},\texttt{v}){*}\texttt{R}(\texttt{n},\texttt{v}),\ \texttt{H}_2(\texttt{b},\texttt{v}){*}\texttt{R}(\texttt{a},\texttt{v}))$$

```
bool check_sorted(struct node* x, int v)
  requires H(x,v)  ensures G(x,v)∧res;
{
  if (x==NULL) return true ;
  else
    return (v≤x->val) && check_sorted(x->next,x->val);
}
```

Figure 5-12: Code of `check_sorted` Method.

which generates $\mathtt{R}(\mathtt{v},\mathtt{n})$, an unknown predicate over two numerical parameters. By solving the generated constraints, our system can synthesize a new heap predicate over sorted linked-lists, as follows:

$$\mathtt{H}(\mathtt{x},\mathtt{v}) \equiv \mathtt{emp}{\wedge}\mathtt{x{=}NULL} \;\vee\; \mathtt{x}{\mapsto}\mathtt{node}(\mathtt{n},\mathtt{q}){*}\mathtt{H}(\mathtt{q},\mathtt{n}){\wedge}\mathtt{v}{\leq}\mathtt{n}$$

5.8 Discussions

Typically, shape analysis infers invariants and post-abstractions for pointers of programs based on predefined shape abstractions. As shape analysis follows the structure of programs, it may produce an overtly complex and less understandable results. Furthermore, due to unbounded data structures, computing fix point of shape analysis may be not terminating. Hence, shape analysis typically introduces an *abstract post operator* to obtain fix points and produce suitable results.

In separation logic, for the termination of fix point computation, shape analysis typically introduces a finite "canonical abstraction", which is a fragment of separation logic with a set of predefined shape predicates. For example, the canonical abstraction of the shape analyses presented in [9, 28, 51, 138] limits to variants of list data structure. To transform the results of shape analysis into the canonical abstraction, those analyses made use of abstract post operators, i.e. widening [51] and join [28, 138], with a set of rewrite rules. Compared to those analyses, our proposal analyzes shapes for pointers of both pre- and post-condition at the same time. Hence, our analysis needs additional step to distinguish abstractions of pre-predicates from abstractions of post-predicates. After that the analysis applies appropriate rewrite rules for each set of abstractions. For a combination of shape and numerical domain, authors in [104] propose a analysis based on symbolic execution mechanism. This analysis also introduces a set of rewrite rules for an attempt to find fix points.

The closest to our system is the shape analysis CABER presented in [16]. CABER infers a set of assumptions for both safety and termination of heap-manipulating programs. While our transformation attempts to derive a concise, understandable and usable definition for each shape predicate, the post operator of CABER only checks and removes inconsistent assumptions [18]. Thus, the results of CABER may not be immediately usable by verification systems.

Chapter 6

Conclusion

In this thesis, we have proposed new mechanisms to specify, verify and synthesize complete specification of programs using complex heap-based data structures. Complete specifications capture functional properties, for both safety and error scenarios, of programs.

Foundation of the complete specification mechanism is the error calculus. Instead of using exception mechanism, we have proposed a lattice domain with four-point status values that explicitly captures erroneous (and safe) program states. To integrate reasoning on errors into verification systems, we have introduced a calculus on these status values. We have also carefully re-designed an entailment procedure of separation logic to support reasoning on both safety and errors in the presence of data structures with sophisticated invariant, via user-defined predicates and lemmas. Furthermore, through the calculus we can provide a fairly precise and concise explanation when the verification fails to either prove safety or validate errors.

We have also developed an inference methodology that centers on a second-order bi-abduction that can synthesize arbitrary shape predicates, from scratch, needed to ensure memory safety. Second-order variables are place-holders for unknown predicates that can be synthesized from proof obligations gathered by

Hoare-style verification. The soundness of our inference is based on the soundness of the entailment procedure itself, and is not subjected to a re-verification process. Our proposal for shape analysis has been structured into two key stages:

1. Gathering of relational assumptions on unknown shape predicates.

2. Synthesis of predicate definitions via derivation and normalization steps to provide concise, understandable and usable shape definitions.

As shown in Appendix .2, our analysis has been proven sound. Compared to state-of-the-art work [13, 20, 132], our work is capable of synthesizing shape predicates for preconditions (and can directly work with recursive methods). We have discussed a preliminary extension of the second-order bi-abduction to complete specification. The key point of this extension is an abduction mechanism to disprove proof obligations. We believe that this work is particularly useful for automatically discovering effective formal specifications required by automated verification systems.

We have designed and implemented a prototype system of the proposed verification system within an existing verification infrastructure for C language and conducted experiments with medium-sized programs manipulating complex data structures. Our initial experiments suggest that our complete specification language is expressive enough to capture functional properties of programs with both safe and erroneous scenarios. Moreover, our verification system can provide concise error explanations with modest overheads. The experimental results also showed that the inference system can synthesize non-trivial shape predicates definitions that could not be inferred before. Lastly, our framework can support a mix of verification and inference through partial specifications. It will be interesting to explore how our framework could be structured to support incremental inference.

6.1 Future Works

One possible future direction is to enhance the second-order bi-abductive mechanism to infer both complete and concise specification.

To enhance completeness, our second-order bi-abduction may be extended to support *safety and errors* as well as *combined domains* inference. We have discussed an extension of the second-order bi-abduction to capture preconditions that lead to implicit (memory) error execution. It is also useful to carry out an analysis that is able to capture specifications that lead to explicit abnormal execution, such as *abort* statements (e.g. the method *zip* in Figure 5-6) or negative returns in Linux kernel API [67, 87]). Our second-order bi-abduction mechanism was formulated for shape domain and did not fully consider information from pure domain. A recent work [135] proposed a semi-automatic mechanism based on unknown pure predicates to extend shape predicates with pure properties (numerical and bag domains). To further enhance on completeness, it is interesting to extend the mechanism to a combination of shape and pure domains.

David et. al. [44] pointed out that it is more concise, precise and efficient when specification logic captures immutability information. To enhance conciseness, we would like to infer specification with immutability annotation. To do that, we need to investigate mechanisms to infer immutability information.

Another possible future work is to develop our S2 tool further into a more robust and scalable verification system, targeting on real-world software such as API of Linux kernel. As this work requires significant engineering efforts, we have left this topic to be explored in the future.

With these extensions, we hope that the new analysis can automatically verify a wide range of programs to a higher level correctness and scalability.

Bibliography

[1] VSTTE 2012 Software Verification Competition. [Online; accessed 27-July-2012]. https://sites.google.com/site/vstte2012/compet, 2012.

[2] Glib-2.38.2. https://developer.gnome.org/glib/, 2013. [Online; accessed 13-Nov-2013].

[3] Evaluation Assurance Level. [Online; accessed 01-Aug-2014]. http://cygnacom.com/labs/cc_assurance_index/ccinhtml/part3/part36.htm, 2014.

[4] V.S. Alagar and K. Periyasamy. Vienna development method. In *Specification of Software Systems*, Texts in Computer Science, pages 405–459. Springer London, 2011.

[5] M. Alpuente, M. A. Feliú, and A. Villanueva. Automatic inference of specifications using matching logic. In *PEPM*, pages 127–136, 2013.

[6] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In *Int'l Workshop on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, volume 3362, pages 49–69. Springer-Verlag, LNCS, 2004.

[7] Mike Barnett, Bor-YuhEvan Chang, Robert DeLine, Bart Jacobs, and K.RustanM. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *FMCO*, volume 4111, pages 364–387. Springer Berlin Heidelberg, 2006.

[8] Mike Barnett, Manuel Fhndrich, Peter Mller, K. Rustan M. Leino, Wolfram Schulte, and Herman Venter. Specification and verification: The spec# experience, 2009.

[9] J. Berdine, C. Calcagno, B. Cook, D. Distefano, P. W. O'Hearn, T. Wies, and H. Yang. Shape analysis for composite data structures. In *CAV*, pages 178–192, 2007.

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

[11] J. Berdine, C. Calcagno, and P. W. O'Hearn. Symbolic Execution with Separation Logic. In *APLAS*, volume 3780, pages 52–68. Springer-Verlag, November 2005.

[12] J. Berdine, C. Calcagno, and P. W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *FMCO*, Springer LNCS 4111, pages 115–137, 2006.

[13] J. Berdine, B. Cook, and S. Ishtiaq. SLAYER: memory safety for systems-level code. In *CAV*, pages 178–183, 2011.

[14] Dirk Beyer, ThomasA. Henzinger, and Grgory Thoduloz. Lazy shape analysis. In Thomas Ball and RobertB. Jones, editors, *Computer Aided Verification*, volume 4144 of *Lecture Notes in Computer Science*, pages 532–546. Springer Berlin Heidelberg, 2006.

[15] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. Design and implementation of a special-purpose static program analyzer for safety-critical real-time embedded software, invited chapter. In *The Essence of Computation: Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones*, LNCS 2566, pages 85–108. Springer-Verlag, October 2002.

[16] J. Brotherston and N. Gorogiannis. Cyclic abduction of inductively defined safety and termination preconditions. SAS, 2014.

[17] James Brotherston, Dino Distefano, and Rasmus L. Petersen. Automated cyclic entailment proofs in separation logic. In *Proceedings of CADE-23*, LNAI, pages 131–146. Springer, 2011.

[18] James Brotherston, Carsten Fuhs, Nikos Gorogiannis, and Juan Navarro Pérez. A decision procedure for satisfiability in separation logic with inductive predicates. In *Proceedings of CSL-LICS*. ACM, 2014.

[19] L. Burdy, Y. Cheon, D. R. Cok, M. D. Ernst, J. R. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of JML tools and applications. *Software Tools for Technology Transfer*, 2005.

[20] C. Calcagno, D. Distefano, P. W. O'Hearn, and H. Yang. Compositional shape analysis by means of bi-abduction. In *POPL*, pages 289–300, 2009.

[21] C. Calcagno, D. Distefano, and V. Vafeiadis. Bi-abductive resource invariant synthesis. In *APLAS*, pages 259–274, 2009.

[22] Cristiano Calcagno and Dino Distefano. Infer: An automatic program verifier for memory safety of c programs. In *NASA Formal Methods*, volume 6617, pages 459–465. Springer Berlin Heidelberg, 2011.

[23] Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. Beyond reachability: Shape abstraction in the presence of pointer arithmetic. In *SAS*, pages 182–203, 2006.

[24] Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. Footprint analysis: A shape analysis that discovers preconditions. In *SAS*, pages 402–418, 2007.

[25] Cristiano Calcagno, Hongseok Yang, and Peter W. O'Hearn. Computability and complexity results for a spatial assertion language for data structures. In *APLAS*, pages 289–300, 2001.

[26] S. Chandra, S. J. Fink, and M. Sridharan. Snugglebug: a powerful approach to weakest preconditions. In *PLDI*, pages 363–374, 2009.

[27] Satish Chandra, Stephen J. Fink, and Manu Sridharan. Snugglebug: A powerful approach to weakest preconditions. In *PLDI*, pages 363–374, New York, NY, USA, 2009. ACM.

[28] B.-Y. E. Chang and X. Rival. Relational inductive shape analysis. In *POPL*, pages 247–260, 2008.

[29] W.-N. Chin, C. David, H. H. Nguyen, and S. Qin. Enhancing modular oo verification with separation logic. In *POPL*, pages 87–99, 2008.

[30] Wei-Ngan Chin, Cristina David, Huu Hai Nguyen, and Shengchao Qin. Multiple pre/post specifications for heap-manipulating methods. In *HASE*, pages 357–364, 2007.

[31] Wei-Ngan Chin, Cristina David, Huu Hai Nguyen, and Shengchao Qin. Enhancing modular oo verification with separation logic. *SIGPLAN Not.*, 43(1):87–99, January 2008.

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

[33] W.N. Chin, C. David, H.H. Nguyen, and S. Qin. Automated verification of shape, size and bag properties via user-defined predicates in separation logic. *Sci. Comput. Program.*, 77(9):1006–1036, 2012.

[34] Cristina Cifuentes, Nathan Keynes, Lian Li, Nathan Hawes, and Manuel Valdiviezo. Transitioning parfait into a development tool. *IEEE Security and Privacy*, 10(3):16–23, May 2012.

[35] Robert Clarisó and Jordi Cortadella. The octahedron abstract domain. *Sci. Comput. Program.*, 64(1):115–139, January 2007.

[36] Ernie Cohen, Markus Dahlweid, Mark Hillebrand, Dirk Leinenbach, Micha Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. Vcc: A practical system for verifying concurrent c. In *Theorem Proving in Higher Order Logics*, volume 5674, pages 23–42. Springer Berlin Heidelberg, 2009.

[37] Michael Colón and Henny Sipma. Synthesis of linear ranking functions. In *Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS 2001, pages 67–81, London, UK, UK, 2001. Springer-Verlag.

[38] Byron Cook, Christoph Haase, Joël Ouaknine, Matthew J. Parkinson, and James Worrell. Tractable reasoning in a fragment of separation logic. In *CONCUR*, pages 235–249, 2011.

[39] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252, New York, NY, USA, 1977. ACM.

[40] Patrick Cousot, Radhia Cousot, Manuel Fhndrich, and Francesco Logozzo. Automatic inference of necessary preconditions. In *VMCAI*, volume 7737, pages 128–148. Springer Berlin Heidelberg, 2013.

[41] Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *POPL*, pages 84–96, New York, NY, USA, 1978. ACM.

[42] Christoph Csallner and Yannis Smaragdakis. Check 'n' Crash: combining static checking and testing. In *ICSE*, pages 422–431, 2005.

[43] Christoph Csallner, Yannis Smaragdakis, and Tao Xie. DSD-Crasher: A hybrid analysis tool for bug finding. *ACM Trans. Softw. Eng. Methodol.*, 17(2), 2008.

[44] Cristina David and Wei-Ngan Chin. Immutable specifications for more concise and precise verification. In *OOPSLA*, pages 359–374, New York, NY, USA, 2011. ACM.

[45] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In *TACAS*, 2008.

[46] I. Dillig, T. Dillig, and A. Aiken. Reasoning about the unknown in static analysis. *Commun. ACM*, 53(8):115–123, 2010.

[47] I. Dillig, T. Dillig, A. Aiken, and M. Sagiv. Precise and compact modular procedure summaries for heap manipulating programs. In *PLDI*, pages 567–577, 2011.

[48] Isil Dillig, Thomas Dillig, and Alex Aiken. Automated error diagnosis using abductive inference. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 181–192, New York, NY, USA, 2012. ACM.

[49] Isil Dillig, Thomas Dillig, Boyang Li, and Ken McMillan. Inductive invariant generation via abductive inference. *SIGPLAN Not.*, 48(10):443–456, October 2013.

[50] D. Distefano, P. W. O'Hearn, and H. Yang. A Local Shape Analysis based on Separation Logic. In *TACAS*, volume 3920, pages 287–302. Springer-Verlag, March 2006.

[51] D. Distefano, P. W. O'Hearn, and H. Yang. A local shape analysis based on separation logic. In *TACAS*, pages 287–302, 2006.

[52] Hyunsook Do, Sebastian Elbaum, and Gregg Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Softw. Engg.*, 10:405–435, October 2005.

[53] DO-178C. Software considerations in airborne systems and equipment certification. Technical report, Radio Technical Commission on Aviation, 2011.

[54] Julian Dolby, Mandana Vaziri, and Frank Tip. Finding bugs efficiently with a SAT solver. In *ESEC/SIGSOFT FSE*, pages 195–204, 2007.

[55] A. Dolzmann and Thomas Sturm. Redlog: computer algebra meets computer logic. *SIGSAM Bull.*, 31:2–9, June 1997.

[56] Nurit Dor, Michael Rodeh, and Mooly Sagiv. Cleanness checking of string manipulations in c programs via integer analysis. In *Static Analysis*, volume 2126, pages 194–212. Springer Berlin Heidelberg, 2001.

[57] K. Dudka, P. Peringer, and T. Vojnar. Predator: a practical tool for checking manipulation of dynamic data structures using separation logic. In *CAV*, pages 372–378, 2011.

[58] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. In *Proceedings of the 21st International Conference on Software Engineering*, ICSE '99, pages 213–224, New York, NY, USA, 1999. ACM.

[59] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *PLDI*, pages 234–245, 2002.

[60] Cristian Gherghina, Cristina David, Shengchao Qin, and Wei-Ngan Chin. Structured specifications for better verification of heap-manipulating programs. In *FM*, pages 386–401, 2011.

[61] Roberto Giacobazzi. Abductive analysis of modular logic programs, 1994.

[62] A. Gotsman, J. Berdine, and B. Cook. Interprocedural Shape Analysis with Separated Heap Abstractions. In *SAS*, pages 240–260, 2006.

[63] Philippe Granger. Static analysis of linear congruence equalities among variables of a program. In *TAPSOFT*, pages 169–192, New York, NY, USA, 1991.

[64] S. Grebenshchikov, Nuno P. Lopes, C. Popeea, and A. Rybalchenko. Synthesizing software verifiers from proof rules. In *PLDI*, pages 405–416, 2012.

[65] Andreas Griesmayer, Stefan Staber, and Roderick Bloem. Automated fault localization for C programs. *Electron. Notes Theor. Comput. Sci.*, 174:95–111, May 2007.

[66] B. S. Gulavani, S. Chakraborty, S. Ramalingam, and A. V. Nori. Bottom-up shape analysis. In *SAS*, pages 188–204, 2009.

[67] Haryadi S. Gunawi, Cindy Rubio-gonzlez, Andrea C. Arpaci-dusseau, Remzi H. Arpaci-dusseau, and Ben Liblit. Eio: Error handling is occasionally correct.

[68] B. Guo, N. Vachharajani, and D. I. August. Shape analysis with inductive recursion synthesis. In *ACM PLDI*, pages 256–265, 2007.

[69] Christoph Haase, Samin Ishtiaq, Joël Ouaknine, and Matthew J. Parkinson. Seloger: A tool for graph-based reasoning in separation logic. In *CAV*, pages 790–795, 2013.

[70] G. He, S. Qin, W.-N Chin, and F. Craciun. Automated specification discovery via user-defined predicates. In *ICFEM*, 2013.

[71] Michael Hind. Pointer analysis: Haven't we solved this problem yet? In *PASTE*, pages 54–61, New York, NY, USA, 2001. ACM.

[72] Jochen Hoenicke, K. Rustan Leino, Andreas Podelski, Martin Schäf, and Thomas Wies. It's doomed; we can prove it. In *FM 2009: Formal Methods*, FM '09, pages 338–353, 2009.

[73] L. Holik, O. Lengál, A. Rogalewicz, J. Simácek, and T. Vojnar. Fully automated shape analysis based on forest automata. *SAS*, abs/1304.5806, 2013.

[74] David Hovemeyer and William Pugh. Finding more null pointer bugs, but not too many. In *PASTE*, pages 9–14, New York, NY, USA, 2007. ACM.

[75] T. Reps: http://www.grammatech.com/codesonar. Codesonar, last access: 2014.

[76] R. Iosif, A. Rogalewicz, and J. Simácek. The tree width of separation logic with recursive definitions. In *CADE*, pages 21–38, 2013.

[77] Radu Iosif, Adam Rogalewicz, and Tomás Vojnar. Deciding entailments in inductive separation logic with tree automata. *ATVA*, 2014.

[78] S. Ishtiaq and P. W. O'Hearn. BI as an Assertion Language for Mutable Data Structures. In *ACM POPL*, London, January 2001.

[79] Daniel Jackson. Alloy: A lightweight object modelling notation. *ACM Trans. Softw. Eng. Methodol.*, 11(2):256–290, April 2002.

[80] Daniel Jackson and Mandana Vaziri. Finding bugs with a constraint solver. In *Proceedings of the 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '00, pages 14–25, 2000.

[81] Bertrand Jeannet, Alexey Loginov, Thomas Reps, and Mooly Sagiv. A relational approach to interprocedural shape analysis. *ACM Trans. Program. Lang. Syst.*, 32(2):5:1–5:52, February 2010.

[82] H. B. M. Jonkers. Upgrading the pre- and postcondition technique. In *VDM '91: Proceedings of the 4th International Symposium of VDM Europe on Formal Software Development-Volume I*, pages 428–456, 1991.

[83] Manu Jose and Rupak Majumdar. Cause clue clauses: error localization using maximum satisfiability. In *PLDI*, pages 437–446, New York, NY, USA, 2011. ACM.

[84] Michael Karr. Affine relationships among variables of a program. *Acta Informatica*, 6(2):133–151, 1976.

[85] N. Klarlund and A. Moller. MONA Version 1.4 - User Manual. BRICS Notes Series, 2001.

[86] N. Klarlund and M. I. Schwartzbach. Graph Types. In *ACM POPL*, Charleston, South Carolina, January 1993.

[87] J. Lawall, B. Laurie, R.R. Hansen, N. Palix, and G. Muller. Finding error handling bugs in openssl using coccinelle. In *EDCC, 2010 European*, pages 191–196, 2010.

[88] Duy-Khanh Le, Wei-Ngan Chin, and Yong-Meng Teo. An expressive framework for verifying deadlock freedom. In *ATVA*, volume 8172, pages 287–302. Springer International Publishing, 2013.

[89] Q.L. Le, C. Gherghina, S. Qin, and W.N. Chin. Shape analysis via second-order bi-abduction. In *Technical Report*, Soc, NUS, February 2014. http://loris-7.ddns.comp.nus.edu.sg/∼project/s2/beta/src/TRs2.pdf.

[90] Quang Loc Le, Cristian Gherghina, Shengchao Qin, and Wei-Ngan Chin. Shape analysis via second-order bi-abduction. In *CAV*, pages 52–68, 2014.

[91] Quang Loc Le, Asankhaya Sharma, Florin Craciun, and Wei-Ngan Chin. Towards complete specifications with an error calculus. In *NASA Formal Methods*, pages 291–306, 2013.

[92] Ton Chanh Le, Cristian Gherghina, Razvan Voicu, and Wei-Ngan Chin. A proof slicing framework for program verification. In *ICFEM*, pages 53–69, 2013.

[93] Gary T. Leavens and Albert L. Baker. Enhancing the pre- and postcondition technique for more expressive specifications. In *FM99*, pages 1087–1106. Springer, 1999.

[94] O. Lee, H. Yang, and R. Petersen. Program analysis for overlaid data structures. In *CAV*, 2011.

[95] O. Lee, H. Yang, and K. Yi. Automatic verification of pointer programs using grammar-based shape analysis. In *ESOP*, pages 124–140, 2005.

[96] K. Rustan M. Leino. Specification and verification of object-oriented software, 2008. Marktoberdorf International Summer School 2008, lecture notes.

[97] K. Rustan M. Leino. Automating induction with an smt solver. In *VMCAI*, pages 315–331, Berlin, Heidelberg, 2012.

[98] K. Rustan M. Leino and Michal Moskal. Co-induction simply - automatic co-inductive proofs in a program verifier. In *FM*, pages 382–398, 2014.

[99] K.RustanM. Leino. Dafny: An automatic program verifier for functional correctness. In *Logic for Programming, Artificial Intelligence, and Reasoning*, volume 6355, pages 348–370. Springer-Verlag, 2010.

[100] T. Lev-Ami, M. Sagiv, T. Reps, and S. Gulwani. Backward analysis for inferring quantified preconditions. Technical Report TR-2007-12-01, Tel Aviv University, 2007.

[101] Tobias Lindahl and Konstantinos Sagonas. Practical type inference based on success typings. In *PPDP*, pages 167–178, 2006.

[102] Alexey Loginov, Thomas Reps, and Mooly Sagiv. Abstraction refinement via inductive learning. In *CAV*, volume 3576, pages 519–533. Springer-Verlag, 2005.

[103] Parthasarathy Madhusudan, Xiaokang Qiu, and Andrei Stefanescu. Recursive proofs for inductive tree data-structures. In *POPL*, pages 123–136, New York, NY, USA, 2012. ACM.

[104] S. Magill. Inferring invariants in separation logic for imperative list-processing programs. In *3rd SPACE Workshop*, 2006.

[105] S. Magill, J. Berdine, E. M. Clarke, and B. Cook. Arithmetic strengthening for shape analysis. In *SAS*, pages 419–436, 2007.

[106] S. Magill, M.-H. Tsai, P. Lee, and Y.-K. Tsay. Automatic numeric abstractions for heap-manipulating programs. In *POPL*, pages 211–222, 2010.

[107] Bill McCloskey, Thomas Reps, and Mooly Sagiv. Statically inferring complex heap, array, and numeric invariants. In *SAS*, pages 71–99, Berlin, Heidelberg, 2010. Springer-Verlag.

[108] B. Meyer. Eiffel: The language (prentice hall object-oriented series), 1991.

[109] Antoine Miné. The octagon abstract domain. *Higher Order Symbol. Comput.*, 19(1):31–100, March 2006.

[110] A. Moeller and M. I. Schwartzbach. The Pointer Assertion Logic Engine. In *ACM PLDI*, pages 221–231, June 2001.

[111] JuanAntonio Navarro Prez and Andrey Rybalchenko. Separation logic modulo theories. In Chung-chieh Shan, editor, *APLAS*, volume 8301, pages 90–106. Springer International Publishing, 2013.

[112] G. Necula, S. McPeak, S. Rahul, and W. Weimer. CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In *CC'02*, pages 213–228, 2002.

[113] H. H. Nguyen and W.-N. Chin. Enhancing program verification with lemmas. In *CAV*, pages 355–369, 2008.

[114] H.H. Nguyen, C. David, S.C. Qin, and W.N. Chin. Automated Verification of Shape And Size Properties via Separation Logic. In *VMCAI*, pages 251–266, January 2007.

[115] P. W. O'Hearn, J. Reynolds, and H. Yang. Local Reasoning about Programs that Alter Data Structures. In *Proceedings of the 15th Annual Conference of the European Association for Computer Science Logic*, Paris, France, September 2001.

[116] Peter O'Hearn. Tutorial on separation logic (invited tutorial). In *CAV*, 2008.

[117] DavidLorge Parnas. Precise documentation: The key to better software. In Sebastian Nanz, editor, *The Future of Software Engineering*, pages 125–148. Springer Berlin Heidelberg, 2011.

[118] Juan Antonio Navarro Pérez and Andrey Rybalchenko. Separation logic + superposition calculus = heap theorem prover. In *PLDI*, pages 556–566, 2011.

[119] Juan Antonio Navarro Pérez and Andrey Rybalchenko. Separation logic modulo theories. In *APLAS*, pages 90–106, 2013.

[120] R. Piskac, T. Wies, and D. Zufferey. Automating separation logic using smt. In Natasha Sharygina and Helmut Veith, editors, *CAV*, volume 8044, pages 773–789. Springer-Verlag, 2013.

[121] Ruzica Piskac, Thomas Wies, and Damien Zufferey. Automating separation logic with trees and data. In *CAV*, volume 8559, pages 711–728. Springer International Publishing, 2014.

[122] Ruzica Piskac, Thomas Wies, and Damien Zufferey. Grasshopper: Complete heap verification with mixed specifications. *TACAS'14*, 2014.

[123] Nadia Polikarpova, Carlo A. Furia, Yu Pei, Yi Wei, and Bertrand Meyer. What good are strong specifications? In *ICSE*, pages 262–271, Piscataway, NJ, USA, 2013. IEEE Press.

[124] Corneliu Popeea and Wei-Ngan Chin. Inferring disjunctive postconditions. In *ASIAN*, pages 331–345, 2006.

[125] W. Pugh. The Omega Test: A fast practical integer programming algorithm for dependence analysis. *Communications of the ACM*, 8:102–114, 1992.

[126] X. Qiu, P. Garg, A. Ştefănescu, and P. Madhusudan. Natural proofs for structure, data, and separation. In *PLDI*, pages 231–242, New York, NY, USA, 2013. ACM.

[127] J. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *IEEE LICS*, pages 55–74, 2002.

[128] John C. Reynolds. Automatic computation of data set definitions. In *IFIP Congress (1)*, pages 456–461, 1968.

[129] N. Rinetzky, M. Sagiv, and E. Yahav. Interprocedural shape analysis for cutpoint-free programs. In *SAS*, pages 284–302, 2005.

[130] X. Rival and B.-Y. E. Chang. Calling context abstraction with shapes. In *POPL*, pages 173–186, 2011.

[131] G. Rosu and A. Stefanescu. Checking reachability using matching logic. In *OOPSLA*, pages 555–574. ACM, 2012.

[132] S. Sagiv, T. Reps, and R. Wilhelm. Parametric Shape Analysis via 3-valued Logic. In *ACM POPL*, January 1999.

[133] S. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM TOPLAS*, 24(3), May 2002.

[134] Makoto Tatsuta and Wei-Ngan Chin. Completeness of separation logic with inductive definitions for program verification. In *SEFM*, volume 8702, pages 20–34. Springer International Publishing, 2014.

[135] Minh-Thai Trinh, Quang Loc Le, Cristina David, and Wei-Ngan Chin. Bi-abduction with pure properties for specification inference. In *APLAS*, pages 107–123. Springer International Publishing, 2013.

[136] Mandana Vaziri and Daniel Jackson. Checking properties of heap-manipulating procedures with a constraint solver. In *TACAS*, pages 505–520, 2003.

[137] Daniel von Dincklage and Amer Diwan. Explaining failures of program analyses. In *PLDI*, pages 260–269, 2008.

[138] H. Yang, O. Lee, J. Berdine, C. Calcagno, B. Cook, D. Distefano, and P. W. O'Hearn. Scalable shape analysis for systems code. In *CAV*, pages 385–398, 2008.

Appendices

.1 Proof of the Soundness of the Structural Rules for $\vdash_p$

We prove Theorem 1 inductively on the structural rules through $\sqcup$ operator, $\otimes$ operator and $\oplus$ operator.

.1.1 JOIN ($\sqcup$) Operator

$$\frac{\begin{array}{c}\left[\underline{\textbf{EC}-[\sqcup\ \textbf{JOIN}]}\right]\\ \pi_1 \vdash_p \pi \rightsquigarrow \tau_1 \\ \pi_2 \vdash_p \pi \rightsquigarrow \tau_2\end{array}}{\pi_1 \vee \pi_2 \vdash_p \pi \rightsquigarrow \tau \ \textit{and}\ \tau_1 \sqcup \tau_2 = \tau}$$

We prove Theorem 1 by the case analysis on the returned τ.

Case $\tau = \bot$.

Based on the lattice of program status, $\tau_1 \sqcup \tau_2 = \bot$ if $\tau_1 = \bot$ and $\tau_2 = \bot$. It means $\tau_1 \sqcup \tau_2 = \bot$ if $\pi_1 \vdash_p \pi \rightsquigarrow \bot$ and $\pi_2 \vdash_p \pi \rightsquigarrow \bot$.

Follow the entailment procedure $\vdash_p$, we have $\pi_i \vdash_p \pi \rightsquigarrow \bot$ infers that $\textbf{unsat}(\pi_\mathbf{i})$ with $i \in \{1, 2\}$.

We have:

$$\begin{aligned} & \textbf{unsat}(\pi_\mathbf{1}) \wedge \textbf{unsat}(\pi_\mathbf{2}) \\ \equiv\ & \neg\pi_1 \wedge \neg\pi_2 \\ \equiv\ & \neg(\pi_1 \vee \pi_2) \\ \equiv\ & \textbf{unsat}(\pi_\mathbf{1} \vee \pi_\mathbf{2}) \end{aligned}$$

Again, follow the entailment procedure $\vdash_p$ we conclude: $\pi_1 \vee \pi_2 \vdash_p \pi \rightsquigarrow \bot$

Therefore, $\tau_1 \sqcup \tau_2 = \tau$.

Case $\tau = \surd$.

Based on the lattice of program status, $\tau_1 \sqcup \tau_2 = \surd$ if

1. $\tau_1 = \surd$ and $\tau_2 = \surd$. Or

2. One of them is $\perp$ and another is $\surd$. We assume $\tau_1 \ = \ \perp$ and $\tau_2 \ = \ \surd$

Case $\tau_1 = \surd$ and $\tau_2 = \surd$

$\tau_1 = \surd$, it means $\pi_1 \vdash_p \pi \rightsquigarrow \surd$. Follow the entailment procedure $\vdash_p$, we have:

$$\textbf{unsat}(\pi_{\mathbf{1}} \wedge \neg\pi) \qquad (1.a.1)$$

Similarly, with $\tau_2 = \surd$, we have:

$$\textbf{unsat}(\pi_{\mathbf{2}} \wedge \neg\pi) \qquad (1.a.2)$$

From (1.a.1) and (1.a.2), we have:

$$\begin{aligned}
& \neg\ (\pi_1 \ \wedge \ \neg\pi) \ \wedge \ \neg\ (\pi_2 \ \wedge \ \neg\pi) \\
\equiv \ & (\neg\pi_1 \ \vee \ \pi) \ \wedge \ (\neg\pi_2 \ \vee \ \pi) \\
\equiv \ & \equiv (\neg\pi_1 \ \wedge \ \neg\pi_2) \ \vee \ \pi \\
\equiv \ & \neg(\pi_1 \ \vee \ \pi_2) \ \vee \ \pi \\
\equiv \ & \neg((\pi_1 \ \vee \ \pi_2) \ \wedge \ \neg\pi) \\
\equiv \ & \textbf{unsat}(\pi_{\mathbf{1}} \vee \pi_{\mathbf{2}}) \ \wedge \neg\ \pi) \qquad (1.1)
\end{aligned}$$

From (1.1), and follow the entailment procedure $\vdash_p$ we conclude: $\pi_1 \vee \pi_2 \vdash_p \pi \rightsquigarrow \surd$.

So, $\tau_1 \sqcup \tau_2 = \tau$.

Case $\tau_1 \ = \ \perp$ and $\tau_2 \ = \ \surd$

It means $\pi_1 \vdash_p \pi \rightsquigarrow \perp$ and $\pi_2 \vdash_p \pi \rightsquigarrow \surd$.

Follow the entailment procedure $\vdash_p$, we have:

$$\mathbf{unsat}(\pi_{\mathbf{1}})\wedge \qquad (1.b.1)$$

$$\mathbf{unsat}(\pi_{\mathbf{2}} \wedge \neg\pi) \qquad (1.b.2)$$

From $\mathbf{unsat}(\pi_{\mathbf{1}})$, we have $\pi_1 \implies \pi$.

Moreover, with $\pi_1 \implies \pi$ and $\pi_2 \implies \pi$, follow the same proof leading to (1.2) of **case** $\tau_1 = \surd$ **and** $\tau_2 = \surd$ we have:

$$(\pi_1 \vee \pi_2) \implies \pi \qquad (1.3)$$

From (1.3), and follow the entailment procedure $\vdash_p$ we conclude: $\pi_1 \vee \pi_2 \vdash_p \pi \rightsquigarrow \surd$.

Therefore, $\tau_1 \sqcup \tau_2 = \tau$.

Case $\tau = \mho$**.**

Based on the lattice of program status, $\tau_1 \sqcup \tau_2 = \mho$ if

1. $\tau_1 = \mho$ and $\tau_2 = \mho$. Or

2. One of them is $\perp$ and another is $\mho$. We assume $\tau_1 = \perp$ and $\tau_2 = \mho$

Case $\tau_1 = \mho$ **and** $\tau_2 = \mho$

$\tau_1 = \mho$, it means $\pi_1 \vdash_p \pi \rightsquigarrow \mho$. Follow the entailment procedure $\vdash_p$, we have:

$$\mathbf{unsat}(\pi_{\mathbf{1}} \wedge \pi) \quad (1.c.1)$$

Similarly, with $\tau_2 = \mho$, we have:

$$\mathbf{unsat}(\pi_{\mathbf{2}} \wedge \pi) \quad (1.c.2)$$

From (1.c.1) and (1.c.2), we have:

$$\begin{aligned}
& \mathbf{unsat}(\pi_{\mathbf{1}} \wedge \pi) \wedge \mathbf{unsat}(\pi_{\mathbf{2}} \wedge \pi) \\
\equiv\ & \neg(\pi_1 \wedge \pi) \wedge \neg(\pi_2 \wedge \pi) \\
\equiv\ & (\neg\pi_1 \vee \neg\pi) \wedge (\neg\pi_2 \vee \neg\pi) \\
\equiv\ & (\neg\pi_1 \wedge \neg\pi_2) \vee \neg\pi \\
\equiv\ & \neg(\pi_1 \vee \pi_2) \vee \neg\pi \\
\equiv\ & \neg((\pi_1 \vee \pi_2) \wedge \pi) \\
\equiv\ & \mathbf{unsat}((\pi_{\mathbf{1}} \vee \pi_{\mathbf{2}}) \wedge \pi) \qquad (1.4)
\end{aligned}$$

From (1.4), and follow the entailment procedure $\vdash_p$ we conclude: $\pi_1 \vee \pi_2 \vdash_p \pi \rightsquigarrow \mho$.

So, $\tau_1 \sqcup \tau_2 = \tau$.

Case $\tau_1 = \bot$ **and** $\tau_2 = \mho$

$\tau_1 = \bot$, it means $\pi_1 \vdash_p \pi \rightsquigarrow \bot$. Follow the entailment procedure $\vdash_p$, we have:

$$\mathbf{unsat}(\pi_{\mathbf{1}}) \qquad (1.d.1)$$

$\tau_2 = \mho$, it means $\pi_2 \vdash_p \pi \rightsquigarrow \mho$. Follow the entailment procedure $\vdash_p$, we have:

$$\mathbf{unsat}(\pi_{\mathbf{2}} \wedge \pi) \quad (1.d.2)$$

From (1.d.1) we have:

$$\begin{aligned}
& \mathbf{unsat}(\pi_{\mathbf{1}}) \\
\equiv\ & \neg\pi_1 \\
\Rightarrow\ & \neg\pi_1 \vee \neg\pi \qquad (1.d.3)
\end{aligned}$$

From (1.d.2) we have:

$$\begin{aligned}
& \mathbf{unsat}(\pi_{\mathbf{2}} \wedge \pi) \\
\equiv\ & \neg(\pi_2 \wedge \pi) \\
\equiv\ & \neg\pi_2 \vee \neg\pi \qquad (1.d.4)
\end{aligned}$$

From (1.d.3) and (1.d.4) we have:

$$\begin{aligned} & (\neg\pi_1 \vee \neg\pi) \wedge (\neg\pi_2 \vee \neg\pi) \\ \equiv\ & (\neg\pi_1 \wedge \neg\pi_2) \vee \neg\pi \\ \equiv\ & \neg(\pi_1 \vee \pi_2) \vee \neg\pi \\ \equiv\ & \neg((\pi_1 \vee \pi_2) \wedge \pi) \\ \equiv\ & \mathbf{unsat}((\boldsymbol{\pi_1} \vee \boldsymbol{\pi_2}) \wedge \pi) \qquad (1.5) \end{aligned}$$

From (1.5), and follow the entailment procedure $\vdash_p$ we conclude: $\pi_1 \vee \pi_2 \vdash_p \pi \rightsquigarrow \mho$.

Therefore, $\tau_1 \sqcup \tau_2 = \tau$.

Case $\tau = \top$.

Based on the lattice of program status, $\tau_1 \sqcup \tau_2 = \top$ if

1. Either τ_1 or τ_2 is $\top$. Assume $\tau_1 = \top$. Or
2. $\tau_1 = \mho$ and $\tau_2 = \surd$.

Case $\tau_1 = \top$

$\tau_1 = \top$, it means $\pi_1 \vdash_p \pi \rightsquigarrow \top$.

Follow the entailment procedure $\vdash_p$, we have:

$$\begin{aligned} & \mathbf{sat}(\boldsymbol{\pi_1} \wedge \neg\pi) \wedge \qquad (1.e.1) \\ & \mathbf{sat}(\boldsymbol{\pi_1} \wedge \pi) \qquad (1.e.2) \end{aligned}$$

From (1.e.1) we have:

$$\begin{aligned} & \mathbf{sat}(\boldsymbol{\pi_1} \wedge \neg\pi) \\ \Rightarrow\ & \mathbf{sat}((\boldsymbol{\pi_1} \wedge \neg\pi) \vee (\boldsymbol{\pi_2} \wedge \neg\pi)) \\ \equiv\ & \mathbf{sat}((\boldsymbol{\pi_1} \vee \boldsymbol{\pi_2}) \wedge \neg\pi) \qquad (1.6) \end{aligned}$$

From (1.e.2) we have:

$$\begin{aligned} & \textbf{sat}(\pi_{\mathbf{1}} \wedge \pi) \\ \Rightarrow\ & \textbf{sat}((\pi_{\mathbf{1}} \wedge \pi) \vee (\pi_{\mathbf{2}} \wedge \pi)) \\ \equiv\ & \textbf{sat}((\pi_{\mathbf{1}} \vee \pi_{\mathbf{2}}) \wedge \pi) \qquad (1.7) \end{aligned}$$

From (1.6), (1.7) and follow the entailment procedure $\vdash_p$ we conclude: $\pi_1 \vee \pi_2 \vdash_p \pi \rightsquigarrow \top$.

Therefore, $\tau_1 \sqcup \tau_2 = \tau$.

Case $\tau_1 = \mho$ and $\tau_2 = \surd$

$\tau_1 = \mho$, it means $\pi_1 \vdash_p \pi \rightsquigarrow \mho$. Follow the entailment procedure $\vdash_p$, we have:

$$\textbf{unsat}(\pi_{\mathbf{1}} \wedge \pi) \qquad (1.f.1)$$

$\tau_2 = \surd$, it means $\pi_2 \vdash_p \pi \rightsquigarrow \surd$. Follow the entailment procedure $\vdash_p$, we have:

$$\textbf{unsat}(\pi_{\mathbf{2}} \wedge \neg\pi) \qquad (1.f.2)$$

We prove $\textbf{sat}(\pi_{\mathbf{2}} \wedge \pi)$ by contradiction. Assume that $\textbf{unsat}(\pi_{\mathbf{2}} \wedge \pi)$.

$$\begin{aligned} & \textbf{unsat}(\pi_{\mathbf{2}} \wedge \pi) \\ \equiv\ & \neg(\pi_2 \wedge \pi) \\ \equiv\ & \neg\pi_2 \vee \neg\pi \end{aligned}$$

Combined with (1.f.2), we have:

$$\begin{aligned} & (\neg\pi_2 \vee \neg\pi) \wedge (\neg\pi_2 \vee \pi) \\ \equiv\ & \neg\pi_2 \wedge (\neg\pi \vee \pi) \\ \equiv\ & \neg\pi_2 \quad \text{contradict with } (1.f.2) \end{aligned}$$

Hence, we conclude $\textbf{sat}(\pi_{\mathbf{2}} \wedge \pi)$.

$$\begin{aligned} & \textbf{sat}(\pi_{\mathbf{2}} \wedge \pi) \\ \Rightarrow\ & \textbf{sat}((\pi_{\mathbf{1}} \wedge \pi) \vee (\pi_{\mathbf{2}} \wedge \pi)) \\ \equiv\ & \textbf{sat}((\pi_{\mathbf{1}} \vee \pi_{\mathbf{2}}) \wedge \pi) \qquad (1.8) \end{aligned}$$

Similarly, we can prove that

$$\mathbf{sat}((\pi_{\mathbf{1}} \vee \pi_{\mathbf{2}}) \wedge \neg\pi) \quad (1.9)$$

From (1.8), (1.9) and follow the entailment procedure $\vdash_p$ we conclude: $\pi_1 \vee \pi_2 \vdash_p \pi \rightsquigarrow \top$.

Therefore, $\tau_1 \sqcup \tau_2 = \tau$.

.1.2 COMPOSE (⊗) Operator

$$\frac{\begin{array}{l}[\underline{\mathbf{EC}\text{–}[\otimes\ \mathbf{COMPOSE}]}] \\ \pi \vdash_p \pi_1 \rightsquigarrow \tau_1 \\ \pi \vdash_p \pi_2 \rightsquigarrow \tau_2\end{array}}{\pi \vdash_p \pi_1 \wedge \pi_2 \rightsquigarrow \tau \textit{ and } \tau_1 \otimes \tau_2 = \tau}$$

We prove Theorem 1 by the case analysis on the returned τ.

Case $\tau = \bot$.

Based on $\otimes$ operator, the result of $\tau_1 \otimes \tau_2$ is $\bot$ if either τ_1 or τ_2 is $\bot$. Assume $\tau_1 = \bot$. It means $\pi \vdash_p \pi_1 \rightsquigarrow \bot$.

Follow the entailment procedure $\vdash_p$, we infer: $\mathbf{unsat}(\pi_{\mathbf{1}})$.

Again, follow the entailment procedure $\vdash_p$ we conclude $\pi \vdash_p \pi_1 \wedge \pi_2 \rightsquigarrow \bot$

So, $\tau_1 \otimes \tau_2 = \tau$.

Case $\tau = \surd$.

Based on $\otimes$ operator, the result of $\tau_1 \otimes \tau_2$ is $\surd$ if both τ_1 and τ_2 are $\surd$. It means $\pi \vdash_p \pi_1 \rightsquigarrow \surd$ and $\pi \vdash_p \pi_2 \rightsquigarrow \surd$.

Follow the entailment procedure $\vdash_p$, we have:

$$\mathbf{unsat}(\pi \wedge \neg\pi_{\mathbf{1}}) \wedge \qquad (2.a.1)$$

$$\mathbf{unsat}(\pi \wedge \neg\pi_{\mathbf{2}}) \qquad (2.a.2)$$

From (2.a.1) and (2.a.2), we have:

$$\begin{aligned} & \mathbf{unsat}(\pi \wedge \neg\boldsymbol{\pi_1}) \wedge \mathbf{unsat}(\pi \wedge \neg\boldsymbol{\pi_2}) \\ \Rightarrow\ & (\neg\pi \vee \pi_1) \wedge (\neg\pi \vee \pi_2) \\ \equiv\ & \neg\pi \vee (\pi_1 \wedge \pi_2) \\ \equiv\ & \mathbf{unsat}(\pi \wedge \neg(\boldsymbol{\pi_1} \wedge \boldsymbol{\pi_2})) \qquad (2.1) \end{aligned}$$

From (2.1), and follow the entailment procedure $\vdash_p$ we conclude: $\pi \vdash_p \pi_1 \wedge \pi_2 \rightsquigarrow \surd$

Therefore, $\tau_1 \otimes \tau_2 = \tau$.

Case $\tau = \mho$.

Based on $\otimes$ operator, the result of $\tau_1 \otimes \tau_2$ is $\mho$ if one of them (τ_1, τ_2) is $\mho$, and another is not $\bot$. Assumme $\tau_1 = \mho$ and $\tau_2 \neq \bot$.

$\tau_1 = \mho$ means $\pi \vdash_p \pi_1 \rightsquigarrow \mho$. Follow the entailment procedure $\vdash_p$, we have:

$$\mathbf{unsat}(\pi \wedge \boldsymbol{\pi_1}) \qquad (2.b.1)$$

From (2.b.1), we have:

$$\begin{aligned} & \mathbf{unsat}(\pi \wedge \boldsymbol{\pi_1}) \\ \Rightarrow\ & \mathbf{unsat}(\pi \wedge \boldsymbol{\pi_1} \wedge \boldsymbol{\pi_2}) \qquad (2.2) \end{aligned}$$

From (2.2) and follow the entailment procedure $\vdash_p$ we conclude: $\pi \vdash_p \pi_1 \wedge \pi_2 \rightsquigarrow \mho$

Therefore, $\tau_1 \otimes \tau_2 = \tau$.

Case $\tau = \top$.

Based on $\otimes$ operator, $\tau_1 \otimes \tau_2 = \top$ if

1. τ_1=$\top$ and $\tau_2 = \top$. Or

2. One of them (τ_1, τ_2) is $\top$, another is $\surd$. Assume $\tau_1{=}\top$ and $\tau_2 = \surd$.

Case $\tau_1{=}\top$ and $\tau_2 = \top$

$\tau_1{=}\top$ means $\pi \vdash_p \pi_1 \rightsquigarrow \top$. Follow the entailment procedure $\vdash_p$, we have:

$$\mathbf{sat}(\pi \wedge \pi_{\mathbf{1}}) \qquad (2.c.1)$$

Similary, with $\tau_2{=}\top$, we have:

$$\mathbf{sat}(\pi \wedge \pi_{\mathbf{2}}) \qquad (2.c.2)$$

We prove $\mathbf{sat}(\pi \wedge \pi_{\mathbf{1}} \wedge \pi_{\mathbf{2}})$ by contradiction. Assume $\neg(\pi \wedge \pi_1 \wedge \pi_2)$.

$$\begin{aligned} & \neg(\pi \wedge \pi_1 \wedge \pi_2) \\ \equiv\ & \neg((\pi \wedge \pi_1) \wedge (\pi \wedge \pi_2)) \\ \equiv\ & \neg(\pi \wedge \pi_1) \vee \neg(\pi \wedge \pi_2) \quad (2.c.3) \end{aligned}$$

(2.c.3) contradicts with both (2.c.1) and (2.c.2). Hence, we conclude:

$$\mathbf{sat}(\pi \wedge \pi_{\mathbf{1}} \wedge \pi_{\mathbf{2}}) \qquad (2.3)$$

From (2.3), and follow the entailment procedure $\vdash_p$ we conclude: $\pi \vdash_p \pi_1 \wedge \pi_2 \rightsquigarrow \top$

Therefore, $\tau_1 \otimes \tau_2 = \tau$.

Case $\tau_1 = \top$ and $\tau_2 = \surd$

$\tau_1{=}\top$ means $\pi \vdash_p \pi_1 \rightsquigarrow \top$. Follow the entailment procedure $\vdash_p$, we have:

$$\mathbf{sat}(\pi \wedge \pi_{\mathbf{1}}) \qquad (2.d.1)$$

$\tau_2{=}\surd$ means $\pi \vdash_p \pi_2 \rightsquigarrow \surd$.

$$\mathbf{unsat}(\pi \wedge \neg\pi_{\mathbf{2}}) \qquad (2.d.2)$$

We prove $\textbf{sat}(\pi \wedge \pi_{\mathbf{1}} \wedge \pi_{\mathbf{2}})$ by contradiction. Assume $\neg(\pi \wedge \pi_1 \wedge \pi_2)$.

$$\begin{aligned} & \neg(\pi \wedge \pi_1 \wedge \pi_2) \\ \equiv \;& \neg(\pi \wedge \pi_1) \vee \neg\pi_2) \end{aligned}$$

Combined with (2.d.2), we have:

$$\begin{aligned} & \neg(\pi \wedge \pi_1) \vee \neg\pi_2) \wedge (\neg\pi \vee \pi_2) \\ \Rightarrow \;& \neg\pi \end{aligned}$$

This contradicts with (2.d.1).

Hence, we conclude:

$$\textbf{sat}(\pi \wedge \pi_{\mathbf{1}} \wedge \pi_{\mathbf{2}}) \qquad (2.4)$$

From (2.4), and follow the entailment procedure $\vdash_p$ we conclude: $\pi \vdash_p \pi_1 \wedge \pi_2 \rightsquigarrow \top$

Therefore, $\tau_1 \otimes \tau_2 = \tau$.

.1.3 UNION ($\oplus$) Operator

$$\frac{\underline{[\textbf{EC}-[\oplus\ \textbf{UNION}]]} \quad \pi \vdash_p \pi_1 \rightsquigarrow \tau_1 \quad \pi \vdash_p \pi_2 \rightsquigarrow \tau_2}{\pi \vdash_p \pi_1 \vee \pi_2 \rightsquigarrow \tau \ \textit{and}\ \tau_1 \oplus \tau_2 = \tau}$$

We prove Theorem 1 by the case analysis on the returned τ.

Case $\tau = \bot$.

Based on $\oplus$ operator, the result of $\tau_1 \oplus \tau_2$ is $\bot$ if either τ_1 or τ_2 is $\bot$. Assume $\tau_1 = \bot$. It means $\pi \vdash_p \pi_1 \rightsquigarrow \bot$.

Follow the entailment procedure $\vdash_p$, we infer: $\textbf{unsat}(\pi_{\mathbf{1}})$.

Again, follow the entailment procedure $\vdash_p$ we conclude $\pi \vdash_p \pi_1 \vee \pi_2 \rightsquigarrow \bot$.

Therefore, $\tau_1 \oplus \tau_2 = \tau$.

Case $\tau = \surd$.

Based on $\oplus$ operator, the result of $\tau_1 \oplus \tau_2$ is $\surd$ if either (τ_1 or τ_2) is $\surd$. Assume $\tau_1 = \surd$. It means $\pi \vdash_p \pi_1 \rightsquigarrow \surd$.

Follow the entailment procedure $\vdash_p$, we have:

$$\textbf{unsat}(\pi \wedge \neg\boldsymbol{\pi_1}) \qquad (3.a.1)$$

From (3.a.1), we have:

$$\begin{aligned} & \textbf{unsat}(\pi \wedge \neg\boldsymbol{\pi_1}) \\ \Rightarrow\ & (\neg\pi \vee \pi_1) \vee \pi_2 \\ \equiv\ & \neg\pi \vee (\pi_1 \vee \pi_2) \\ \equiv\ & \textbf{unsat}(\pi \wedge \neg(\boldsymbol{\pi_1} \vee \boldsymbol{\pi_2})) \quad (3.1) \end{aligned}$$

From (3.1), and follow the entailment procedure $\vdash_p$ we conclude $\pi \vdash_p \pi_1 \vee \pi_2 \rightsquigarrow \surd$.

Therefore, $\tau_1 \oplus \tau_2 = \tau$.

Case $\tau = \mho$.

Based on $\oplus$ operator, the result of $\tau_1 \oplus \tau_2$ is $\mho$ if both τ_1 and τ_2 are $\mho$.

$\tau_1 = \mho$ means $\pi \vdash_p \pi_1 \rightsquigarrow \mho$.

Follow the entailment procedure $\vdash_p$, we have:

$$\textbf{unsat}(\pi \wedge \boldsymbol{\pi_1}) \qquad (3.b.1)$$

Similarly, with $\tau_1 = \mho$ we have:

$$\textbf{unsat}(\pi \wedge \boldsymbol{\pi_2}) \qquad (3.b.2)$$

From (3.b.1) and (3.b.2), we have:

$$\begin{aligned} & \mathbf{unsat}(\pi \wedge \pi_{\mathbf{1}}) \wedge \mathbf{unsat}(\pi \wedge \pi_{\mathbf{2}}) \\ \Rightarrow\ & (\neg\pi \vee \neg\pi_1) \wedge (\neg\pi \vee \neg\pi_2) \\ \equiv\ & \neg\pi \vee (\neg\pi_1 \wedge \neg\pi_2) \\ \equiv\ & \neg\pi \vee \neg(\pi_1 \vee \pi_2) \\ \equiv\ & \neg(\pi \wedge (\pi_1 \vee \pi_2)) \\ \equiv\ & \mathbf{unsat}(\pi \wedge (\pi_{\mathbf{1}} \vee \pi_{\mathbf{2}})) \qquad (3.2) \end{aligned}$$

From (3.2) and follow the entailment procedure $\vdash_p$ we conclude $\pi \vdash_p \pi_1 \vee \pi_2 \rightsquigarrow \mho$. Therefore, $\tau_1 \oplus \tau_2 = \tau$.

Case $\tau = \top$.

Based on $\oplus$ operator, the result of $\tau_1 \oplus \tau_2$ is $\mho$ if one of them (τ_1, τ_2) is $\top$, and another is neither $\bot$ nor $\surd$. We assume $\tau_1 = \top$ and τ_2 is neither $\bot$ nor $\surd$.

$\tau_1 = \top$ means $\pi \vdash_p \pi_1 \rightsquigarrow \top$.

Follow the entailment procedure $\vdash_p$, we have:

$$\mathbf{sat}(\pi \wedge \pi_{\mathbf{1}}) \qquad (3.c.1)$$

τ_2 is neither $\bot$ nor $\surd$, then $\pi \vdash_p \pi_1 \rightsquigarrow t$ and $t \neq \bot \wedge t \neq \surd$. Follow the entailment procedure $\vdash_p$, we have:

$$\mathbf{sat}(\pi \wedge \neg\pi_{\mathbf{2}}) \qquad (3.c.2)$$

From (3.c.1), we have:

$$\begin{aligned} & \mathbf{sat}(\pi \wedge \pi_{\mathbf{1}}) \\ \Rightarrow\ & \mathbf{sat}((\pi \wedge \pi_{\mathbf{1}}) \vee (\pi \wedge \pi_{\mathbf{2}})) \\ \equiv\ & \mathbf{sat}((\pi \wedge (\pi_{\mathbf{1}} \vee \pi_{\mathbf{2}})) \qquad (3.3) \end{aligned}$$

From (3.3) and follow the entailment procedure $\vdash_p$ we conclude $\pi \vdash_p \pi_1 \vee \pi_2 \rightsquigarrow \top$. Therefore, $\tau_1 \oplus \tau_2 = \tau$.

.2 Expanded Soundness of Shape Synthesis

.2.1 Proof for Lemma 1

We will show that for all $\Delta_{\texttt{ante}}$ and $\Delta_{\texttt{conseq}}$ such that

$$\Delta_{\texttt{ante}} \vdash \Delta_{\texttt{conseq}} \leadsto (\mathcal{R}, \Delta_{\texttt{frame}})$$

and $\Gamma{=}\{\texttt{U}_1(\bar{\texttt{v}}_1){\equiv}\Delta_1, ..\texttt{U}_\texttt{n}(\bar{\texttt{v}}_\texttt{n}){\equiv}\Delta_\texttt{n}\}$, a set of instantiations for unknown predicates such that $\mathcal{R}(\Gamma)$ then the entailment $\Gamma{:}\Delta_{\texttt{ante}} \vdash \Delta_{\texttt{conseq}}{*}\Delta_{\texttt{frame}}$ holds. We will show by structural induction on $\Delta_{\texttt{conseq}}$.

Due to the construction of the $\Delta_{\texttt{ante}} \vdash \Delta_{\texttt{conseq}} \leadsto (\mathcal{R}, \Delta_{\texttt{frame}})$ procedure as an extension of an existing entailment procedure with frame inference, for all $\Delta_{\texttt{ante}}$ and $\Delta_{\texttt{conseq}}$ not involving unknown predicates $\mathcal{R} = \texttt{true}$ and $\Gamma{:}\Delta_{\texttt{ante}} \vdash \Delta_{\texttt{conseq}}{*}\Delta_{\texttt{frame}}$.

Bellow we consider the cases that actually involve unknown predicates. These cases fall under two categories:

- $\Delta_{\texttt{ante}} = \texttt{U}(\texttt{r}, \bar{\texttt{v}}_\texttt{i}, \bar{\texttt{v}}_\texttt{n}\#){*}\kappa_1{\wedge}\pi_1$ and $\Delta_{\texttt{conseq}} = \kappa_s{*}\kappa_2{\wedge}\pi_2$ where $\kappa_\texttt{s} \equiv \texttt{r}{\mapsto}\texttt{c}(\bar{\texttt{d}}, \bar{\texttt{p}})$ or $\kappa_\texttt{s} \equiv \texttt{P}(\texttt{r}, \bar{\texttt{d}}, \bar{\texttt{p}})$. By hypothesis $\Delta_{\texttt{ante}} \vdash \Delta_{\texttt{conseq}} \leadsto (\mathcal{R}, \Delta_{\texttt{frame}})$. Then, as described in Sec. 4.6 the [**SO-ENTAIL-UNFOLD**] step must hold ensuring the following assertion holds:

  $$\kappa_1{*}\Delta_{\texttt{dangl}}{*}\Delta_{\texttt{rem}}{\wedge}\pi_1 \vdash \kappa_2{\wedge}\pi_2 \leadsto (\mathcal{R}', \Delta_{\texttt{frame}})$$

  where $\mathcal{R} = \mathcal{R}' \wedge (\texttt{U}(\texttt{r}, \bar{\texttt{v}}_\texttt{i}, \bar{\texttt{v}}_\texttt{n}\#) \wedge \pi_\texttt{a} \Rightarrow \kappa_\texttt{s} * \Delta_{\texttt{dangl}} * \Delta_{\texttt{rem}} \wedge \pi_\texttt{c})$ It follows from the structural induction hypothesis that:

  $$\Gamma{:}\kappa_1{*}\Delta_{\texttt{dangl}}{*}\Delta_{\texttt{rem}}{\wedge}\pi_1 \vdash (\kappa_2{\wedge}\pi_2) * \Delta_{\texttt{frame}} \qquad (1)$$

From $\mathcal{R}(\Gamma)$ it follows that

$$\Gamma{:}\mathtt{U}(\mathtt{r},\bar{\mathtt{v}}_{\mathtt{i}},\bar{\mathtt{v}}_{\mathtt{n}}\#){\wedge}\pi_{\mathtt{a}}{\vdash}\kappa_{\mathtt{s}}{*}\Delta_{\mathtt{dangl}}{*}\Delta_{\mathtt{rem}}{\wedge}\pi_{\mathtt{c}} \qquad (2)$$

From equations 1 and 2 it follows that $\Gamma{:}\Delta_{\mathtt{ante}} \vdash \Delta_{\mathtt{conseq}} * \Delta_{\mathtt{frame}}$.

- $\Delta_{\mathtt{ante}}{=}\kappa_1{\wedge}\pi_1$ and $\Delta_{\mathtt{conseq}}{=}\mathtt{U}_{\mathtt{c}}(\bar{\mathtt{w}},\bar{\mathtt{z}}\#){*}\kappa_2{\wedge}\pi_2$.

  Let $\pi_r = \mathtt{project}(\bar{r},\pi_1)$ and $\pi_w = \mathtt{project}(\bar{w},\pi_1)$. By hypothesis $\Delta_{\mathtt{ante}} \vdash \Delta_{\mathtt{conseq}} \rightsquigarrow (\mathcal{R},\Delta_{\mathtt{frame}})$. Then, as described in Sec.4.6 the [**SO-ENTAIL-FOLD**] step must hold ensuring the following assertions hold:

  - $\kappa_1 = \kappa_{11}{*}\kappa_{12}$
  - $\kappa_{12} \wedge \pi_1 \vdash \kappa_2 \wedge \pi_2 \rightsquigarrow (\mathcal{R}',\Delta_{\mathtt{frame}})$ which by structural induction leads to $\Gamma{:}\kappa_{12} \wedge \pi_1 \vdash (\kappa_2 \wedge \pi_2) * \Delta_{\mathtt{frame}}$
  - $\mathcal{R}{=}(\kappa_{11}{\wedge}\pi_{\mathtt{w}}{\Rightarrow}\mathtt{U}_{\mathtt{c}}(\bar{\mathtt{w}},\bar{\mathtt{z}}\#)\,@\,\kappa_{\mathtt{g}}{\wedge}\pi_{\mathtt{r}}) \ \wedge\mathcal{R}'$ which by $\mathcal{R}(\Gamma)$ leads to: $\Gamma{:}\kappa_{11}{\wedge}\pi_{\mathtt{w}} \vdash \mathtt{U}_{\mathtt{c}}(\bar{\mathtt{w}},\bar{\mathtt{z}}\#)\,@\,\kappa_{\mathtt{g}}{\wedge}\pi_{\mathtt{r}}$ Note that by the definition in Sec.2.2 for guarded assumptions, it follows that $\Delta\,@\,(\kappa_g{\wedge}\pi_r) * \kappa_g{\wedge}\pi_r$ is equivalent with $\Delta * (\kappa_g{\wedge}\pi_r)$.

  From the above three assertions it follows that $\Gamma{:}\Delta_{\mathtt{ante}} \vdash \Delta_{\mathtt{conseq}} * \Delta_{\mathtt{frame}}$

□

.2.2 Proof for Lemma 2

We will show that given a set of relational assumptions $\mathcal{R}$ and one of the synthesis rules is applied to obtain $\mathcal{R}'$ then if exists Γ such that $\mathcal{R}'(\Gamma)$ then $\mathcal{R}(\Gamma)$.

- If [**Syn-Base**] was applied to $\mathcal{R} \wedge (\mathtt{U}^{\mathtt{pre}}(\bar{\mathtt{x}}){*}\kappa{\wedge}\pi{\Rightarrow}\mathtt{U}^{\mathtt{post}}(\bar{\mathtt{y}}))$ then the resulting assumptions are either:

1. $\mathcal{R}'{=}\mathcal{R} \wedge (\mathtt{U}^{\mathtt{pre}}(\bar{\mathtt{x}}){\wedge}\pi'{\Rightarrow}\mathtt{emp}) \wedge (\kappa{\wedge}\pi \Rightarrow \mathtt{U}^{\mathtt{post}}(\bar{\mathtt{y}}))$

   OR

2. $\mathcal{R}'{=}\mathcal{R} \wedge (\mathtt{U}^{\mathtt{fr}}(\bar{\mathtt{x}}) \Rightarrow \top) \;\wedge\; (\mathtt{U}^{\mathtt{pre}}(\bar{\mathtt{x}}){\Rightarrow}\mathtt{U}^{\mathtt{fr}}(\bar{\mathtt{x}})\,\mathtt{@}\,(\kappa_{\mathtt{g}}{\wedge}\pi_{\mathtt{g}})){\wedge}$
   $(\mathtt{U}^{\mathtt{pre}}(\bar{\mathtt{x}}){*}\kappa{\wedge}\pi \Rightarrow \mathtt{U}^{\mathtt{post}}(\bar{\mathtt{y}}))$ where $\mathtt{U}^{\mathtt{fr}}$ is a fresh unknown predicate

If there exists Γ such that $\mathcal{R}'(\Gamma)$ then by definition, using the Γ interpretation for the unknown predicates then either:

1. $\Gamma{:}\mathtt{U}^{\mathtt{pre}}(\bar{\mathtt{x}}){\wedge}\pi' \vdash \mathtt{emp}$ and $\Gamma{:}\kappa{\wedge}\pi \vdash \mathtt{U}^{\mathtt{post}}(\bar{\mathtt{y}})$ and since by construction $\pi{\vdash}\pi'$ it follows that $\Gamma{:}\mathtt{U}^{\mathtt{pre}}(\bar{\mathtt{x}}){*}\kappa{\wedge}\pi \vdash \mathtt{U}^{\mathtt{post}}(\bar{\mathtt{y}})$ thus $\mathcal{R}(\Gamma)$.

2. $\Gamma{:}\mathtt{U}^{\mathtt{pre}}(\bar{\mathtt{x}}){*}\kappa{\wedge}\pi \vdash \mathtt{U}^{\mathtt{post}}(\bar{\mathtt{y}})$ and $\Gamma{:}\mathtt{U}^{\mathtt{pre}}(\bar{\mathtt{x}}) \vdash \mathtt{U}^{\mathtt{fr}}(\bar{\mathtt{x}})\,\mathtt{@}\,(\kappa_{\mathtt{g}}{\wedge}\pi_{\mathtt{g}})$ and $\Gamma{:}\mathtt{U}^{\mathtt{fr}}(\bar{\mathtt{x}}) \vdash \top$ leading to $\Gamma{:}\mathtt{U}^{\mathtt{pre}}(\bar{\mathtt{x}}){*}\kappa{\wedge}\pi \vdash \mathtt{U}^{\mathtt{post}}(\bar{\mathtt{y}}) * \top\,\mathtt{@}\,(\kappa_g{\wedge}\pi_g)$ which by construction of $\kappa_g{\wedge}\pi_g$ and σ_2 leads to $\Gamma{:}\mathtt{U}^{\mathtt{pre}}(\bar{\mathtt{x}}){*}\kappa{\wedge}\pi \vdash \mathtt{U}^{\mathtt{post}}(\bar{\mathtt{y}})$ and thus $\mathcal{R}(\Gamma)$.

- If [**Syn-Case**] was applied to:

$$\mathcal{R} \wedge (\mathtt{U}(\bar{\mathtt{x}}){\wedge}\pi_1{\Rightarrow}\Delta_1\,\mathtt{@}\,\Delta_{1\mathtt{g}}){\wedge}(\mathtt{U}(\bar{\mathtt{x}}){\wedge}\pi_2{\Rightarrow}\Delta_2\,\mathtt{@}\,\Delta_{2\mathtt{g}})$$

To generate:

$$\begin{aligned}\mathcal{R}' = \mathcal{R} \wedge (&\mathtt{U}(\bar{\mathtt{x}}){\wedge}\pi_1{\wedge}\neg\pi_2{\Rightarrow}\Delta_1\,\mathtt{@}\,\Delta_{3\mathtt{g}}){\wedge}\\ &(\mathtt{U}(\bar{\mathtt{x}}){\wedge}\pi_2{\wedge}\neg\pi_1{\Rightarrow}\Delta_2\,\mathtt{@}\,\Delta_{3\mathtt{g}}) \wedge\; (\mathtt{U}(\bar{\mathtt{x}}){\wedge}\pi_1{\wedge}\pi_2{\Rightarrow}\Delta_3\,\mathtt{@}\,\Delta_{3\mathtt{g}})\end{aligned}$$

with $\pi_1{\wedge}\pi_2 \not\Rightarrow \mathtt{false}$ and $\Delta_1{\wedge}\Delta_2{\Rightarrow}^{\bar{\mathtt{x}}}_{\wedge}\Delta_3$ and $\Delta_{1g}{\wedge}\Delta_{2g}{\Rightarrow}^{\bar{x}}_{\wedge}\Delta_{3g}$ and exists a Γ such that $\mathcal{R}'(\Gamma)$. From $\mathcal{R}'(\Gamma)$ it follows that:

$$\Gamma{:}\mathtt{U}(\bar{\mathtt{x}}){\wedge}\pi_1{\wedge}\neg\pi_2 \vdash \Delta_1\,\mathtt{@}\,\Delta_{3\mathtt{g}} \quad \Gamma{:}\mathtt{U}(\bar{\mathtt{x}}){\wedge}\pi_2{\wedge}\neg\pi_1 \vdash \Delta_2\,\mathtt{@}\,\Delta_{3\mathtt{g}}$$
$$\Gamma{:}\mathtt{U}(\bar{\mathtt{x}}){\wedge}\pi_1{\wedge}\pi_2 \vdash \Delta_3\,\mathtt{@}\,\Delta_{3\mathtt{g}}$$

We need to show that:

$\Gamma{:}\mathtt{U}(\bar{\mathtt{x}})\wedge\pi_1 \vdash \Delta_1 \,@\, \Delta_{1g}$ and $\Gamma{:}\mathtt{U}(\bar{\mathtt{x}})\wedge\pi_2 \vdash \Delta_2 \,@\, \Delta_{2g}$.

Note that by the definition of the conjunctive unification, it follows that if $\Delta_1\wedge\Delta_2\Rightarrow^{\bar{\mathtt{x}}}_{\wedge}\Delta_3$ then $\Gamma{:}\Delta_3 \vdash \Delta_1$ and $\Gamma{:}\Delta_3 \vdash \Delta_2$. Thus from $\Gamma{:}\mathtt{U}(\bar{\mathtt{x}})\wedge\pi_1\wedge\pi_2 \vdash \Delta_3 \,@\, \Delta_{3g}$ and follows that:

$\Gamma{:}\mathtt{U}(\bar{\mathtt{x}})\wedge\pi_1\wedge\pi_2 \vdash \Delta_1 \,@\, \Delta_{3g}$ and $\Gamma{:}\mathtt{U}(\bar{\mathtt{x}})\wedge\pi_1\wedge\pi_2 \vdash \Delta_2 \,@\, \Delta_{3g}$ Thus it follows: $\Gamma{:}\mathtt{U}(\bar{\mathtt{x}})\wedge\pi_1\wedge(\neg\pi_2 \vee \pi_2) \vdash \Delta_1 \,@\, \Delta_{3g}$ which simplifies to $\Gamma{:}\mathtt{U}(\bar{\mathtt{x}})\wedge\pi_1 \vdash \Delta_1 \,@\, \Delta_{3g}$ which by the construction of Δ_{3g} leads to $\Gamma{:}\mathtt{U}(\bar{\mathtt{x}})\wedge\pi_1 \vdash \Delta_1 \,@\, \Delta_{1g}$. Similarly we obtain: $\Gamma{:}\mathtt{U}(\bar{\mathtt{x}})\wedge\pi_2 \vdash \Delta_2 \,@\, \Delta_{2g}$.

- $\left[\underline{\textbf{Syn-Group-Pre}}\right]$ The proof obligation reduces to: if there exists Γ such that $\Gamma{:}\mathtt{U}(\bar{\mathtt{v}}) \wedge (\pi_1\vee\pi_2) \vdash \Phi^g{}_1\wedge\pi_1 \vee \Phi^g{}_2\wedge\pi_2$ and $\pi_1\wedge\pi_2 \vdash \mathtt{false}$ then $\Gamma{:}\mathtt{U}(\bar{\mathtt{v}})\wedge\pi_1 \vdash \Phi^g{}_1$ and $\Gamma{:}\mathtt{U}(\bar{\mathtt{v}})\wedge\pi_2 \vdash \Phi^g{}_2$. It follows that $\Gamma{:}\mathtt{U}(\bar{\mathtt{v}})\wedge\pi_1 \vdash \Phi^g{}_1\wedge\pi_1\vee\Phi^g{}_2\wedge\pi_2$ and
$\Gamma{:}\mathtt{U}(\bar{\mathtt{v}})\wedge\pi_2 \vdash \Phi^g{}_1\wedge\pi_1\vee\Phi^g{}_2\wedge\pi_2$. And since $\pi_1\wedge\pi_2 \vdash \mathtt{false}$ it follows that $\Gamma{:}\mathtt{U}(\bar{\mathtt{v}})\wedge\pi_1 \vdash \Phi^g{}_1\wedge\pi_1$ and $\Gamma{:}\mathtt{U}(\bar{\mathtt{v}})\wedge\pi_2 \vdash \Phi^g{}_2\wedge\pi_2$.

- $\left[\underline{\textbf{Syn-Group-Post}}\right]$ It follows trivially that $\Gamma{:}\Delta_{\mathtt{a}} \vdash \mathtt{U}^{\mathtt{post}}(\bar{\mathtt{v}})$ and $\Gamma{:}\Delta_{\mathtt{b}} \vdash \mathtt{U}^{\mathtt{post}}(\bar{\mathtt{v}})$ from $\Gamma{:}\Delta_{\mathtt{a}} \vee \Delta_{\mathtt{b}} \vdash \mathtt{U}^{\mathtt{post}}(\bar{\mathtt{v}})$.

- if $\left[\underline{\textbf{Syn-Inline}}\right]$ was applied to :

$$\mathcal{R}\wedge(\mathtt{U}^{\mathtt{pre}}(\bar{\mathtt{x}})\wedge\pi_{\mathtt{a}} \Rightarrow (\mathtt{U}^{\mathtt{pre}}_{\mathtt{d}}(\bar{\mathtt{x}}) * \kappa\wedge\pi) \,@\, (\kappa_{\mathtt{g}}\wedge\pi_{\mathtt{g}}))$$

resulting in the assumption set:

$$\mathcal{R}' = \mathcal{R}\wedge(\mathtt{U}^{\mathtt{pre}}(\bar{\mathtt{x}})\wedge\pi_{\mathtt{a}}\Rightarrow \bigvee_{\Delta_{\mathtt{i}}\in\mathcal{S}_1\cup\mathcal{S}_2} (\Delta_{\mathtt{i}} \,@\, (\kappa_{\mathtt{g}}\wedge\pi_{\mathtt{g}})))$$

when

$$\mathtt{U}_{\mathtt{d}}^{\mathtt{pre}}(\bar{\mathtt{x}}) \equiv \Delta_1 \,@\,(\kappa_1 \wedge \pi_1) \vee \ldots \vee \Delta_{\mathtt{n}} \,@\,(\kappa_{\mathtt{n}} \wedge \pi_{\mathtt{n}})$$

$$\mathcal{S}_1 = \{\Delta_{\mathtt{i}} * \Delta_{\mathtt{r}} * \kappa_{\mathtt{i}} \mid \kappa \wedge \pi \vdash \kappa_{\mathtt{i}} \wedge \pi_{\mathtt{i}} \rightsquigarrow (\emptyset, \Delta_{\mathtt{r}})\}$$

$$\mathcal{S}_2 = \{\kappa * \Delta_{\mathtt{i}} \wedge \pi \,@\,(\kappa_{\mathtt{i}} \wedge \pi_{\mathtt{i}}) \mid \mathtt{SAT}(\kappa \wedge \kappa_{\mathtt{i}} \wedge \pi \wedge \pi_{\mathtt{i}}), \kappa \wedge \pi \not\vdash \kappa_{\mathtt{i}} \wedge \pi_{\mathtt{i}}\}$$

We need to prove that if exists Γ such that $\mathcal{R}'(\Gamma)$ then, $\mathcal{R}(\Gamma)$. That is: $\Gamma{:}\mathtt{U}^{\mathtt{pre}}(\bar{\mathtt{x}}) \wedge \pi_{\mathtt{a}} \vdash (\mathtt{U}_{\mathtt{d}}^{\mathtt{pre}}(\bar{\mathtt{x}}) * \kappa \wedge \pi) \,@\,(\kappa_{\mathtt{g}} \wedge \pi_{\mathtt{g}})$. Which by using the $\mathtt{U}_{\mathtt{d}}^{\mathtt{pre}}$ definition translates in having to prove:

$$\Gamma{:}\mathtt{U}^{\mathtt{pre}}(\bar{\mathtt{x}}) \wedge \pi_{\mathtt{a}} \vdash \bigvee_{\mathtt{i} \in 1 \ldots \mathtt{n}} ((\Delta_{\mathtt{i}} \,@\,(\kappa_{\mathtt{i}} \wedge \pi_{\mathtt{i}}) * \kappa \wedge \pi) \,@\,(\kappa_{\mathtt{g}} \wedge \pi_{\mathtt{g}}))$$

From $\mathcal{R}'(\Gamma)$ it follows that:

$$\Gamma{:}\mathtt{U}^{\mathtt{pre}}(\bar{\mathtt{x}}) \wedge \pi_{\mathtt{a}} \vdash \bigvee_{\Delta_{\mathtt{i}} \in \mathcal{S}_1 \cup \mathcal{S}_2} (((\kappa \wedge \pi) * \Delta_{\mathtt{i}}) \,@\,(\kappa_{\mathtt{g}} \wedge \pi_{\mathtt{g}}))$$

We will show that:

$$\begin{aligned} &\textstyle\bigvee_{\Delta_{\mathtt{i}} \in \mathcal{S}_1 \cup \mathcal{S}_2} (((\kappa \wedge \pi) * \Delta_{\mathtt{i}}) \,@\,(\kappa_{\mathtt{g}} \wedge \pi_{\mathtt{g}})) \equiv \\ &\qquad\qquad \textstyle\bigvee_{\mathtt{i} \in 1 \ldots \mathtt{n}} (\Delta_{\mathtt{i}} \,@\,(\kappa_{\mathtt{i}} \wedge \pi_{\mathtt{i}}) * \kappa \wedge \pi) \,@\,(\kappa_{\mathtt{g}} \wedge \pi_{\mathtt{g}}) \end{aligned}$$

Observe that by the definition of the guard assertion, a RHS disjunction $(\Delta_{\mathtt{i}} \,@\,(\kappa_{\mathtt{i}} \wedge \pi_{\mathtt{i}}) * \kappa \wedge \pi)$ where the guard $\kappa_{\mathtt{i}} \wedge \pi_{\mathtt{i}}$ contradicts the context $\kappa \wedge \pi$ is equivalent to `false` and thus can be discarded, leaving only disjuncts that do not contradict the context. Note that by construction, $\mathcal{S}_1 \cup \mathcal{S}_2$ denotes exactly that set. Furthermore, by the definition of the guarded assumption, assertions $(\Delta_{\mathtt{i}} \,@\,(\kappa_{\mathtt{i}} \wedge \pi_{\mathtt{i}}) * \kappa \wedge \pi)$ in which $\kappa \wedge \pi \vdash \kappa_{\mathtt{i}} \wedge \pi_{\mathtt{i}}$ can be reduced to $\Delta_{\mathtt{i}} * \kappa \wedge \pi$. Observe that the result of the application of the above two equivalence preserving simplification steps on the RHS is identical to the LHS. Thus the required disjunction equivalence holds.

.2.3 Proof for Lemma 3

Follows from the observation that there are only two rules generating predicate definitions: $\boxed{\textbf{Syn-Pre-Def}}$ and $\boxed{\textbf{Syn-Post-Def}}$. Each applicable only if there exists only one assumption corresponding to the predicate that is currently being derived. Each rule generates exactly the predicate definition that would satisfy the unique assumption.

.2.4 Proof for Lemma 4

We observe that the algorithm in Fig.5-1 finishes only when all assumptions have been catered for: assumptions used for synthesis have been reduced to a unique assumption which becomes the predicate definition; assumptions not included in the synthesis are discharged by an entailment step. Thus by the previous lemmas and the soundness of the underlying entailment checker the resulting definitions satisfy all the initial assumptions.

.2.5 Proof for Lemma 5

We will show that all normalization steps are meaning preserving:

- Dangling elimination: We need to show that if $\mathtt{U_d}(\mathtt{x}, \bar{\mathtt{v}}) \equiv \top$ and $\mathtt{x} \notin \bar{\mathtt{y}}$ then:

$$\mathtt{U}(\bar{\mathtt{y}}) \equiv ([\mathtt{x} \mapsto \mathcal{D}_{\mathtt{U_d}}](\kappa \wedge \pi)) \,@\, (\kappa_{\mathtt{g}} \wedge \pi_{\mathtt{g}}) \vee \Phi^g$$

  is equivalent to:

$$\mathtt{U}(\bar{\mathtt{y}}) \equiv (\mathtt{U_d}(\mathtt{x}, \bar{\mathtt{v}}) * \kappa \wedge \pi) \,@\, (\kappa_{\mathtt{g}} \wedge \pi_{\mathtt{g}}) \;\vee\; \Phi^g$$

  We first observe that variables local to the predicate definition, not part of the predicate arguments, are implicitly existentially quantified.As mentioned, we use the $\mathcal{D}_{\mathtt{U}_d}$ notation as a visual aid, to identify an instance of the predicate $\mathtt{U}_d$ whose root pointer $\mathtt{x}$ is reachable but has been neither read nor written to. By expanding the notation, the equivalence to be proven becomes:

$$\exists \mathtt{x}.(\mathtt{U_d}(\mathtt{x},\bar{\mathtt{v}}) * \kappa \wedge \pi) \equiv \exists \mathtt{f_v}.[\mathcal{D}_{\mathtt{U_d}} \mapsto \mathtt{f_v}]([\mathtt{x} \mapsto \mathcal{D}_{\mathtt{U_d}}](\kappa \wedge \pi)) * \mathtt{U_d}(\mathtt{f_{v2}}, \ldots)$$

By applying the predicate definitions:

$$\exists \mathtt{x}.(\top * \kappa \wedge \pi) \equiv \exists \mathtt{f_v}.[\mathcal{D}_{\mathtt{U_d}} \mapsto \mathtt{f_v}]([\mathtt{x} \mapsto \mathcal{D}_{\mathtt{U_d}}](\kappa \wedge \pi)) * \top$$

Which holds trivially.

- Eliminating useless parameters: We need to show that if at this step a predicate $\mathtt{P}(\bar{\mathtt{x}}) \equiv \Delta_1$ is distilled into $\mathtt{Q}(\bar{\mathtt{x}}') \equiv \Delta_2$ then $P'(\bar{x}) \equiv Q(\bar{x}')$ that is, Δ_1 holds iff Δ_2 holds. By construction $\bar{\mathtt{x}}' = \bar{\mathtt{x}} \setminus \bar{\mathtt{z}}$ and $\exists \bar{\mathtt{z}}.\Delta_1 \vdash \Delta_2$ and also $\Delta_2 \vdash \exists \bar{\mathtt{z}}.\Delta_1$ which leads to $\Delta_2 \equiv \exists \bar{\mathtt{z}}.\Delta_1$ and by the soundness of the flow analysis used to detect that variables $\bar{z}$ are not used in Δ_1 it follows that $\Delta_1 \equiv \exists \bar{\mathtt{z}}.\Delta_1$.

- Re-using predicates: We need to show that if at this step a predicate $\mathtt{P}(\bar{\mathtt{x}}) \equiv \Delta_1$ is found to be equivalent with $\mathtt{Q}(\bar{\mathtt{x}}') \equiv \Delta_2$ then Δ_1 holds iff Δ_2 holds. By the premise of this normalization step, $\Delta_1 \vdash \mathtt{Q}(\bar{\mathtt{x}}')$ and also $\Delta_2 \vdash \mathtt{P}(\bar{\mathtt{x}})$ which leads to $\Delta_2 \equiv \Delta_1$.

- Predicate splitting: Soundness follows from the construction. Given a predicate $\mathtt{P}(\bar{\mathtt{x}})$ we need to show that if the bi-abduction succeeds in discovering definitions for $\mathtt{U_1}(\bar{\mathtt{x}})$ and $\mathtt{U_2}(\bar{\mathtt{x}})$ such that $\mathtt{P}(\bar{\mathtt{x}}) \vdash \mathtt{U_1}(\bar{\mathtt{x}}) * \mathtt{U_2}(\bar{\mathtt{x}})$ and if the derived predicate definitions can be used to prove $\mathtt{P}(\bar{\mathtt{x}}) \equiv \mathtt{U_1}(\bar{\mathtt{x}}) * \mathtt{U_2}(\bar{\mathtt{x}})$ which follows from the soundness of the bi-abduction and of the entailment methods.