View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by ScholarBank@NUS

ENHANCING TOTAL CORRECTNESS PROOFS IN PROGRAM VERIFICATION

LE TON CHANH

B.Eng. in Computer Science

Ho Chi Minh City University of Technology

A THESIS SUBMITTED

FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

SCHOOL OF COMPUTING

NATIONAL UNIVERSITY OF SINGAPORE

2014

DECLARATION

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

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

Le Ton Chanh

24 November 2014

ACKNOWLEDGMENTS

First of all, I would like to express my gratitude and appreciation to my advisor, Professor Wei-Ngan Chin, for his continuous and thoughtful guidance, advice and support throughout my Ph.D journey. I deeply thank him for his patience and encouragement that carried me on through the most difficult time when I lost my faith in research and for his insights and inspiration that brought me back to the right track.

I wish to thank my Thesis Committee Members: Professors Joxan Jaffar, Hugh Anderson and Kwangkeun Yi for their invaluable comments and feedbacks on my thesis. I am thankful to Shengchao Qin, Razvan Voicu, Aquinas Hobor and especially to Cristian Gherghina for the fruitful research collaborations. I am mostly grateful to my Teachers – Mr. Khuong, Mr. Chi, Mr. Loc and Mr. Tiep – who fostered my love in math and Dr. Nguyen Hua Phung who showed me the beauty of programming languages and guided my first steps in research.

I thank Khanh, Trung, Hoang, An, Truc, Trang and other VNUTown++ members for their friendship. Activities like cooking, trekking, cycling, gaming, etc. with them made Singapore my second home.

Last but certainly not least, I thank my dear grandfather, my parents, my sister and my brother for their unconditional love and support. I thank my better half – Diem My – for her endless love and her trust, patience and understanding.

Enhancing Total Correctness Proofs in Program Verification
by
Le Ton Chanh
Submitted to the School of Computing,
National University of Singapore
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy in Computer Science

Abstract

Proving the total correctness of large-scale software systems with complex safety and liveness properties is a great challenge in program verification. To specify these properties and verify or analyze them successfully, the software verification systems usually require expressive specification logics with scalable verification techniques to be developed. However, recent advances in software verification mainly focus on partial correctness with safety properties.

The aim of this thesis is to develop methodologies to enhance *expressiveness*, focusing on program termination and non-termination reasoning, and *scalability*, focusing on the concept of modularity, of total correctness proofs in program verification.

Firstly, we propose a logical framework for specifying and verifying termination and non-termination properties of programs. These properties are defined as resource capacity of execution length and reasoned about in terms of resource reasoning. This approach allows the termination and non-termination assertions to be seamlessly integrated into available logics for functional properties to conduct more intricate termination and non-termination proofs. Its result is a unified framework, which can verify both partial correctness, termination and non-termination of various programs, including heap-manipulating programs. Experimental evaluation shows the expressiveness, usability and practicality of our approach on over 300

challenging programs.

Secondly, we propose a modular inference mechanism for summarizing termination and non-termination behaviors of each method in programs. We extend the proposed termination logic with second-order termination predicates and leverage the available Hoare-style verification infrastructure to collect a set of relational assumptions on them. We then solve these assumptions with case analysis to determine both termination and non-termination behaviors of analyzed methods. The inference result is expressed in a compatible logic form of the underlying verification system, so that they can be re-verified. Experimental evaluation on the benchmark suite of a recent termination competition shows the scalability and efficiency of our mechanism against state-of-the-art termination analyzers.

Lastly, we propose a formal framework for proof slicing in verification that can aggressively reduce the size of the discharged proof obligations as a means of performance improvement. Our proposal is built on top of existing automated theorem provers and can be viewed as a re-engineering effort in proof decomposition that attempts to avoid large-sized proofs for which these provers may be particularly inefficient. Our theoretical development is supported by experimental results, which show significant improvements in the verification of complex programs.

Table of Contents

1 Introduction **1**

1.1 Thesis Objectives 6

1.2 Contributions of the Thesis 8

1.3 Organization of the Thesis 10

2 HIP/SLEEK Verification System **13**

2.1 Programming Languages 14

2.2 Specification Language 15

2.3 Forward Verification Rules 16

3 A Resource-Based Logic for Termination and Non-Termination Proofs **19**

3.1 Overview 20

3.2 From Resource to Temporal Logic 24

3.3 A Logic for Resource Reasoning 28

3.3.1 Resource Capacity 28

3.3.2 Assertion Language and Semantics for a Resource-Aware Logic 30

3.3.3 Resource-Enhanced Entailment with Frame Inference . . 32

3.3.4 Hoare Logic for Resource Verification 33

3.4 (Non-)Termination Proofs via Resource Reasoning 34

3.4.1 From Termination Measures to Execution Capacity's Finite Upper Bounds 36

3.4.2 Termination and Non-Termination Verification 38

3.4.3 Flow-Insensitive Temporal Logic 40

3.5 Experiments 42

3.5.1 Numerical Programs 42

3.5.2 Heap-manipulating Programs 44

3.6 Discussion 46

4 Termination and Non-Termination Specification Inference 49

4.1 Introduction 50
4.2 Overview of Our Approach 51
4.2.1 Other Examples 59
4.3 From Verification to Inference 62
4.4 Generation of Temporal Relational Assumptions 65
4.5 A Mechanism for Termination and Non-Termination Inference . 67
4.5.1 Inferring Base Case Termination 69
4.5.2 Specializing Relational Assumptions 72
4.5.3 Resolving Temporal Reachability Graph 72
4.5.4 Inferring Ranking Function 75
4.5.5 Inferring Inductive Unreachability 77
4.5.6 Abductive Case-Splitting 79
4.6 Experiments 80
4.6.1 Tricky While Program 82
4.6.2 Phase Change Recursive Programs 83
4.7 Discussion 85

5 A Proof Slicing Framework for Program Verification 87

5.1 Overview 88
5.2 Proof Slicing for Program Verification 90
5.3 A Framework for Proof Slicing 93
5.4 Complete Proof Slicing 97
5.5 An Annotation Scheme for Proof Slicing 99
5.6 Aggressive Proof Slicing 103
5.6.1 Annotation Scheme 103
5.6.2 Annotation Reduction 103
5.6.3 Slicing Criterion 105
5.6.4 Relevance Criterion 106
5.7 Experiments 107
5.8 Discussion 112

6 Conclusions 115

6.1 Future Work 117

A **133**

A.1 Proofs for Lemmas 133

A.2 Soundness Proofs 134

B **141**

List of Figures

2-1 A Core Imperative Language 14
2-2 HIP/SLEEK's Specification Language 15
2-3 Hoare-style Forward Verification Rules 17

3-1 The Shuffle problem from the Termination Competition 21
3-2 Examples on numerical programs 25
3-3 A specification in separation logic to verify the correctness of Shuffle's methods 27
3-5 Semantics of Assertions in the Resource-Aware Logic 30
3-4 The Assertion Language 31
3-6 Hoare Verification Rules: Method Call and Return 38
3-7 Termination Verification for Numerical Programs 42
3-8 Termination Verification for Heap-manipulating Programs . . . 44
3-9 Termination Verification for the SIR/Siemens and TPDB Benchmark 46

4-1 The `foo` example 51
4-2 A Specification Language with Unknown Predicates for Inference 53
4-3 Functions with Nested Recursion: Ackermann function (a) and McCarthy 91 function (b) 59
4-4 Specification with Implementation for `append` method of two linked lists 61
4-5 Overall Inference Algorithm 67
4-6 Core TNT Inference Algorithm 74
4-7 Procedure for Proving Termination over a scc 75
4-8 Proc. for Proving Non-Termination over a scc 77
4-9 Termination Outcomes on SV-COMP 2015's C Programs. . . . 81
4-10 A simple non-terminating program adapted from [48] 82
4-11 A phase-change recursive program 84

5-1 A code snippet and its verification conditions for total correctness proof 92
5-2 Framework for Proof Slicing Mechanisms 94
5-3 Complete Slicing Mechanism 98
5-4 The `avl` heap predicate 100
5-5 Support Logic with Annotation Scheme 103
5-6 Rules for Annotation Reduction 104
5-7 Annotated Slicing Mechanism 105
5-8 Relative Comparison (%) of CS over NS with various theorem provers. 108
5-9 Comparison of CS (c) and AS (a) over NS on examples with Weakly Linking Components (s: size, h: height, b: balance factor, t: sets, n#: number of (annotated) weakly linking components) 109
5-10 Comparison (%) of CS and AS over NS on the Spaguetti Benchmark with the number of heap variables from 10 to 20 (+ indicates caching used) 110
5-11 A simple contrived procedure 111
5-12 Comparison (%) of AS over NS on the Spring Benchmark with Frama-C. The number of parameters ranges from 2 to 30 and the number of generated proof obligations are given in the parentheses. 111

A-1 Key Rules in Operational Semantics 136

Chapter 1

Introduction

Nowadays, computer systems play a crucial role in many areas, which vary from scientific research, industrial manufacturing, financial transaction, etc. to our daily activities. Of significance are computers that are being used to control safety-critical systems, such as fly-by-wire airplanes, spacecrafts and health-care systems. Ensuring that the softwares of these systems behave correctly and reliably is a very important and long standing problem in software engineering practice since any failure from their operations might result in fatal consequences, which cost time, money and even human lives.

For example, in 1996, a faulty exception-handling routine in the software system of the Ariane 5 satellites launcher made the rocket explode right after its lift-off [108]. The accident caused a loss of billions of dollars and a decade of research and development. This is an example where serious repercussion are being caused by software bugs. Some other well-known software failures are listed in [142].

In most current day scenarios, *software testing* has been a common method to find the presence of bugs in programs but is not sufficient to guarantee that the programs are bug-free [53]. This highlights the importance of *software*

verification which aims to prove bug absence by mathematical techniques. In software verification, the correctness of a system with respect to the specified requirements is formally proven for all possible executions. As a result, the users can be assured that the system is safe to execute; *i.e.*, no unexpected behavior occurs during its execution.

There are two main *formal approaches* [39] in software verification to prove program correctness: model-based approach with *model checking* [40] and proof-based approach with *theorem proving* [135]. In both approaches, a specification Φ in the form of an appropriate logic, namely *specification language*, is provided for each software system to specify the desired properties of this system. However, these approaches differ from the way the software system is being represented and on how the given specification is proven.

In a model-based approach, the software system is described by a model $\mathcal{M}$. The model checkers, such as SPIN [82], UPPSLA [96] and BLAST [23], then exhaustively check that this model meets the given specification Φ.

In a proof-based approach, the software system is represented by a set of logical formulas Γ. Program verifiers, such as HIP/SLEEK [38], Dafny [106], Frama-C [50], KeY [4] and Boogie [13], then automatically generate a set of verification conditions to formally prove the specification Φ given the premises Γ. These verification conditions are finally discharged by automated theorem provers, like Z3 [52] and CVC family [14–16, 140] or interactive theorem provers, like ACL2 [89] and Coq [1].

In this thesis, we focus on program verification with Hoare logic. Hoare logic, first proposed by Floyd [60] and Hoare [76], is an important proof-based approach to specify and verify functional correctness properties (*i.e.*, the

input-output behaviors) of sequential programs. The fundamental concept of Hoare logic is the Hoare triple in the form of $\{P\}c\{Q\}$, which describes how a program state is changed due to the execution of the code segment c via a specification with precondition P and postcondition Q. The assertions P and Q, expressed using logical formulas over the input and output of the program, denote abstract program states, where each abstraction represents a set of concrete program states.

For each programming language, Hoare logic provides a set of axioms, which are Hoare triples defined for the basic statements of the language. However, a variant of Hoare logic together with an appropriate logical framework can be additionally constructed to express and reason about a specific feature of the programming language. For example, separation logic, introduced by Ishtiaq and O'Hearn [84] and Reynolds [133], is an extension of Hoare logic for specifying and verifying functional properties as well as proving memory safety of programs with shared mutable data structures. A big advantage of separation logic is that the information about nonsharing resources allocated on the heap can be concisely and precisely captured by the separating conjunction $*$. A heap space satisfies the assertion $p_1 * p_2$ if and only if this heap can be partitioned into two disjoint sub-heaps which respectively satisfy the assertion p_1 and p_2. With separation logic, explicitly handling all possible aliasing information of nonsharing portions of heap is not necessary. In addition, as local reasoning is enabled with the frame rule, separation logic also avoid scalability issues of the traditional Hoare logic when reasoning with heap-based programs [134]. Moreover, separation logic is also expressive and powerful enough to be further extended to reason about concurrency [25, 67, 117, 145], overlaid/sharing data structures [80, 103] and

even object-oriented programs [37, 122].

The correctness of a program c with respect to a formal specification is distinguished into either *(i) partial correctness*, which assumes the termination of the code c when proving the given specification, or *(ii) total correctness*, which requires a termination proof in addition to the partial correctness proof. Given the precondition P and the postcondition Q, the partial correctness proof of the program c aims to show that the Hoare triple $\{P\}c\{Q\}$ holds. That is, if c is executed from a concrete prestate satisfying the abstract state P and it terminates then the concrete poststate after the execution can be captured by Q.

Without proving the termination of c, a successful partial correctness proof cannot assure that this program is responsive and eventually returns the computational results upon to its termination. Non-termination is also considered as a software bug [70, 131] but more difficult to detect. For example, a non-termination bug in the Microsoft Zune's clock driver, which causes the devices to hang on the New Year's Eve of a leap year, had not been discovered until it occurred in 2008. Obviously, software testing is not capable of finding non-termination bugs. In [66], the authors proposed a special technique to do testing on legally non-terminating systems from reactive systems, not to detect non-termination per se.

To overcome this problem, Hoare logic for total correctness [9], denoted by the Hoare triple $[P]c[Q]$, requires the code fragment c to be shown terminating under the given precondition P in addition to satisfying the postcondition Q after its execution. The termination of a while-loop statement or a recursive method can be proven by showing that the given termination measure[1], a mathematical function over program variables, is decreased with respect to a

[1] Another term is ranking function or loop variant (only for while-loop), which can be used interchangeably.

well-founded relation. To do that, the Hoare rules for partial correctness of these constructs are modified to incorporate the decreasing and bounded checks on the termination measure into the proof. For comparison, we show below the Hoare rule of while-loop for partial correctness proof [PAR−WHILE] and for total correctness proof [TOT−WHILE].

$$\frac{\vdash \{P \wedge C\}\ S\ \{P\}}{\vdash \{P\}\ \texttt{while}\ C\ \texttt{do}\ S\ \{P \wedge \neg C\}}\ [\textsc{par-while}]$$

$$\frac{\vdash [P \wedge C \wedge (\mathcal{V} = \mathcal{V}_0)]\ S\ [P \wedge (\mathcal{V} < \mathcal{V}_0)] \qquad \vdash P \wedge C \Rightarrow (\mathcal{V} \geq 0)}{\vdash [P]\ \texttt{while}\ C\ \texttt{do}\ S\ [P \wedge \neg C]}\ [\textsc{tot-while}]$$

In the rule [TOT−WHILE], $\mathcal{V}$ denotes a loop variant (*i.e.*, the termination measure for a while-loop) and $\mathcal{V}_0$ captures the initial value of $\mathcal{V}$ at the beginning of a loop iteration. To reduce the annotation burden, loop variants can be inferred automatically by [41, 42, 126]. Furthermore, the program termination can be separately proved by termination analysis approaches, such as [27, 102, 120, 128].

On the other hand, the non-termination of a code c can be specified by the postcondition `false` as a partial correctness property. That is, the Hoare triple $\{P\}c\{\texttt{false}\}$ denotes the fact that the statements after c are *unreachable*, thus c does not terminate, for satisfiable instances of P, assuming that there is no "`assume (false)`" statement inside c. A non-termination specification mechanism is important to construct more comprehensive specifications for a better program understanding. In addition, it helps to clearly distinct expected non-terminating behaviors and failure of termination proofs.

1.1 Thesis Objectives

The overall aim of this dissertation is to enhance total correctness proofs for program verification in two dimensions: *expressiveness* of specification languages and *scalability* of proof techniques. These are two important dimensions for program verification to adapt toward the growth of large-scale software systems with various complex properties.

The research in this thesis is inspired from the success of separation logic [84, 133], which initiates a fruitful line of research on both theory and tool development for the verification of heap-manipulating programs [18, 19, 38, 54, 85, 105, 109, 125, 147], concurrent programs [25, 31, 78, 79, 117, 144, 145] and object-oriented programs [37, 119, 122]. In addition, separation logic is not only expressive, it also helps to build scalable tools and methods for program verification [22, 34, 57, 86, 141, 149]. However, the main theme of this thesis focuses on the other aspect of total correctness proofs; that is, developing an expressive and scalable logical framework for reasoning about program termination and non-termination.

The first study conducted in this thesis originates from the lack of a logic to properly specify the termination and the non-termination of a program in its specification. Although the Hoare triple $[P]c[Q]$ of total correctness denotes that the code c terminates, the termination measure involved in the proof rule is never explicitly declared in the triple (*e.g.*, see the rule [TOT−WHILE]). This is due to the termination measure being a mathematical function, not a logical assertion, so that it cannot be part of the pre/post specifications. This fact runs contrary to Cliff Jones' well-regarded expectation [87]. As a result, the termination proof is currently considered as being *logically distinct* from the partial correctness

proof.

Moreover, because non-termination obviously cannot be specified by a total correctness specification, *i.e.*, $\nvDash [P]$ `while` C `do` S [`false`] and requires partial correctness proof, termination and non-termination proof are separate. Therefore, the termination reasoning might not take non-termination into account, leading to inaccurate verification results.

To overcome these shortcomings, we propose a specification logic in which program termination and non-termination as well as functional correctness properties are specified and reasoned about in the same unified verification framework. In this logic, program termination and non-termination are specified by so-called temporal predicates. Their semantics are defined in terms of resource reasoning, which allows them to be soundly and seamlessly integrated into underlying logics for functional correctness. We expect that this proposal would result in an expressive specification language that can specify different termination and non-termination behaviors of each method in a modular fashion.

In the second part of this thesis, we propose an automated termination and non-termination inference to enhance the scalability of the above verification framework. Our aim is not to develop distinct termination and non-termination analyses which are separate from the verification process. Instead, we propose a unified mechanism that can simultaneously derive the preconditions for both termination and non-termination behaviors of a program. Moreover, this inference mechanism is proposed to be modularly built on top of existing verification systems and/or other (orthogonal) inference mechanisms for safety properties, so that it can handle a wider class of programs. These two proposals distinguish our mechanism from the current inference approaches which only

prove either termination [21, 29, 46, 48, 74, 92, 129] or non-termination [30, 35, 70, 94, 123, 146] for specific classes of programs.

As our termination and non-termination inference mechanism leverages on existing verification systems, we propose to improve the performance of the verification process as a means to further enhance the scalability of our framework, especially on large code bases. In the last part of this thesis, we develop a formal framework which accepts various proof slicing mechanisms to detect and filter irrelevant information in proof obligations generated during the verification process. This proof slicing framework is prover-independent and tailored to program verification, so that not only the termination and non-termination reasoning but also other verification and inference mechanisms would benefit from it. Moreover, the proposed framework also provides a common environment for further investigation and development of modular and extensible slicing mechanisms.

1.2 Contributions of the Thesis

The contributions of this thesis can be summarized by three main themes:

A Resource-based Logic for Termination and Non-Termination Proofs (Chapter 3, first presented in [98])

- A new *resource logic* that can capture lower and upper bounds on resource usage via the concept of resource capacity, together with an entailment procedure to support correctness proofs with resource-related properties.
- A *temporal logic* that is abstracted from the resource logic to reason about both program termination and non-termination. We introduce three

new temporal constraints, its *entailment* and *Hoare rules* lifted from the resource logic.

– A successful integration of both resource and temporal logics into an existing separation logic based verifier, namely HIP/SLEEK system [38]. The new temporal logic is expressive enough to specify and successfully verify the (non-)termination behaviors for about 300 benchmark programs collected from a variety of sources, including the SIR/Siemens test suite [55] and problems from the Termination Competition [111].

Termination with Non-Termination Specification Inference (Chapter 4, first presented in [100])

– A novel use of *unknown pre/post predicates* for inferring termination and non-termination properties.

– An inference of *base-case termination* preconditions from the absence of post-predicate.

– A concept of *inductive unreachability* to infer definite non-termination.

– A *ranking function synthesis* for termination from relational assumptions derived by pre-condition proving.

– An *abductive case splitting* to divide input spaces into terminating and non-terminating scenarios.

A Proof Slicing Framework for Program Verification (Chapter 5, first presented in [99])

– A formal and general framework for uniformly describing different proof slicing mechanisms. We prove the proposed slicing mechanisms to be

both sound and convergent, in the sense that, while non-deterministic, the framework always produces the same result for a given input. To showcase the framework, we introduce the formalism of a *complete slicing* as one of its applications.

– An annotation scheme for slicing that is suitable for a variety of logics. This is aimed at allowing parts of formulas to be identified as carrying information *linking* distinct properties. Then, an *aggressive proof slicing* mechanism can leverage on annotation schemes to achieve greater reductions of the proof slices. This also creates the opportunity for applying proof caching, which is particularly effective for smaller-sized proofs.

– An implementation of the both proof slicing mechanisms within HIP/SLEEK verification system. Our experiments show compelling performance gain of about 61% for complete proof slicing, and a further gain of 74% for aggressive proof slicing.

1.3 Organization of the Thesis

The thesis is organized as following.

Chapter 2 describes the technical background of our proposals. In this chapter, we introduce the programming language, the specification logic as well as the verification process of the HIP/SLEEK system [38], on which we will build the prototype for our proposals.

Chapter 3 presents a resource-based logic for specifying and reasoning about program termination and non-termination via a Hoare-style verification.

Chapter 4 introduces an inference mechanism that leverages on relational

assumptions over unknown termination behaviors of mutually recursive methods to construct a comprehensive summary of their termination and non-termination characteristics.

Chapter 5 presents a formalism of proof slicing mechanisms and demonstrate its application in the HIP/SLEEK verification system to achieve impressive improvement on verification time.

Chapter 6 concludes the thesis and discusses future research directions.

Chapter 2

HIP/SLEEK Verification System

In this chapter, we introduce the HIP/SLEEK verification system [38], on which we build a prototype for the proposals in this thesis.

HIP is a general-purpose automated verification system. By providing a rich specification language, the system allows users to specify the functional correctness as well as various safety properties of both numerical programs and heap-manipulating programs. In order to prove the partial correctness of a program with respect to a given specification, the system firstly relies on Hoare logic with forward verification rules to automatically construct a set of verification conditions and then discharges them by off-the-shelf theorem provers, such as the Omega calculator [132], the Z3 SMT solver [52], the monadic second-order logic prover MONA [90] and the Reduce/Redlog algebra system [56].

Specifically, the HIP verifier uses separation logic [84, 133] in the verification of heap-based programs. The shape and other pure properties (*e.g.*, size and bag) of complex data structures can be modeled via inductive heap predicates. To discharge proof obligations in the form of heap entailments, the system develops its own separation logic prover, namely SLEEK, with the

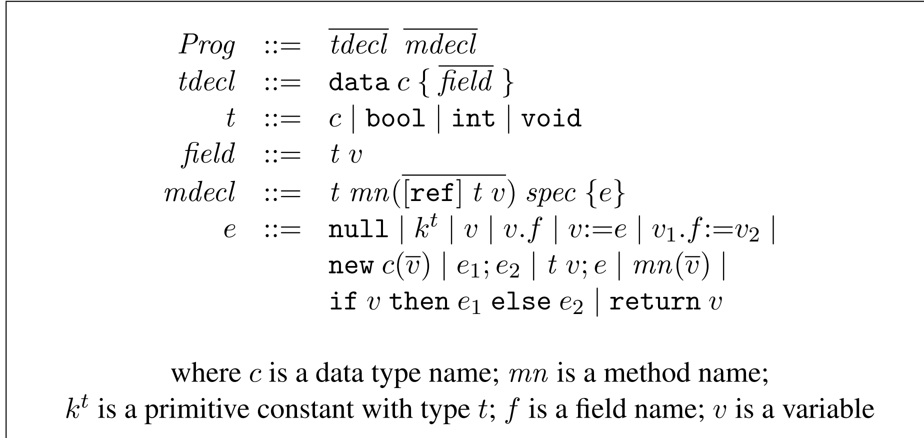


Figure 2-1. A Core Imperative Language

capability of frame inference for unconsumed heap space. The underlying entailment procedure of HIP can be described in the general form $\Phi_a \vdash \Phi_c \leadsto \Phi_r$ denoting that Φ_a entails Φ_c with the residue Φ_r. In this form, the residue Φ_r of an entailment containing only pure constraints is simply the antecedent Φ_a.

We now summarize the overall system in three aspects: the supporting programming languages, the specification language and the verification procedure.

2.1 Programming Languages

To facilitate the verification for programs in multiple front-end imperative programming languages (such as C and Java), the HIP/SLEEK system provides a core strict language in Fig. 2-1 with usual constructs, such as type/data structure declaration *tdecl*, method declaration *mdecl* with imperative statements like method call, assignment, etc. inside its body. A program consists of a set of data declarations and method declarations while a method declaration also contains a specification *spec* beside its prototype and body. We

$$
\begin{array}{rcl}
\mathit{spec} & ::= & \texttt{requires}\ \Phi_{pr}\ \texttt{ensures}\ \Phi_{po} \\
\mathit{hpred} & ::= & \texttt{pred}\ c(\overline{v}) \equiv \Phi\ \texttt{inv}\ \pi \\
\Phi & ::= & \bigvee(\exists \overline{v} \cdot \kappa \wedge \pi) \\
\kappa & ::= & \texttt{emp} \mid v{\mapsto}d(\overline{u}) \mid c(\overline{v}) \mid \kappa_1 * \kappa_2 \\
\pi & ::= & \phi \mid b \mid a \mid \pi_1 \wedge \pi_2 \mid \neg\pi \mid \exists v \cdot \pi \\
b & ::= & \texttt{false} \mid v \mid b_1 = b_2 \\
a & ::= & e_1{=}e_2 \mid e_1{<}e_2 \mid v{=}\texttt{null} \\
e & ::= & k \mid v \mid k \times e \mid e_1 + e_2 \mid -e \\
\phi & ::= & v{\in}\mathcal{B} \mid \mathcal{B}_1{=}\mathcal{B}_2 \mid \mathcal{B}_1{\sqsubset}\mathcal{B}_2 \mid \exists v{\in}\mathcal{B} \cdot \pi \\
\mathcal{B} & ::= & \mathcal{B}_1{\sqcup}\mathcal{B}_2 \mid \mathcal{B}_1{\sqcap}\mathcal{B}_2 \mid \mathcal{B}_1{-}\mathcal{B}_2 \mid \{\} \mid \{v\}
\end{array}
$$

where `emp` represents empty heap; k is a constant; v is a variable

Figure 2-2. HIP/SLEEK's Specification Language

use the overlie symbol $\overline{m}$ to indicate a set or a list of m.

For simplicity, this core language does not support loop constructs but it assumes an automatic translation of (nested) loops into tail-recursive methods with call-by-reference parameters (prefixed by the keyword `ref`). As a result, we only need to construct a general verification procedure for (recursive) method calls.

2.2 Specification Language

The syntax of HIP/SLEEK's specification language is given in Fig. 2-2. The basic form of a specification is a pair of precondition Φ_{pr} and postcondition Φ_{po}, which are logical formulas in disjunctive normal form. Furthermore, the specification language of HIP/SLEEK can be extended to multiple pre/post specification [36] or structured specification with case analysis [63] to capture more behaviors of the verified programs.

Each disjunct in the pre and postconditions consists of a heap formula κ and pure (heap-free) formula π. For specifying heap properties, the HIP/SLEEK

system uses a fragment of separation logic with the separation conjunction $*$ to denote the disjointness of heap parts and the heap predicate $hpred$ to denote various data structures. On the other hand, the system uses first-order logic with Presburger arithmetic and set theory for specifying pure properties of heap or constraints over program variables.

For example, given the data declaration

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

a list segment from $\texttt{root}$ to $\texttt{p}$ comprising $\texttt{n}$ data nodes can be described by the inductive heap predicate

$$\begin{aligned}&\mathbf{pred}\,\texttt{lseg}(\texttt{root},\texttt{p},\texttt{n}) \equiv \texttt{root}{=}\texttt{p} \wedge \texttt{n}{=}0\\&\quad\vee\, \exists \texttt{v},\texttt{q} \cdot \texttt{root}{\mapsto}\texttt{node}(\texttt{v},\texttt{q}) * \texttt{lseg}(\texttt{q},\texttt{p},\texttt{n}{-}1)\\&\quad\mathbf{inv}\,\texttt{n} \geq 0;\end{aligned}$$

The pure invariant property $\texttt{n}{\geq}0$ in the predicate declaration indicates that the list's size is always non-negative. Later, the predicate $\texttt{lseg}(\texttt{root},\texttt{p},\texttt{n})$ can be used to specify either null-terminating lists (when $\texttt{p} = \texttt{null}$) or circular lists (when $\texttt{p} = \texttt{root}$) for some examples in the next chapters.

2.3 Forward Verification Rules

The HIP verifier uses a set of standard Hoare-style forward verification rules to systematically check that the implementation of a method does not violate the given specification. The Hoare judgment for each program statement of the core language is formalized in Fig. 2-3 in the form of a triple $\vdash \{\Phi_{pr}\}\, e\, \{\Phi_{po}\}$. In these rules, the primed version of a variable v, *i.e.* v', denotes the latest value of

$$\frac{\vdash \{\Phi \wedge v'\}\, e_1\, \{\Phi_1\} \quad \vdash \{\Phi \wedge \neg v'\}\, e_2\, \{\Phi_2\}}{\vdash \{\Phi\}\, \texttt{if}\ v\ \texttt{then}\ e_1\ \texttt{else}\ e_2\, \{\Phi_1 \vee \Phi_2\}}\ [\textbf{FV-IF}] \qquad \frac{\vdash \{\Phi\}\, e\, \{\Phi_1\}}{\vdash \{\Phi\}\, t\ v; e\, \{\exists v, v' \cdot \Phi_1\}}\ [\textbf{FV-BLK}]$$

$$\frac{\vdash \{\Phi\}\, e\, \{\Phi_1\} \quad \Phi_2 \equiv \exists \texttt{res} \cdot (\Phi_1 \wedge v'{=}\texttt{res})}{\vdash \{\Phi\}\, v := e\, \{\Phi_2\}}\ [\textbf{FV-ASSIGN}] \qquad \frac{\vdash \{\Phi\}\, e_1\, \{\Phi_1\} \quad \vdash \{\Phi_1\}\, e_2\, \{\Phi_2\}}{\vdash \{\Phi\}\, e_1; e_2\, \{\Phi_2\}}\ [\textbf{FV-SEQ}]$$

$$\frac{\Phi_1 \equiv (\Phi \wedge \texttt{res} = v')}{\vdash \{\Phi\}\, v\, \{\Phi_1\}}\ [\textbf{FV-VAR}] \qquad \frac{\Phi_1 \equiv (\Phi \wedge \texttt{res} = k^t)}{\vdash \{\Phi\}\, k^t\, \{\Phi_1\}}\ [\textbf{FV-CONST}] \qquad \frac{\Phi_1 \equiv (\Phi \wedge \texttt{res} = v')}{\vdash \{\Phi\}\, \texttt{return}\ v\, \{\Phi_1\}}\ [\textbf{FV-RET}]$$

$$\frac{t_0\ mn(\overline{t\ v})\ (\Phi_{pr}, \Phi_{po})\ \{e\} \in Prog \quad \Phi \vdash \Phi_{pr} \rightsquigarrow \Phi_1 \quad \Phi_2 \equiv \Phi_1 \wedge \Phi_{po}}{\vdash \{\Phi\}\, mn(\overline{v})\, \{\Phi_2\}}\ [\textbf{FV-CALL}] \qquad \frac{\vdash \{\Phi_{pr}\}\, e\, \{\Phi\} \quad \Phi \vdash \Phi_{po} \rightsquigarrow \Phi_r}{\vdash \{\texttt{true}\}\, t_0\ mn(\overline{t\ v})\ (\Phi_{pr}, \Phi_{po})\ \{e\}\, \{_\}}\ [\textbf{FV-METH}]$$

Figure 2-3. Hoare-style Forward Verification Rules

this variable at a verification point.

Based on the given rules, the verifier can perform an interprocedural verification in a modular fashion with the most important rule [FV−METH] for method declarations. That is, for each method declaration, the verifier initially assumes the given precondition and then sequentially computes the poststate after each program statement inside its body with regard to the respective verification rule in Fig. 2-3. Specifically, at each method call, the verifier checks that the callee's precondition is satisfied before adding its postcondition into the poststate (see the rule [FV−CALL]). Finally, the verifier uses the poststate obtained at the end of the method to prove the declared postcondition.

The HIP/SLEEK verification system currently focuses on proving partial correctness of programs. In this thesis, we shall extend it to reason about program termination and non-termination. A traditional approach for such extension is the Hoare logic for total correctness, which additionally include

the termination proofs into the verification rule of loops. In the next section, we will give an overview about some verification system which are based on this approach.

In contrast to these verifiers, our proposal is to model the program termination and non-termination properties as logical assertions and uniformly integrate both termination and non-termination reasoning into the entailment procedure. As a result, the set of verification rules still remain in their standard-looking form. Moreover, by enhancing the entailment procedure for reasoning about termination and non-termination, we can easily leverage the available verification infrastructures to construct a termination specification inference mechanism for a wider class of programs. Lastly, we improve the performance of the entailment procedure with proof slicing mechanisms as a means to enhance the scalability of the verification system.

Chapter 3

A Resource-Based Logic for Termination and Non-Termination Proofs

In this chapter, we introduce a unified logical framework for specifying and proving *both* termination and non-termination of various programs. Our framework is based on a resource logic which captures both upper *and* lower bounds on resources used by the programs. By an abstraction, we evolve this resource logic for execution length into a temporal logic with three predicates to reason about termination, non-termination or unknown. We introduce a new logical entailment system for temporal constraints and show how Hoare logic can be seamlessly used to prove termination and non-termination in our unified framework. Though our focus is on the formal foundations for a new unified framework, we also report on the usability and practicality of our approach by specifying and verifying both termination and non-termination properties for about 300 programs, collected from a variety of sources. This adds a modest 5-10% verification overhead when compared to underlying partial-correctness

verification system.

3.1 Overview

Termination proving is an important part of correctness proofs for software systems as "so-called *partial correctness* is inadequate: if a program is intended to terminate, that fact must be part of its specification." – Cliff Jones [87]. Thus, *total correctness* proofs, denoted by the Hoare triple $[P]c[Q]$, require the code fragment c to be shown terminating in addition to meeting the postcondition Q after execution. The termination of a loop or a recursive method is usually proven by a well-founded termination measure given to the specification. However, such a measure is not a component of the logical formulas for pre/post specifications. A reason for this distinction is that specification logic typically describes program states, while the termination proofs are concerned with the existence of well-founded measures to bound the execution length of loops/recursions, as argued by Hehner in [73]. Due to this distinction, we cannot automatically leverage richer logics that have been developed for safety properties to conduct more intricate termination and non-termination reasoning.

For illustration, let us use the Shuffle problem proposed in the Java Bytecode Recursive category of the annual Termination Competition [111]. In this problem, an acyclic linked list is shuffled by the `shuffle` method together with the auxiliary `reverse` method, whose source code is shown in Fig. 3-1. To prove that `shuffle` terminates, we need to firstly show that `reverse` also terminates. While the termination of `reverse` can be easily proved by current approaches, such as [21, 32, 102], proving `shuffle` terminates is harder because it requires a functional correctness related fact: the `reverse` method

```
public static List shuffle(List xs) {
  if (xs==null) return null;
  else {
    List next = xs.next;
    return new List(xs.value, shuffle(reverse(next))); }}

public static List reverse(final List l) {
  if (l==null || l.next==null) return l;
  final List nextItem = l.next;
  final List reverseRest = reverse(nextItem);
  l.next = null;   nextItem.next = l;
  return reverseRest; }
```

Figure 3-1. The Shuffle problem from the Termination Competition

does not change the length of the list. Based on this fact, it is possible to show that the linked list's length is also decreasing across the recursive method call `shuffle`; as a result, the method always terminates.

Therefore, without an integration of termination specification into logics for functional correctness, such as separation logic [133], the termination of `shuffle` is hardly specified and proved by verification systems based on the traditional Hoare logic for total correctness. Note that automated termination provers, such as AProVE [65] and COSTA [5], are not able to show that `shuffle` terminates, even after applying a numeric abstraction on the size property to `shuffle` [110], due to the lack of information flow between the correctness and the termination arguments. We believe that relatively complex problems, such as Shuffle, highlight the need of a more expressive logic with the ability of integration into various safety logics for termination reasoning.

Moreover, if the termination proof fails, *e.g.*, when the input list of `shuffle` is cyclic, the program will be *implicitly* assumed to be *possibly* non-terminating. That is, *definite* non-termination is neither *explicitly* stated nor proven by Hoare logic. Explicitly proving non-termination has two

benefits. First, it allows more comprehensive specifications to be developed for better program understanding. Second, it allows a clearer distinction between expected non-termination (*e.g.*, reactive systems where loops are designed to be infinite) and failure of termination proofs, paving the way for focusing on real non-termination bugs that minimize on false positives.

Some specification languages, such as Dafny [106], ACSL [17] and JML [101], allow the specification of *possible* non-termination but their corresponding verifiers provide limited support for this feature. For example, the verifier of Dafny (version 1.8.2) only allows such specification on loops or tail-recursive methods[1], while Frama-C verifier of ACSL [50] has not implemented it. On the other hand, we can use the `false` postcondition, which indicates that the method's exit is unreachable, to specify *definite* non-termination. However, such postcondition for partial correctness is not preferred as it is logically distinct from termination proofs. This distinction has been designed into Dafny, Frama-C and KeY with JML [4], that makes the tools fail to take into account non-terminating behavior when proving termination.

As a case study, Dafny 1.8.2 succeeds in proving the termination of a recursive method[2] though this method contains a call to a non-terminating method.[3] In fact, for termination proofs, these tools simply check that there is a finite number of mutual recursive calls to the analyzed methods, rather than the methods' termination per se. Later, these two problems have been fixed in Dafny 1.9.1, which only allows a non-termination specification in a non-terminating context. However, this check is performed syntactically and

[1] http://www.rise4fun.com/Dafny/PnRX

[2] http://www.rise4fun.com/Dafny/6FuR

[3] The examples in ACSL and JML are at http://loris-7.ddns.comp.nus.edu.sg/~project/hiptnt/others.zip

the tool cannot prove the conditional termination of a method under its terminating cases[4].

Our proposal. We propose integrating both termination and non-termination requirements directly into the specification logic for functional properties. Our work follows Hoare and He [77] and Hehner [72], in which the termination is reasoned together with partial correctness proof. In [72], the program is instrumented with a time variable t and the termination is proven by a finite bound on the *exact* execution time $t'-t$, where t, t' are the initial, resp. final time. In [77], a special ghost variable ok is used to signify termination. However, these approaches presently do not handle non-termination.

As a formal foundation to unify termination and non-termination reasoning and integrate them into functional correctness proofs, we introduce a new resource logic which captures the concept of resource capacity; tracking both *minimum and maximum* amounts of resources used by some given code. Our logic uses a primitive predicate $\mathtt{RC}\langle l, u\rangle$ with invariant $0 \leq l \leq u$ to capture a semantic notion of resource capacity (l, u) with the lower bound l and the upper bound u. Through this resource logic, we can specify a variety of complexity-related properties, including the notions of termination and non-termination, by tracking the number of calls (and loop iterations) executed by the given code. Termination is denoted by the presence of a *finite* upper bound, while non-termination is denoted an *infinite* lower bound on the execution length.

To support a more effective mechanism, we shall derive a simpler *temporal* logic from the richer resource logic itself. We define three temporal predicates, `Term` M, `Loop` and `MayLoop`, where M is a well-founded termination measure,

[4] http://www.rise4fun.com/Dafny/wnF

and associate them with each method in a given program to denote the termination, definite non-termination and possible non-termination of these methods, respectively. In terms of resource reasoning, these predicates represent $\texttt{RC}\langle 0, \texttt{embed}(M)\rangle$, $\texttt{RC}\langle\infty,\infty\rangle$ and $\texttt{RC}\langle 0,\infty\rangle$, respectively, where $\texttt{embed}(M)$ is a finite bound obtained through an order-embedding of M into naturals. Using the enriched specification logic, functional correctness, termination and non-termination of methods can be verified under a single modular framework. With this unification, the predicate $\texttt{Term}\ M$ denotes exactly definite termination, instead of just denoting the bound on the number of loop iterations or method recursions like the termination measures used in the traditional Hoare logic for total correctness.

3.2 From Resource to Temporal Logic

We introduce a general resource predicate $\texttt{RC}\langle l, u\rangle$ where l is a lower bound and u is an upper bound on resource capacity, with invariant $0 \leq l \leq u$. This resource predicate can be specialized to execution capacity to capture a variety of complexity-related properties, via lower and upper bounds on the total number of method calls during the execution of a given piece of code. We shall give an instrumented semantics for this specific resource logic, and also specialize it for reasoning about termination and non-termination. To prove termination, we simply use the predicate $\texttt{RC}\langle 0, u\rangle$ where u is some finite value, namely $u<\infty$. To prove non-termination, we can use the predicate $\texttt{RC}\langle\infty,\infty\rangle$ which signifies an infinite lower bound. Lastly, if we cannot prove either termination or non-termination, we use the predicate $\texttt{RC}\langle 0,\infty\rangle$ which covers all possibilities.

The resource logic we have outlined is quite expressive, and could

pred isEvenNat(int n) ≡ n≥0 ∧ ∃m · n=2∗m;
int sumE (int n)
  requires isEvenNat(n) ∧ Term [n] ∨
        ¬isEvenNat(n) ∧ Loop
  ensures true;
{ if (n==0) return 0;
  else return n + sumE(n−2); }
(a)
	while (x>y)
	  requires
	   x≤y ∧ Term [] ∨
	   x>y ∧ x<0 ∧ Loop ∨
	   x>y ∧ x≥0 ∧ MayLoop
	  ensures x′≤y′;
	{ y=x+y; x=x−1; }
	(b)

Figure 3-2. Examples on numerical programs

moreover be specialized for reasoning on just termination and non-termination with the direct handling of infinity ∞ value. In order to design a simpler logic, we introduce a temporal logic with three distinct predicates, as follows: *(i)* Term M to denote RC$\langle 0, \texttt{embed}(M)\rangle$, *(ii)* Loop to denote RC$\langle\infty,\infty\rangle$ and *(iii)* MayLoop to denote RC$\langle 0,\infty\rangle$. Such a temporal logic is considerably simpler than the more expressive resource logic, since we can omit reasoning with ∞. We can also use a simpler termination measure M, based on depth of recursion rather than number of calls, but relate to the latter using embed(M). Moreover, these temporal predicates can be made flow-insensitive, and thus need only appear in each method's precondition where they describe execution capacity required for the method's execution. This two-level approach simplifies both the design of a formal semantics, and the development of a verification framework for (non-)termination.

For illustration, let us look at some numerical examples, starting with the method sumE in Fig. 3-2(a). This method is required to return the sum of all even natural numbers that are less than or equal to the input n. However, the implementation satisfies this requirement only when n is an even natural number, denoted by the predicate isEvenNat(n); otherwise, the method does not terminate[5]. In our approach, these distinct scenarios can be described in a

[5]The verification system assumes the use of arbitrary precision integers. When finite integers

termination-enriched specification by seamlessly integrating the temporal constraints Term [n] and Loop into a logic with disjunctions.

JML and ACSL also support the specification of several method behaviors. However, the current ACSL implementation in Frama-C does not allow fine-grained termination related specification of each behavior and ignores conditional termination clauses. As a result, it cannot verify all the (non-)terminating behaviors of sumE together. KeY allows the specification of termination for each individual method behavior but it cannot disprove the termination of sumE when n is an odd positive number, because the variant n is still valid under this precondition.[3] In contrast, our unified termination and non-termination reasoning does not accept the temporal constraint Term [n] in these prestates because the execution starting from them will eventually reach a non-terminating execution when n<0. In terms of resource reasoning, Term [n], denoting a finite resource, is invalid as it cannot satisfy the infinite resource required by the non-termination.

The next example in Fig. 3-2(b) illustrates a usage of MayLoop constraint. Starting from any prestate satisfying $x>y \wedge x\geq 0$, the execution of the given loop may reach either the base case (when $x\leq y$, indicated by Term []) or the non-terminating case (when $x>y \wedge x<0$, indicated by Loop). We observe that this MayLoop precondition can be strengthened to the *non-linear* constraint $4x^2+4x+8y+9\geq 0$ for non-termination, but this requires stronger arithmetic solvers.

Though our proposal is independent of the underlying logics on functional properties, it can leverage infrastructures of richer logics[6] to conduct termination and non-termination reasoning for more complex domains. For

are used, we may give a different temporal specification for those prestates.

[6] In comparison with the first-order logic with linear arithmetic for numerical programs.

```
data node { node next; }

pred lseg(root, p, n) ≡ root=p ∧ n=0
  ∨ ∃q · root↦node(q)∗lseg(q, p, n−1)
  inv n ≥ 0;

List reverse (List l)
  requires lseg(l, null, n) ∧ Term [n]
  ensures lseg(res, null, n);
List shuffle (List xs)
  requires lseg(xs, null, n) ∧ Term [n]
  ensures lseg(res, null, n);
```

Figure 3-3. A specification in separation logic to verify the correctness of Shuffle's methods

example, our proposed temporal constraints are easily integrated into formulas of separation logic to reason about the termination and non-termination of heap-based programs.

For the Shuffle problem, we can use the predicate `lseg` denoting linked list segments in the pre and postconditions of its two methods `reverse` and `shuffle` to describe their input-output relations, as shown in Fig. 3-3. The specification of each method indicates that the method's result `res` is a linked list with the same size `n` as the input list. From these safety specifications, the temporal constraint `Term` [n] integrated into the precondition of each method is able to specify that the depth of recursion is bounded by the size of the input list, thus indicating the method's termination.

From the perspective of resource reasoning, a temporal constraint in the precondition of a method defines the bounds of available resource allowed for program executions from prestates satisfying (safety part of) this precondition. This idea is similar to Atkey's logic [12], a type-based amortized resource analysis for imperative programs, which associates a piece of resource with each element of the data structures prior program execution. However, Atkey's approach only tracks the upper bound of resource usage, so that it cannot

reason about non-termination. This shortcoming also applies to other type-based approaches for termination reasoning, such as [3, 148]. In addition, while the amortized resource analysis accounts for individual time-step (or heap chunk), we use termination measures, which are much simpler, to facilitate termination proofs. For example, to analyze `shuffle`, Atkey's logic requires the global length property to present the polynomial resource associated with the input list using the technique of Hoffmann and Hofmann [81], which is much harder than locally reasoning about each node of the list as stated in his paper. Finally, this logic is built on top of just separation logic, rather than being generic as our proposal.

3.3 A Logic for Resource Reasoning

In proving termination and non-termination, our goal is to use resource reasoning based on execution capacity to provide a means for quantitatively assessing the execution length of a program. For this purpose, we introduce a resource logic to formally assess the minimum and a maximum bounds on a program's resource consumption. We first extend the program state model with a mechanism to track resource capacities of the underlying machine. Since the particular consumed resource is countable and possibly infinite, we use the set $\mathbb{N}^\infty$, short for $\mathbb{N} \cup \{\infty\}$, as its domain.

3.3.1 Resource Capacity

Definition 1 (Program states) *A program state σ is a triple (s, h, r) of stack $s \in \mathcal{S}$ (locals), heap $h \in \mathcal{H}$ (memory) and $r \in \mathcal{R}$, resource capacity where r is a pair (r_l, r_u) of bounds in $\mathbb{N}^\infty$, with $0 \leq r_l \leq r_u$, denoting the allowed minimum and maximum resource consumption for executions starting from the current*

program state.

Intuitively, a program state's resource capacity (r_l, r_u) ensures that any execution starting from this state must consume *at least* r_l and *at most* r_u of the tracked resource.

Definition 2 (Resource Capacity Ordering) *Let* $(\leq_c) \subset \mathbb{N}^\infty \times \mathbb{N}^\infty$ *be the resource capacity ordering, such that* $(b_l, b_u) \leq_c (a_l, a_u)$ *iff* $a_l \leq b_l$ *and* $b_u \leq a_u$*.*

The resource capacity (a_l, a_u) is considered larger (or more general) than (b_l, b_u) if $a_l \leq b_l$ and $b_u \leq a_u$. The intuition is that under this condition, any execution which guarantees the capacity (b_l, b_u) also guarantees the capacity (a_l, a_u). Based on this observation, $(0, \infty)$ is the largest resource capacity. In fact, it indicates an unconstrained resource consumption.

In order to properly define an operational semantics in terms of the proposed program state model, we also need to be able to express resource consumption. To this end we define a splitting operation over the resource capacity. We will say that a capacity (a_l, a_u) can be split into capacities (b_l, b_u) and (c_l, c_u), written $(a_l, a_u) \ominus (b_l, b_u) = (c_l, c_u)$, if whenever an execution that guarantees the capacity (b_l, b_u) starts from a state with the capacity (a_l, a_u) then the remaining capacity is (c_l, c_u). In other words, the executions allowed by (a_l, a_u) can be decomposed into executions required by (b_l, b_u) followed by executions required by (c_l, c_u).

Definition 3 (Resource Capacity Splitting) *Given resource capacities* (a_l, a_u), (b_l, b_u) *with* $b_u \leq a_u$ *and* $a_l + b_u \leq a_u + b_l$ *then* $(a_l, a_u) \ominus (b_l, b_u) = (c_l, c_u)$ *where*

$$c_l = \mathtt{min}\{x_l \in \mathbb{N}^\infty \mid x_l + b_l \geq a_l\} \text{ and } c_u = \mathtt{max}\{x_u \in \mathbb{N}^\infty \mid x_u + b_u \leq a_u\}.$$

$$
\begin{array}{lcl}
(s,h,r) \models \Psi_1 \vee \Psi_2 & \equiv & (s,h,r) \models \Psi_1 \text{ or } (s,h,r) \models \Psi_2 \\
(s,h,r) \models \Psi_1 \wedge \Psi_2 & \equiv & (s,h,r) \models \Psi_1 \text{ and } (s,h,r) \models \Psi_2 \\
(s,h,r) \models \exists \overline{x_i} \cdot \Psi & \equiv & \exists \nu_i^* \cdot (s[(x_i \mapsto \nu_i)], h, r) \models \Psi \\
(s,h,r) \models \rho & \equiv & (s,h) \models \rho \\
(s,h,r) \models \mathtt{RC}\langle a_l, a_u \rangle & \equiv & (s,h) \models r_l = a_l \wedge r_u = a_u \quad \text{where } r{=}(r_l, r_u) \\
(s,h,r) \models \theta_1 \blacktriangleright \theta_2 & \equiv & \forall r' \cdot \text{if } (s,h,r') \models \theta_1 \text{ then } (s,h,r \ominus r') \models \theta_2
\end{array}
$$

Figure 3-5. Semantics of Assertions in the Resource-Aware Logic

From Defn. 3, (c_l, c_u) is the *largest* resource consumption allowed for any execution following executions satisfying (b_l, b_u) such that the overall resource consumption is described by (a_l, a_u). Under this interpretation it follows naturally that when $b_u{>}a_u$ the splitting operation is undefined as c_u does not exist. In addition, when $a_l{+}b_u{>}a_u{+}b_l$, the splitting operation is also undefined as it would lead to $c_l{>}c_u$.

3.3.2 Assertion Language and Semantics for a Resource-Aware Logic

To support resource reasoning, we extend a minimalistic assertion language with two resource assertions θ, as shown in Fig. 3-4. Recall that we use v and $\overline{v}$ for denoting variables and sequences of variables, $f(\overline{v})$ for functions from variables to $\mathbb{N}^\infty$. The resource-free formulas is represented by ρ and Φ. For example, in the HIP/SLEEK system, the formula ρ comprises two parts: the heap formula κ and the pure formula π, as shown in Fig. 2-2.

The resource assertion θ ranges over *(i) atomic* resource assertions $\mathtt{RC}\langle a_l, a_u\rangle$, where a_l, a_u are functions from variables to $\mathbb{N}^\infty$; and *(ii) splitting* resource assertions $\theta_1 \blacktriangleright \theta_2$, which holds for states that allow executions to be split into two execution fragments, on which ρ_1 and ρ_2 hold respectively.

$$\begin{array}{lll}\Psi & ::= & \bigvee(\exists\overline{v}\cdot\rho\wedge\theta)\\ \Phi & ::= & \bigvee(\exists\overline{v}\cdot\rho)\\ \theta & ::= & \texttt{RC}\langle a_l, a_u\rangle \mid \theta_1 \blacktriangleright \theta_2\\ a & ::= & f(\overline{v})\end{array}$$

Figure 3-4. The Assertion Language

We concisely list in Fig. 3-5 the semantic model for the assertion language. We observe that the usual semantics of the logical connectives, *e.g.*, conjunctions and disjunctions, lifts naturally over resource assertions. The semantics of the resource-free assertions is straightforward: a resource-free formula ρ holds for all states (s, h, r) such that $(s, h) \models \rho$ with respect to the semantics of the corresponding underlying logic. Note that the operator $\models$ is overloaded for both the underlying logic and the resource-aware logic.

We point out that we have chosen to model the $\texttt{RC}\langle a_l, a_u\rangle$ assertion as a precise predicate. That is, a program state σ satisfies a resource constraint θ if the resource capacity in σ is equal to the evaluation, in the context of σ, of the upper and lower functions associated with θ. This modeling relation ensures that the resource assertion θ is *precise* with regards to the resource capacity, where $(s, h, r) \models \theta$ does not imply $(s, h, r') \models \theta$ whenever r' is larger than r, *i.e.*, $r' \geq_c r$. Consequently, $\texttt{RC}\langle a_l, a_u\rangle \vdash \texttt{RC}\langle b_l, b_u\rangle$ iff $(s, h) \models a_l{=}b_l \wedge a_u{=}b_u$. Additionally, $\texttt{RC}\langle a_l, a_u\rangle \wedge \texttt{RC}\langle b_l, b_u\rangle \equiv \texttt{RC}\langle a_l, a_u\rangle$ iff $a_l{=}b_l \wedge a_u{=}b_u$; otherwise, $\texttt{RC}\langle a_l, a_u\rangle \wedge \texttt{RC}\langle b_l, b_u\rangle \equiv \texttt{false}$.

To provide a precise modular resource reasoning, we lift the semantic split operation into a resource splitting assertion $\theta_1 \blacktriangleright \theta_2$. This enables our proof construction to follow the same style of other resource manipulating logics, such as separation logic. The intuition behind the splitting resource assertions is that $\theta_1 \blacktriangleright \theta_2$ holds for any program state from which it is possible to consume as many resources as θ_1 requires and end in a state that satisfies ρ_2. Or equivalently, $\theta_1 \blacktriangleright \theta_2$ holds for all states whose resource capacity can be

split into two portions, such that the resulting capacities satisfy θ_1 and θ_2, respectively. In addition, we can use $\blacktriangleright$ to add a resource capacity θ_1 into the current available resource capacity θ, resulting in $\theta \blacktriangleright \theta_1$. The semantics of $\theta_1 \blacktriangleright \theta_2$ is also given in Fig. 3-5.

3.3.3 Resource-Enhanced Entailment with Frame Inference

Based on the semantics of resource assertions and the standard definition of the logical entailment relation (*i.e.*, $\Psi_1 \vdash \Psi_2$ iff $\forall \sigma \cdot$ if $\sigma \models \Psi_1$ then $\sigma \models \Psi_2$), it is possible to define an entailment for resource constraints of the form $\theta \vdash \theta_1 \blacktriangleright \theta_2$ as follows:

Lemma 1 (Resource Entailments) *Given resource assertions θ, θ_1 and θ_2, $\theta \vdash \theta_1 \blacktriangleright \theta_2$ iff $\forall s, h, r, r_1 \cdot$ if $(s, h, r) \models \theta$ and $(s, h, r_1) \models \theta_1$ then $(s, h, r \ominus r_1) \models \theta_2$.*

Proof. The proofs of all lemmas in this chapter can be found in Appendix A.1.

It follows that given $\ominus_f$, a lifting of resource capacity splitting to functions, then:

$$\frac{(\theta_l^2, \theta_u^2) = (\theta_l, \theta_u) \ominus_f (\theta_l^1, \theta_u^1)}{\mathtt{RC}\langle \theta_l, \theta_u \rangle \vdash \mathtt{RC}\langle \theta_l^1, \theta_u^1 \rangle \blacktriangleright \mathtt{RC}\langle \theta_l^2, \theta_u^2 \rangle}$$

Entailments of the form $\theta \vdash \theta_1 \blacktriangleright \theta_2$ are of particular interest in the context of program verification as they naturally encode the restriction imposed at a method call and the remaining restriction after the execution of this method. For the proposed resource logic, we construct a general entailment system with frame inference by merging the entailment of resource constraints presented earlier with the entailment system corresponding to the underlying logic. Let the underlying entailment system be of the general form $\Psi \vdash \Phi \leadsto \Phi_r$ denoting that Ψ implies Φ with frame Φ_r. In sub-structural logics such as separation logic, the frame captures any residual state that is not required by the entailment. In

pure logics where the program states are not changed, the frame is simply the antecedent of the entailment.

To support logics with disjunctions, the entailment system firstly deconstructs disjunctive antecedents (*e.g.*, using the rule [ENT−DISJ−LHS]) and consequents until formulas of the form $\rho \wedge \theta$ with a single resource constraint[7] are encountered in both sides of the sub-entailments. The judgment system then applies the rule [ENT−CONJ] that is slightly changed to handle resource constraints by splitting an entailment into two parts, namely *logical* part and *resource* part. The logical goal is solved by the entailment system $\mu_a \vdash \mu_c \leadsto \Phi_r$ of the underlying logic. The resource goal is solved by using the resource entailment rules presented above. The solving process for the resource part leverages the entailment outcome Φ_r from the underlying logic, which is simply added to the antecedent of the resource entailment, to check the condition stated in Defn. 3 for the resource capacity splitting operation to be defined.

[ENT−DISJ−LHS]

$$\frac{\Psi \;=\; \bigvee \exists \overline{v_i} \cdot (\rho_i \wedge \theta_i) \qquad \forall i \cdot\ (\rho_i \wedge \theta_i) \vdash \Phi \leadsto \Psi_r^i}{\Psi \vdash \Phi \leadsto \bigvee \exists \overline{v_i} \cdot \Psi_r^i}$$

[ENT−CONJ]

$$\frac{\rho_a \vdash \rho_c \leadsto \rho_r \qquad \rho_r \wedge \theta_a \vdash \theta_c \blacktriangleright \theta_r}{\rho_a \wedge \theta_a \vdash \rho_c \wedge \theta_c \leadsto (\rho_r \wedge \theta_r)}$$

3.3.4 Hoare Logic for Resource Verification

Language. We use the core strict imperative language of HIP/SLEEK system (see Fig. 2-1). For simplicity, this core language does not support while-loop constructs. A preprocessing step shall be applied to automatically translate while-loops into tail-recursive methods with reference-type parameters.

[7] A conjunction of resource constraints can be simplified to either a single resource constraint or `false` as discussed in Sec. 3.3.2.

The pre and post conditions of a method are specified by the `requires` and `ensures` keywords, followed by logic formulas in the assertion language in Fig. 3-4. Resource-related assertions always appear in the method preconditions to denote resource requirements imposed on the caller for its execution. In contrast, resource assertions in the postconditions denote unspent/generated fuel returned to the caller, so that these assertions may not appear in the postconditions, depending on the analyzed resource. For example, as execution length (*i.e.*, a temporal resource) can only be consumed, it is safe and convenient to assume that the method consumes all the initially required resource; thus we can avoid the need for execution length related assertions in postconditions.

Hoare Logic. We observe that the resource consumption of each program statement is dependent on the tracked resource. As a result, the resource-aware Hoare logic needs to be adapted accordingly for each resource type. In terms of termination and non-termination reasoning, we are interested in the execution length as the tracked resource capacity. In the next section, we will construct a specific Hoare logic to reason about this resource.

3.4 (Non-)Termination Proofs via Resource Reasoning

For termination and non-termination reasoning, we have proposed three temporal constraints to capture: guaranteed termination `Term` X, guaranteed non-termination `Loop` and possible non-termination `MayLoop`, where X is a ranking function built from program variables. First, we define these constraints as resource capacity assertions, using the more general `RC`

predicate. Next, we leverage the resource logic in Sec. 3.3, specialized in execution capacity, to construct a logic for termination and non-termination reasoning. A resource-based definition for the proposed temporal constraints is as follows:

Definition 4 (Temporal Constraints) *Temporal constraints are resource assertions over program execution lengths, such that* $\mathtt{Term}\ X \equiv \mathtt{RC}\langle 0_f, \varpi\rangle$, $\mathtt{Loop} \equiv \mathtt{RC}\langle \infty_f, \infty_f\rangle$ *and* $\mathtt{MayLoop} \equiv \mathtt{RC}\langle 0_f, \infty_f\rangle$ *where* 0_f *and* ∞_f *denote the constant functions always returning* 0 *respectively* ∞. ϖ *is a function of program variables to naturals, imposing a finite upper bound on the execution length of a terminating program.*

Using the definition of resource entailments in Lemma 1, we formalize the set of valid entailments for temporal constraints below:

$$\begin{array}{ll}
\mathtt{MayLoop} \vdash \mathtt{MayLoop} \blacktriangleright \mathtt{MayLoop} & \mathtt{Loop} \vdash \mathtt{Term}\ X \blacktriangleright \mathtt{Loop} \\
\mathtt{MayLoop} \vdash \mathtt{Term}\ X \blacktriangleright \mathtt{MayLoop} & \mathtt{Loop} \vdash \mathtt{Loop} \blacktriangleright \mathtt{MayLoop} \\
\mathtt{MayLoop} \vdash \mathtt{Loop} \blacktriangleright \mathtt{MayLoop} & \dfrac{\mu \Rightarrow Y \leq_d X}{\mu \wedge \mathtt{Term}\ X \vdash \mathtt{Term}\ Y \blacktriangleright \mathtt{Term}\ X -_d Y} \\
\mathtt{Loop} \vdash \mathtt{MayLoop} \blacktriangleright \mathtt{Loop} &
\end{array}$$

where $\leq_d$ and $-_d$ are the ordering and the subtraction operation on the domain of the termination measures X and Y, respectively. All other decomposition attempts, such as $\mathtt{Term}\ X \vdash \mathtt{MayLoop} \blacktriangleright _$ and $\mathtt{Term}\ X \vdash \mathtt{Loop} \blacktriangleright _$, describe unfeasible splits. Thus in those cases, the entailment fails and an error is signaled.

3.4.1 From Termination Measures to Execution Capacity's Finite Upper Bounds

In Defn. 4, as X denotes a termination measure, a bounded function that decreases across recursive method calls, the resource upper bound ϖ must also follow. Thus, the mapping function from X to ϖ must be an *order-embedding* denoted by $\texttt{embed}(X)$. In our approach, the termination measure X is a (finite) list of arithmetic formulas over naturals $[\overline{e}]$ whose order is based on the lexicographic ordering $<_l$ as defined below.

Definition 5 (Lexicographic Ordering)

$$\frac{}{[] <_l e{:}_} \qquad \frac{(e_1 < e_2) \vee (e_1 = e_2 \wedge es_1 <_l es_2)}{e_1{:}es_1 <_l e_2{:}es_2}$$

where $e{:}es$ denotes a non-empty list with e and es as its head and tail, respectively.

We choose this formulation since it is simpler to write than a single but more complex termination measure and it can be used for a wider range of programs. In general, an order-embedding of lists of unbounded elements requires *ordinals*. However, transfinite ordinals are not suitable to model finite computational resources denoted by $\texttt{Term}\ X$.

By a *co-inductive* argument that every execution of a terminating method only computes finitely many different values, it follows that every non-negative element of a lexicographic termination measure applied to states of the corresponding call tree is upper-bounded. We then show that there always exists an order-embedding $\mathcal{L}$ from the codomain of a termination measure (*i.e.*, tuples of bounded naturals) to naturals, such that $\texttt{embed}(X) = \mathcal{L} \circ X$.

Lemma 2 *If the termination of a program can be proven by a given lexicographic termination measure, then for each call tree τ of the program, every element of the termination measure applied to the program states corresponding to the nodes in the call tree τ is bounded.*

If every element x_i, where $0 \leq i \leq n-1$, of a lexicographic termination measure $[x_n, x_{n-1}, \ldots, x_0]$ corresponding to a given call tree τ is bounded by a constant k, we can use the base $b{=}k{+}1$ to construct a possible order-embedding function $\mathcal{D}([x_n, x_{n-1}, \ldots, x_0]) = x_n * b^n + x_{n-1} * b^{n-1} + \ldots + x_0$. The function $\mathcal{D}$ preserves the order of the given measure along every trace of τ, as stated by Lemma 3.

Lemma 3 *For all $x_n, \ldots, x_0, y_n, \ldots, y_0 \in \mathbb{N}$ such that $\forall i \in \{0..n-1\} \cdot x_i, y_i < b$, $[x_n, \ldots, x_0] >_l [y_n, \ldots, y_0]$ iff $\mathcal{D}([x_n, \ldots, x_0]) > \mathcal{D}([y_n, \ldots, y_0])$, where $>_l$ is the lexicographic ordering.*

In general, such a bounded constant k for a call tree τ can be determined by a function $\mathcal{K}$ of initial values of the call tree's variables. Since the execution of a loop has only a single trace, the order-embedding $\mathcal{D}$, constructed from the constant k, would be enough to ensure the sufficiency of execution capacity for the loop. However, in order to give a proper estimate of the execution capacity for more complex recursion patterns, especially when the termination measures are based on the depth of recursion, we propose using a more refined embedding for a call tree, that is $\mathcal{L} = \begin{cases} \mathcal{D} & ,\mathcal{N} \leq 1 \\ \mathcal{N}^{\mathcal{D}} & ,\mathcal{N} > 1 \end{cases}$, where $\mathcal{N}$ is the maximum number of children for each node of the call tree.

Therefore, given the termination measure X of a terminating program, there always exists an order-embedding $\mathcal{L}$ from the codomain of X to naturals. The function $\mathcal{L}$ can be constructed from initial values of program variables and the

$$\frac{\mathsf{CheckMin}(\Psi_1) \quad \mathsf{CheckMin}(\Psi_2)}{\mathsf{CheckMin}(\Psi_1 \vee \Psi_2)} \qquad \frac{\rho \vdash \rho_l = 0}{\mathsf{CheckMin}(\rho \wedge \mathtt{RC}\langle \rho_l, \rho_u \rangle)}$$

$$\boxed{\textbf{FV-CALL}}$$

$$\frac{t_0\ mn(\overline{t\ v})\ (\Psi_{\text{Pre}}, \Phi_{\text{Post}})\ \{code\} \in Prog \qquad \Psi \vdash \mathtt{RC}\langle 1, 1 \rangle \rightsquigarrow \Theta \quad \Theta \vdash \Psi_{\text{Pre}} \rightsquigarrow \Phi \quad \Psi_r = \Phi \wedge \Phi_{\text{Post}}}{\vdash \{\Psi\}\ mn(v^*)\ \{\Psi_r\}}$$

$$\boxed{\textbf{FV-RET}}$$

$$\frac{\mathsf{CheckMin}(\Psi)}{\vdash \{\Psi\}\ \mathtt{return}\ v\ \{\Psi \wedge \mathsf{res} = v'\}}$$

Figure 3-6. Hoare Verification Rules: Method Call and Return

call trees corresponding to these initial values. As a result, $\mathtt{embed}(X) = \mathcal{L} \circ X$ is a function from program variables to naturals, which describes an upper bound on the number of method calls taken by any execution of the program.

3.4.2 Termination and Non-Termination Verification

Here we elaborate on the construction of both termination and non-termination proofs based on Defn. 4 and the verification framework in Fig. 3-6 for tracking execution length as resource. Although execution length can be tracked at various levels of granularities, we choose to track it only at method calls (*i.e.*, as the total number of method calls) in order to simplify the verification rules and the operational semantics. In Fig. 3-6, we only outline the Hoare logic rules for the method call and the return statements, which are especially relevant to the verification of execution lengths as they encode the resource consumption. The Hoare rules for other constructs are standard because they do not interact with the resource of interest.

As a standard preprocessing step, we check that all predicate invariants are

satisfied, including the invariants of resource constraints: the resource assertion $\mathtt{RC}\langle\rho_l, \rho_u\rangle$ in precondition Ψ_{Pre} is consistent if $0{\leq}\rho_l{\leq}\rho_u$, that is, for each disjunct $\mu \wedge \mathtt{RC}\langle\rho_l, \rho_u\rangle$ of Ψ_{Pre} it follows that $\mu \vdash \rho_u{\geq}\rho_l \wedge \rho_l{\geq}0$. We observe that the invariant check on $\mathtt{Term}\ X$ requires that every element of X be non-negative to ensure a non-negative upper-bound $\mathcal{L}{\circ}X$, so that the execution capacity satisfies the invariant $0{\leq}0_f{\leq}\mathcal{L}{\circ}X$.

In the method call rule [FV−CALL], the available execution capacity is first decreased by one step, denoted by $\mathtt{RC}\langle 1, 1\rangle$, to account the cost of method call, followed by a check that the callee's requirements are met. This check is translated into an entailment for proving the method precondition. Finally, the poststate after this method call is computed. With the help of the resource-enhanced entailment system introduced in Sec. 3.3.3, both logical and resource proving are combined into one entailment, resulting in a standard-looking Hoare rule for method call.

In addition, specifically for temporal constraints, two entailments $\Psi \vdash \mathtt{RC}\langle 1, 1\rangle \rightsquigarrow \Theta$ and $\Theta \vdash \Psi_{\text{Pre}} \rightsquigarrow \Phi$ can be combined into $\Psi \vdash \Psi_{\text{Pre}} \rightsquigarrow \Phi$ by using a new entailment $\vdash_t$ for temporal constraints.

Definition 6 (Unit Reduction Temporal Entailments) *Given temporal constraints θ, θ_1 and θ_2, $\theta \vdash_t \theta_1 \blacktriangleright \theta_2$ iff $\forall s, h, r \cdot$ if $(s, h, r) \models \theta$ then $(s, h, r \ominus (1, 1)) \models \theta_1 \blacktriangleright \theta_2$.*

Therefore, if θ is $\mathtt{Loop}$ or $\mathtt{MayLoop}$ then $\theta \vdash_t \theta_1 \blacktriangleright \theta_2$ iff $\theta \vdash \theta_1 \blacktriangleright \theta_2$. If θ is $\mathtt{Term}\ X$ then $\mu \wedge \mathtt{Term}\ X \vdash_t \mathtt{Term}\ Y \blacktriangleright \mathtt{Term}\ ((X{-_d}1_d){-_d}Y)$ if $\mu{\Rightarrow}Y{<_d}X$, where 1_d is the unit of termination measures' domain. Basically, the check $Y{<_d}X$ is equivalent to the check that termination measures are decreasing across recursive method calls in the traditional termination proof. By introducing the temporal entailment $\vdash_t$, we obtain a resource-based temporal

logic which is related to only the temporal constraints and thus the underlying resource reasoning becomes implicit.

In the method return rule [FV−RET], the CheckMin predicate, which is also defined in Fig. 3-6, ensures that the specified minimum computation resource has been completely consumed when the method returns. Note that if the method does not terminate, the minimum guaranteed execution length is always satisfied since the actual return point is never reached. For temporal constraints, CheckMin holds for any `Term` X and `MayLoop` as the lower bounds in their execution capacities are always 0. In non-termination cases, $\mathsf{CheckMin}(\mu \wedge \texttt{Loop})$ only holds when μ is unsatisfiable. This check ensures that a return statement cannot be executed/reachable from a state satisfying `Loop`.

We now state the soundness of this resource-aware Hoare logic as follows:

Theorem 1 *The standard Hoare rules (*e.g.*, assignment, conditional, sequential composition) and the Hoare rules for method call and return are sound.*

Proof. The proof can be found in Appendix A.2.

3.4.3 Flow-Insensitive Temporal Logic

Observe that the current formulation of the temporal logic with temporal constraints is *flow-sensitive* since the entailment $\theta \vdash_t \theta_1 \blacktriangleright \theta_2$ might return a residue θ_2 distinct from θ. However, with the following observations, we can formalize a *flow-insensitive* version of the temporal logic and provide a further abstraction on the resource-based framework presented so far.

First, it is possible to refine the granularity of the termination and non-termination verification by tracking only execution lengths of (mutually) recursive method calls. Second, using König's lemma [91], it is sufficient to

inspect individual execution traces in the call tree for deciding just termination or non-termination, instead of tracking the total execution length of all traces in the call tree. That is, a program terminates iff every execution trace is finite; otherwise, the program is non-terminating.

Based on these observations, the tracked resource will be abstracted to capture the execution capacity required for the longest trace in the call tree, instead of the execution capacity required for the remaining program. With this, the resource (for the longest trace allowed) remains unchanged after each splitting operation, which determines the residue resource needed for subsequent method calls. Thus, for every method, we endeavor to provide a single abstract resource that is sufficient for executing a given method call and also its remaining code sequences.

By using this abstraction, we can obtain a formulation on temporal entailment that ensures $\theta \vdash_t \theta_1 \blacktriangleright \theta$ whereby the temporal constraint in residue is always identical to the one in the antecedent. Hence, the operator $-_d$ can be fully circumvented. Moreover, the finite upper bound ϖ used for the definition of `Term` X in Defn. 4 can be determined as $\varpi = \mathcal{D} \circ X$, instead of the larger $\mathcal{L} \circ X$. As a result, without any change to the Hoare rules, during a method's verification, the same initial resource capacity is used for the verification of call traces and thus facilitating a simpler verification procedure for temporal constraint. As a direct outcome of this abstraction, the temporal assertions `Loop`, `MayLoop` and `Term` X are now *flow-insensitive*, and therefore closer to the pure logic form, as opposed to the sub-structural form of resource logics. Note that flow-insensitive label applies to only the temporal constraints. In general, program states (*e.g.*, denoted by separation logic as the underlying logic) remain flow-sensitive since they might be changed due to changes on

Benchmarks	Programs	Term	Loop	MayLoop	PC(s)	TC(s)	Overhead (%)
Invel	59	137	81	12	14.88	15.96	6.77
AProVE	124	534	120	8	15.73	17.21	8.60
Pasta	44	219	10	3	4.95	5.79	14.51
Others	48	194	32	22	7.35	8.78	16.29
Totals/(%)	**275**	**1084** (79.0%)	**243** (17.7%)	**45** (3.3%)	**42.91**	**47.74**	10.12%

Figure 3-7. Termination Verification for Numerical Programs

heap state and program variables.

3.5 Experiments

We have implemented the proposed termination and non-termination reasoning into the automated verification system HIP/SLEEK and named the prototype HIPTNT, which is available at

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

The integration of the termination logic into an existing system allows us to utilize the infrastructure that has been developed for some richer specification logics, such as separation logic, beyond a simple first-order logic. Consequently, we are able to *specify* and *verify* both termination and non-termination properties, in addition to correctness properties for a much wider class of programs, including heap-manipulating programs. In this system, the final proof obligations are automatically discharged by off-the-shelf provers, such as Z3 [52]. The expressivity of our new integrated logic is shown in the following experimental results, in which the lexicographic order is needed for about 25% of our experimental programs.

3.5.1 Numerical Programs

The verification system was evaluated using a benchmark of over 200 small numerical programs selected from a variety of sources: *(i)* from the literature,

such as [32, 43], *(ii)* from benchmarks used by other systems (that are AProVE [65], Invel [146] and Pasta [58]) and *(iii)* some realistic programs, such as the Microsoft Zune's clock driver that has a leap-year non-termination bug. Most of the methods in these benchmark programs contain either terminating or non-terminating code fragments, expressed in (mutual) recursive calls or (nested) loops. To construct these benchmarks we added the novel termination specifications to the original examples from the analysis tools for termination and non-termination. We have chosen these benchmarks in order to show the usability and practicality of our approach. A comparison with these tools at this time would be of less relevance as our proposal in this chapter focuses on verifying the given specifications rather than infer them.

Fig. 3-7 summarizes the characteristics and the verification times for a benchmark of numerical programs. Columns 3-5 describe the number of preconditions that have been specified and successfully verified as *terminating*, *non-terminating* or *unknown*, respectively. As hoped for, the number of preconditions annotated by `MayLoop` occupies the smallest fragment (about 3%) of the total number of preconditions. Such `MayLoop` constraints were only used in some unavoidable scenarios as discussed in Sec. 3.2. In contrast, the `Term` constraints (with the given measures) are in the majority because most of the methods are expected to be terminating, except for the Invel benchmark which focuses on mostly non-terminating programs.

Our verification system can perform both correctness and termination proofs. Column 7 (**TC**) gives the total timings (in seconds) needed to perform both termination and correctness proofs for all the programs in each row, while column 6 (**PC**) gives the timings needed for just correctness proofs. The difference in the two timings represents the small overheads needed for

Programs	LOC	Proc.	Term	Loop	MayLoop	PC(s)	TC(s)	Overhead (%)
AVL	390	13	18	0	0	13.89	14.66	5.25
Linked List (LL)	135	13	13	0	0	0.28	0.29	3.45
Sorted LL	480	13	15	0	0	1.33	1.38	3.62
Circular LL	80	4	4	4	0	1.04	1.18	11.86
Doubly LL	174	11	12	0	0	0.41	0.46	10.87
Complete	112	6	7	0	0	2.58	3.53	26.91
Heap Tree	214	5	6	0	0	14.82	15.12	1.98
BST	165	6	6	0	0	0.93	1.04	10.58
Perfect Tree	83	5	5	1	0	0.32	0.33	3.03
Red-Black Tree	556	19	25	0	0	6.22	6.40	2.81
BigNat	235	18	18	0	0	15.13	15.42	1.88
Totals/ (%)	**2624**	**114**	**129** (96.3%)	**5** (3.7%)	**0**	**56.95**	**59.81**	4.78%

Figure 3-8. Termination Verification for Heap-manipulating Programs

termination and non-termination reasoning.

3.5.2 Heap-manipulating Programs

As illustrated in Fig. 3-8, we have also conducted termination reasoning on our own benchmark of heap-based programs using various data structures with a small overhead. The modular structure of the resource reasoning framework in Sec. 3.3 facilitates the embedding of temporal constraints into a richer specification mechanism based on separation logic, automatically extending it to proving termination or non-termination properties over heap-manipulating programs. The temporal entailment judgment in the $\lfloor\text{ENT-CONJ}\rfloor$ rule can leverage the power of the separation logic entailment engine to discharge the temporal constraints in a heap-related entailment.

For example, consider the following entailment, which might be encountered when verifying a method call with heap arguments (*e.g.*, the length method of linked lists),

$$\texttt{ll}(\texttt{x},\texttt{n}) \wedge \texttt{x}{\neq}\texttt{null} \wedge \texttt{Term}\,[\texttt{n}] \vdash \texttt{x}{\mapsto}\texttt{node}(_,\texttt{y}){*}\texttt{ll}(\texttt{y},\texttt{n}_1){\wedge}\texttt{Term}\,[\texttt{n}_1]$$

The entailment prover for separation logic can infer the constraint $\texttt{n}_1{=}\texttt{n}{-}1$ (*e.g.*, by the unfolding mechanism and explicit instantiation mechanism introduced

in [116]) when checking the spatial part of the entailment, which is a necessary condition to ensure the validity of the eventual temporal entailment judgment

$$\mathtt{x}{\neq}\mathtt{null} \wedge \mathtt{n_1}{=}\mathtt{n}{-}1 \wedge \mathtt{Term}\,[\mathtt{n}] \vdash_t \mathtt{Term}\,[\mathtt{n_1}] \blacktriangleright \mathtt{Term}\,[0]$$

Due to the tight integration with the underlying logic, this task of specifying and verifying the termination properties was easy even though some of the programs use non-trivial data structures (*e.g.*, Red-Black and AVL-trees), or non-linear constraints (*e.g.*, the BigNat program, which implements infinite precision natural numbers (by linked lists) with procedures for some arithmetic operations, in addition to a fast multiplication method based on the Karatsuba algorithm).

We have successfully determined that none of the above methods have any unknown termination behaviors. All the methods were terminating, except for some methods in circular list and perfect tree. In the case of the latter, a method to create a perfect tree would go into an infinite loop if a negative number was given as its height. Furthermore, during the verification of termination properties, we discovered a bug in our own merge method (for two AVL trees) that went into a loop due to wrong parameter order. The partial correctness proof did not detect this problem. It was later corrected into a terminating method, a courtesy of our newly integrated feature.

In addition, the termination verification has also been done on some medium programs taken from the SIR/Siemens test suite [55] and selective problems from the Termination Competition [111] as shown in Fig. 3-9. Beside heap data structures, some programs in this benchmark also use arrays in their implementation (*e.g.*, *tcas* and *replace* programs), thus requiring the proposed logic to be integrated into a verification system with array arithmetic.

Programs	LOC	Proc.	Term	Loop	MayLoop	PC(s)	TC(s)	Overhead (%)
TPDB Benchmark								
Shuffle	20	2	2	0	0	0.23	0.26	11.54
LessLeavesRec	22	2	2	0	0	0.30	0.36	16.67
Alternate	23	2	2	0	0	0.37	0.39	5.13
SortCount	32	3	6	0	0	3.06	3.45	11.30
UnionFind	39	5	8	0	0	0.51	0.53	3.92
DivTernary	55	9	12	0	0	0.77	0.87	11.49
WorkingSignals	126	17	23	0	0	8.74	9.50	8.00
MinusUserDefined	21	2	10	0	0	0.30	0.36	16.67
MultiLasso	14	1	3	1	1	0.12	0.13	7.69
Totals/(%)	**352**	**43**	**68** (97.14%)	**1** (1.43%)	**1** (1.43%)	**14.40**	**15.85**	9.15%
SIR/Siemens Benchmark								
printtokens	726	18	41	0	0	15.92	19.51	18.40
printtokens2	570	19	36	0	0	52.14	53.70	2.91
replace	564	21	44	0	0	74.37	76.45	2.72
schedule	412	18	25	0	0	13.94	14.04	0.71
schedule2	374	16	27	0	0	8.44	8.94	5.59
tcas	173	9	9	0	0	12.55	13.21	5.00
Totals/(%)	**3171**	**144**	**250** (99.20%)	**1** (0.40%)	**1** (0.40%)	**191.76**	**201.70**	4.93%

Figure 3-9. Termination Verification for the SIR/Siemens and TPDB Benchmark

Moreover, for the *printtokens* programs of the benchmark, some of their methods required a precondition that the size of input files was finite for their termination; otherwise they might not terminate as indicated by failures of the termination verification. As can be seen, the termination of all programs in this benchmark is also verified successfully with a small overhead (about 5%).

3.6 Discussion

There exists a rich body of related works on automatic analysis for termination [28, 46, 102], non-termination [30, 70, 146], and both [65]. However, they consider termination and non-termination reasoning as distinct from functional correctness reasoning. Therefore, these works cannot leverage the result of functional correctness analysis to conduct more intricate (non-)termination reasoning. Recently, Brockschmidt *et al.* [29] propose a cooperation between safety and termination analysis to find sufficient supporting invariants for the construction of termination arguments but not considering non-termination. Chen *et al.* [35] introduce a similar approach for

proving only non-termination. Our proposal complements these works since our aim is to construct a logic where termination and non-termination properties are directly integrated into specification logics, and thus utilize the available infrastructure on functional correctness proofs. We have achieved this, and have also successfully evaluated its applicability on a wide range of programs, covering both numerical and heap-based programs.

Related to resource verification, [10] introduces a resource logic for a low-level language. While this logic avoids the need of auxiliary counters, it redefines the semantic model of the underlying logic to track the resource consumption via logical assertions, making the proposal harder to retrofit to other logics. Moreover, this logic only targets partial correctness, so that it does not take into account infinite resource consumption.

There are some works that are based on the well-foundedness of inductive definitions of heap predicates [21, 32] or user-defined quantitative functions over data structures [68] to prove termination of heap-manipulating programs. On one hand, they do not require any explicit ranking function. On the other hand, these approaches might have problems with programs like the Karatsuba multiplication method, in which the arguments of the recursive calls are not substructures of the input lists. In addition, the automated tools, such as AProVE and COSTA, cannot prove the termination of this method. In contrast, our approach is more flexible as it allows explicit termination measures, that are possibly non-linear, for proving programs' termination. These termination measures can be constructed from not only the heap structures but also the values of the data structures' elements. For example, we use the actual value of the natural presented by a linked list to bound the execution of the Karatsuba method. Moreover, we also allow non-termination to be specified and verified

for these programs. We believe that relatively complex examples, such as the Karatsuba method, highlight the benefits of our approach, which trades a lower level of automation but gains additional power.

The comparison of our approach with the other specification languages, *i.e.*. Dafny [106], JML [101], etc., has been discussed in Sec. 3.1. Another closely related work to ours is that of Nakata and Uustalu [114]. In this work, a Hoare logic for reasoning about non-termination of simple While programs (without method calls) was introduced. The logic is based on a trace-based semantics, in which the infiniteness of non-terminating traces is defined by coinduction. However, induction is still needed to define the finiteness of traces. In contrast, with resources, we can unify the semantics of the proposed termination and non-termination temporal constraints and allow the Hoare logic for functional correctness to be enhanced for termination and non-termination reasoning with minor changes. Moreover, our logic allows interprocedural verification in a modular fashion.

Chapter 4

Termination and Non-Termination Specification Inference

Techniques for proving termination and non-termination of imperative programs are usually considered as orthogonal mechanisms. In this chapter, we propose a novel mechanism that analyzes and proves both program termination and non-termination at the same time. We first introduce the concept of second-order termination constraints and accumulate a set of relational assumptions on them via a Hoare-style verification. We then solve these assumptions with case analysis to determine the (conditional) termination and non-termination scenarios expressed in the specification logic form proposed in Chapter 3. In contrast to current approaches, our technique can construct a summary of terminating and non-terminating behaviors for each method. This enables modularity and reuse for our termination and non-termination proving processes. We have tested our tool on sample programs from a recent termination competition, and compared favorably against two state-of-the-art termination analyzer(s).

4.1 Introduction

For the last ten years, we have seen a fruitful line of research on proving termination [20, 21, 26, 28, 29, 43–49, 61, 71, 74, 92, 95, 102, 104, 126, 127, 129] and non-termination [11, 30, 35, 70, 94, 123, 146] of imperative programs. However, the problems of proving program termination and non-termination are often considered separately, and not analysed simultaneously by the same tool (as far as we are aware). When a termination prover cannot prove termination, it might deploy its own non-termination analysis mechanism to obtain feasible counterexamples. However, the current techniques for proving non-termination are standalone techniques to existing termination proving mechanisms.

To capture the termination and non-termination behaviors of each program, we have proposed a specification logic with three temporal predicates `Term` M, `Loop` and `MayLoop`, which denote, respectively, the scenarios for definite program termination (with a lexicographic ranking measure M made of a list of positive integers), definite non-termination (with an unreachable post-condition) and indefinite (unknown) non-termination. However, this framework requires temporal specifications to be given by programmers.

We propose in this chapter a modular inference framework that can analyze both the termination and non-termination of each method in a program. This approach is novel in that it guides us to perform suitable case-splits on pre-conditions that lead to definite non-termination or definite termination, where possible. If a definite termination (or non-termination) case is not yet attained, we may perform a further case-split or decide to finish with a `MayLoop` classification to signify an unknown outcome. For each method, our inference mechanism incrementally constructs a *summary* of its termination,

non-termination or unknown behaviors, so that it can be reused in the inference of the remaining methods higher-up in the calling hierarchy.

```
void foo (int x, int y)
  requires U_pr(x, y)
  ensures U_po(x, y);
{ if (x < 0) return;
  else foo(x + y, y); }
```

Figure 4-1. The `foo` example

To support termination and non-termination inference, we introduce *unknown* temporal pre- and post-predicates in our specification logic to capture termination or non-termination behaviors (that are to be resolved by our inference). For example, in Fig. 4-1, the unknown pre-predicate $\mathtt{U_{pr}(x,y)}$ in the precondition of method `foo` denotes that the termination or non-termination status of `foo` is currently unknown. While the pre-predicate $\mathtt{U_{pr}(x,y)}$ in precondition of the method guides the overall inference process with suitable case-splits, the post-predicate $\mathtt{U_{po}(x,y)}$ in its postcondition is meant to capture the reachability or unreachability of the method's exits. This post-predicate will be strengthened to `false` in scenarios where `foo` is definitely non-terminating. This post-predicate can also be used to trivially determine base-case scenarios with immediate termination property. This combined use of unknown pre- and post-predicates is somewhat novel, since it allows us to modularly analyze each method (with the help of case-splits where needed) to obtain a comprehensive summary of the method's termination and non-termination characteristics.

4.2 Overview of Our Approach

Specification Language. In the previous chapter, we proposed three temporal predicates, `Term` M, `Loop` and `MayLoop`, to help reason about program termination and non-termination. For the current evaluation, we adopt

these predicates as well as a rich underlying specification proposed by [133] that is able to express both heap properties with separation logic (κ in Fig. 4-2) and pure (non-heap) properties with Presburger arithmetic (π in Fig. 4-2). This logic uses a fragment of separation logic with the separation conjunction $*$ to denote the disjointness of heap parts and the heap predicate $hpred$ (Fig. 4-2) to specify various data structures. Moreover, to simplify the presentation, we express a specification as a pair of pre- and post-condition (see $spec$ in Fig. 4-2). In our example programs, specifications will be written using the usual `requires...ensures...` form for better readability. Specifically for termination reasoning, we have designed the termination measure M as a (finite) list of arithmetic expressions $[\overline{e}]$ whose order is based on the lexicographic ordering $<_l$ (recall Defn. 5) and $e{:}es$ denotes a non-empty list with e and es as its head and tail, respectively.

$$\frac{}{[] <_l e{:}_} \qquad \frac{(e_1 < e_2) \vee (e_1 = e_2 \wedge es_1 <_l es_2)}{e_1{:}es_1 <_l e_2{:}es_2}$$

To facilitate termination and non-termination inference, we allow the use of unknown temporal pre-predicate $\mathtt{U_{pr}}(\overline{v})$ and post-predicate $\mathtt{U_{po}}(\overline{v})$ in the specification language to indicate the unknown termination status of a program. The solutions of these unknown predicates would be then derived by the inference mechanism, as shown next. Note that the inferred result for each unknown pre-predicate $\mathtt{U_{pr}}(\overline{v})$ will be of the form $\bigvee(\pi \wedge \theta)$ with θ ranging over $\{\mathtt{Term}\ [\overline{e}], \mathtt{Loop}, \mathtt{MayLoop}\}$; while the inferred result for each unknown post-predicate $\mathtt{U_{po}}(\overline{v})$ will be in a guarded conjunction $\bigwedge(\pi \Rightarrow post)$ with $post$ being `true` or `false`. Such a guarded form is equivalent to a disjunctive form $\bigvee(\pi \wedge post)$ when the set of guards are complete.

$$
\begin{array}{llll}
\mathit{hpred} ::= c(\overline{v}) \equiv \bigvee(\exists \overline{u} \cdot \rho) & \rho ::= \kappa \wedge \pi \\
\mathit{spec} ::= (\Psi_{\text{Pre}}, \Phi_{\text{Post}}) & \kappa ::= \texttt{emp} \mid v{\mapsto}d(\overline{u}) \mid c(\overline{v}) \mid \kappa_1{*}\kappa_2 \\
\Psi_{\text{Pre}} ::= \bigvee(\exists \overline{u}{\cdot}(\rho{\wedge}\theta)) & \pi ::= b \mid a \mid \pi_1{\wedge}\pi_2 \mid \neg\pi \mid \exists v{\cdot}\pi \\
\Phi_{\text{Post}} ::= \bigvee(\exists \overline{u}{\cdot}(\rho{\wedge}\texttt{U}_{\texttt{po}}(\overline{v}))) & b ::= \texttt{false} \mid v \mid b_1{=}b_2 \\
\theta ::= \texttt{Term}\ [\overline{e}] \mid \texttt{Loop} \mid & a ::= e_1{=}e_2 \mid e_1{<}e_2 \mid v{=}\texttt{null} \\
\qquad \texttt{MayLoop} \mid \texttt{U}_{\texttt{pr}}(\overline{v}) & e ::= k \mid v \mid k{\times}e \mid e_1{+}e_2 \mid -e
\end{array}
$$

where emp denotes an empty heap; $v{\mapsto}d(\overline{u})$ specifies a heap node of data type d; k is a constant; u, v are variables

Figure 4-2. A Specification Language with Unknown Predicates for Inference

Illustrating Example. We now demonstrate how our inference mechanism derives the preconditions for termination and non-termination of method `foo` in Fig. 4-1. Initially, the termination and non-termination behaviors of method `foo` are captured by a pair of unknown pre-predicate $\texttt{U}_{\texttt{pr}}(\texttt{x}, \texttt{y})$ and unknown post-predicate $\texttt{U}_{\texttt{po}}(\texttt{x}, \texttt{y})$. Like the other known temporal predicates for termination and non-termination reasoning, these unknown predicates are part of the specification logic's formulas and can therefore be reasoned in the same way via a Hoare-style verification. With the help of an enhanced entailment procedure, we shall prove that the precondition of each method call is always satisfied and the postcondition always holds at the end of the method body.

For example, the verification conditions (VCs) encountered by Hoare-style forward verification of method `foo` are:

(c_1) $\texttt{x}{<}0 \wedge \texttt{U}_{\texttt{pr}}(\texttt{x}, \texttt{y}) \vdash \texttt{U}_{\texttt{po}}(\texttt{x}, \texttt{y})$

(c_2) $\texttt{x}{\geq}0 \wedge \texttt{x}'{=}\texttt{x}{+}\texttt{y} \wedge \texttt{y}'{=}\texttt{y} \wedge \texttt{U}_{\texttt{pr}}(\texttt{x}, \texttt{y}) \vdash \texttt{U}_{\texttt{pr}}(\texttt{x}', \texttt{y}')$

(c_3) $\texttt{x}{\geq}0 \wedge \texttt{x}'{=}\texttt{x}{+}\texttt{y} \wedge \texttt{y}'{=}\texttt{y} \wedge \texttt{U}_{\texttt{pr}}(\texttt{x}, \texttt{y}) \wedge \texttt{U}_{\texttt{po}}(\texttt{x}', \texttt{y}') \vdash \texttt{U}_{\texttt{po}}(\texttt{x}, \texttt{y})$,

The first VC (c_1) is obtained from the base-case scenario when the post-condition of the `foo` method is being proven. The second VC (c_2) captures the proving of precondition for the recursive call, while the last VC

(c_3) captures the entailment proving of the postcondition of method `foo` in the recursive branch. These VCs capture the unknown termination behaviors of both the caller (*i.e.* denoted by the pair of predicates $\mathtt{U_{pr}}(\mathtt{x}, \mathtt{y})$ and $\mathtt{U_{po}}(\mathtt{x}, \mathtt{y})$) and the callee (*i.e.* denoted by $\mathtt{U_{pr}}(\mathtt{x}', \mathtt{y}')$ and $\mathtt{U_{po}}(\mathtt{x}', \mathtt{y}')$).

For these unknown predicates, we attempt to derive the strongest possible post-predicate, where possible. As we intend to capture the unreachability of each post-predicate, the strongest post-predicate in our analysis is actually `false`. If our inference for falsity of post-predicates fails, we denote its possible reachability by `true` instead and then attempt to infer the weakest pre-predicate, where possible. The temporal pre-predicates are ordered by the following implication hierarchy `MayLoop` $\Rightarrow_r$ `Loop` and `MayLoop` $\Rightarrow_r$ `Term` $[\overline{e}]$. Amongst them, `MayLoop` is considered as the strongest one, which is analogous to `false` in the domain of logical specification. The intuition is that `MayLoop` can be used to denote the termination property of any program though such a use would form a rather poor specification, similar to how `false` could be naively (and redundantly) used as the precondition for any program. On the other hand, the `Loop` and `Term` $[\overline{e}]$ predicates are incomparable since they denote disjoint classes of programs (*i.e.* definitely non-terminating vs. definitely terminating programs, respectively). Our inference thus attempts to discover the weaker `Loop` and `Term` $[\overline{e}]$ for its unknown pre-predicate, where possible.

From the earlier VCs, we infer three relational assumptions where unknown pre-predicate $\mathtt{U_{pr}}(\mathtt{x}', \mathtt{y}')$ is related inductively to an earlier pre-predicate $\mathtt{U_{pr}}(\mathtt{x}, \mathtt{y})$ (see (a_2^0)), while unknown post-predicate $\mathtt{U_{po}}(\mathtt{x}, \mathtt{y})$ is either expressed in base-case form (see (a_1^0)) or related inductively to an earlier occurrence of the post-predicate $\mathtt{U_{po}}(\mathtt{x}', \mathtt{y}')$ (see (a_3^0)).

(a^0_1) $\mathtt{x}{<}0{\wedge}\mathtt{true} \Rightarrow \mathtt{U_{po}}(\mathtt{x},\mathtt{y})$

(a^0_2) $\mathtt{x}{\geq}0 \wedge \mathtt{x}'{=}\mathtt{x}{+}\mathtt{y} \wedge \mathtt{y}'{=}\mathtt{y}{\wedge}\mathtt{U_{pr}}(\mathtt{x},\mathtt{y}) \Rightarrow \mathtt{U_{pr}}(\mathtt{x}',\mathtt{y}')$

(a^0_3) $\mathtt{x}{\geq}0 \wedge \mathtt{x}'{=}\mathtt{x}{+}\mathtt{y} \wedge \mathtt{y}'{=}\mathtt{y}{\wedge}\mathtt{U_{po}}(\mathtt{x}',\mathtt{y}') \Rightarrow \mathtt{U_{po}}(\mathtt{x},\mathtt{y})$,

We derive inductive definitions for these unknown predicates, in order to give the best possible interpretations to their temporal predicates. In the case of post-predicate, we attempt to determine its reachability or unreachability, so that we can immediately decide on either (base-case scenario for) termination or (inductive-case scenario for) definite non-termination. From the relational assumption (a^0_1), we can immediately infer a base-case scenario $\mathtt{x}{<}0$ where the `foo` method would terminate. The other two relational assumptions occur under a different scenario $\mathtt{x}{\geq}0$ which neither indicates definite termination nor definite non-termination. From these partial instantiations on the two unknown temporal predicates, we refine them to the following definitions:

$$\begin{array}{lccc} \mathtt{U_{pr}}(\mathtt{x},\mathtt{y}) \equiv & \mathtt{x}{<}0 \wedge \mathtt{Term} & \vee & \mathtt{x}{\geq}0 \wedge \mathtt{U^1_{pr}}(\mathtt{x},\mathtt{y}) \\ \mathtt{U_{po}}(\mathtt{x},\mathtt{y}) \equiv & (\mathtt{x}{<}0 \Rightarrow \mathtt{true}) & \wedge & (\mathtt{x}{\geq}0 \Rightarrow \mathtt{U^1_{po}}(\mathtt{x},\mathtt{y})) \end{array}$$

where two auxiliary unknown predicates are introduced for the input scenario $\mathtt{x}{\geq}0$. Note that `Term`, short for `Term` [ ], is used to denote base-case termination scenario where its lexicographic ranking measure is trivially empty. Our unknown pre-predicate is being expressed as a disjunction on either known or unknown temporal resource constraints, while the post-predicate is being expressed as a guarded conjunction of either reachability (`true`), unreachability (`false`) or unknown. That is the two predicates are currently known for the input scenario $\mathtt{x}{<}0$ but unknown for the scenario $\mathtt{x}{\geq}0$. This can also be

expressed as a case structured specification:

$$\texttt{case}\{\ \texttt{x} < 0 \ \rightarrow\ \texttt{requires}\ \texttt{Term}\ \texttt{ensures}\ \texttt{true};$$
$$\texttt{x} \geq 0 \ \rightarrow\ \texttt{requires}\ \texttt{U}^{1}_{\texttt{pr}}(\texttt{x},\texttt{y})\ \texttt{ensures}\ \texttt{U}^{1}_{\texttt{po}}(\texttt{x},\texttt{y});\}$$

As the precondition is now partially known, we could refine each (a^0_i) through a substitution with the partial definition of $\texttt{U}_{\texttt{pr}}(\texttt{x},\texttt{y})$ and $\texttt{U}_{\texttt{po}}(\texttt{x},\texttt{y})$ to get the new relational assumptions (trivially valid assumptions are omitted here):

(a^0_{2a}) $\texttt{x}{\geq}0 \wedge \texttt{x}'{=}\texttt{x}{+}\texttt{y} \wedge \texttt{y}'{=}\texttt{y} \wedge \texttt{x}'{<}0{\wedge}\texttt{U}^{1}_{\texttt{pr}}(\texttt{x},\texttt{y}) \Rightarrow \texttt{Term}$

(a^0_{2b}) $\texttt{x}{\geq}0 \wedge \texttt{x}'{=}\texttt{x}{+}\texttt{y} \wedge \texttt{y}'{=}\texttt{y} \wedge \texttt{x}'{\geq}0{\wedge}\texttt{U}^{1}_{\texttt{pr}}(\texttt{x},\texttt{y}) \Rightarrow \texttt{U}^{1}_{\texttt{pr}}(\texttt{x}',\texttt{y}')$

(a^0_{3a}) $\texttt{x}{\geq}0 \wedge \texttt{x}'{=}\texttt{x}{+}\texttt{y} \wedge \texttt{y}'{=}\texttt{y}{\wedge}(\texttt{x}'{\geq}0{\Rightarrow}\texttt{U}_{\texttt{po}}(\texttt{x}',\texttt{y}')) \Rightarrow (\texttt{x}{\geq}0{\Rightarrow}\texttt{U}_{\texttt{po}}(\texttt{x},\texttt{y}))$.

The relational assumption (a^0_{2a}) describes the reachability of the base-case condition (*i.e.* $\texttt{x}'{<}0$), denoted by $\texttt{Term}$, under the input scenario $\texttt{x}{\geq}0$. As this base-case condition is feasible in the current context, we have a possibly terminating scenario that signifies a termination proof which tries to synthesize a ranking function but this proof fails. We then try a non-termination proof by examining the relational assumption (a^0_{3a}) on unknown post-predicate to determine a pre-condition for unreachability. Such condition would ensure that the base case is not reached in the next recursion, *i.e.* $\texttt{x}'{\geq}0$, and we refer to this as *potential* non-termination pre-condition. The condition $\texttt{x}{+}\texttt{y}{\geq}0$ would be a trivial potential non-termination pre-condition for the relational assumption (a^0_{3a}) (as $\texttt{x}'{=}\texttt{x}{+}\texttt{y}$). However, our inference engine would attempt to discover more precise conditions for definite non-termination with the help of abductive inference [124] and in this case it suggests $\texttt{y}{\geq}0$. With this, a case-split with the condition $\texttt{y}{\geq}0$ and its negation $\texttt{y}{<}0$ is used to refine the definitions for

$\mathtt{U}^1_{\mathtt{pr}}(\mathtt{x},\mathtt{y})$ and $\mathtt{U}^1_{\mathtt{po}}(\mathtt{x},\mathtt{y})$ into

$$\begin{array}{llll} \mathtt{U}^1_{\mathtt{pr}}(\mathtt{x},\mathtt{y}) \equiv & \mathtt{y}{\geq}0 \wedge \mathtt{U}^2_{\mathtt{pr}}(\mathtt{x},\mathtt{y}) & \vee & \mathtt{y}{<}0 \wedge \mathtt{U}^3_{\mathtt{pr}}(\mathtt{x},\mathtt{y}) \\ \mathtt{U}^1_{\mathtt{po}}(\mathtt{x},\mathtt{y}) \equiv & (\mathtt{y}{\geq}0 \Rightarrow \mathtt{U}^2_{\mathtt{po}}(\mathtt{x},\mathtt{y})) & \wedge & (\mathtt{y}{<}0 \Rightarrow \mathtt{U}^3_{\mathtt{po}}(\mathtt{x},\mathtt{y})) \end{array}$$

and consequently the following six specialized assumptions are derived from the earlier ones (a^0_{2a}), (a^0_{2b}) and (a^0_3).

(a^1_1) $\mathtt{x}{\geq}0{\wedge}\mathtt{x}'{=}\mathtt{x}{+}\mathtt{y}{\wedge}\mathtt{y}'{=}\mathtt{y}{\wedge}\mathtt{x}'{<}0{\wedge}\mathtt{y}{\geq}0{\wedge}\mathtt{U}^2_{\mathtt{pr}}(\mathtt{x},\mathtt{y}) \Rightarrow \mathtt{Term}$

(a^1_2) $\mathtt{x}{\geq}0{\wedge}\mathtt{x}'{=}\mathtt{x}{+}\mathtt{y}{\wedge}\mathtt{y}'{=}\mathtt{y}{\wedge}\mathtt{x}'{\geq}0{\wedge}\mathtt{y}{\geq}0{\wedge}\mathtt{U}^2_{\mathtt{pr}}(\mathtt{x},\mathtt{y}) \Rightarrow \mathtt{U}^2_{\mathtt{pr}}(\mathtt{x}',\mathtt{y}')$

(a^1_3) $\mathtt{x}{\geq}0{\wedge}\mathtt{x}'{=}\mathtt{x}{+}\mathtt{y}{\wedge}\mathtt{y}'{=}\mathtt{y}{\wedge}(\mathtt{x}'{\geq}0{\wedge}\mathtt{y}'{\geq}0{\Rightarrow}\mathtt{U}^2_{\mathtt{po}}(\mathtt{x}',\mathtt{y}')) \Rightarrow (\mathtt{x}{\geq}0{\wedge}\mathtt{y}{\geq}0{\Rightarrow}\mathtt{U}^2_{\mathtt{po}}(\mathtt{x},\mathtt{y}))$,

(a^1_4) $\mathtt{x}{\geq}0{\wedge}\mathtt{x}'{=}\mathtt{x}{+}\mathtt{y}{\wedge}\mathtt{y}'{=}\mathtt{y}{\wedge}\mathtt{x}'{<}0{\wedge}\mathtt{y}{<}0{\wedge}\mathtt{U}^3_{\mathtt{pr}}(\mathtt{x},\mathtt{y}) \Rightarrow \mathtt{Term}$

(a^1_5) $\mathtt{x}{\geq}0{\wedge}\mathtt{x}'{=}\mathtt{x}{+}\mathtt{y}{\wedge}\mathtt{y}'{=}\mathtt{y}{\wedge}\mathtt{x}'{\geq}0{\wedge}\mathtt{y}{<}0{\wedge}\mathtt{U}^3_{\mathtt{pr}}(\mathtt{x},\mathtt{y}) \Rightarrow \mathtt{U}^3_{\mathtt{pr}}(\mathtt{x}',\mathtt{y}')$

(a^1_6) $\mathtt{x}{\geq}0{\wedge}\mathtt{x}'{=}\mathtt{x}{+}\mathtt{y}{\wedge}\mathtt{y}'{=}\mathtt{y}{\wedge}(\mathtt{x}'{\geq}0{\wedge}\mathtt{y}'{<}0{\Rightarrow}\mathtt{U}^3_{\mathtt{po}}(\mathtt{x}',\mathtt{y}')) \Rightarrow (\mathtt{x}{\geq}0{\wedge}\mathtt{y}{<}0{\Rightarrow}\mathtt{U}^3_{\mathtt{po}}(\mathtt{x},\mathtt{y}))$

The first three relational assumptions, $(a^1_1) - (a^1_3)$, form a group which will be analyzed together for the given input scenario $\mathtt{x}{\geq}0{\wedge}\mathtt{y}{\geq}0$. The next three relational assumptions, $(a^1_4) - (a^1_6)$, form another group that will be analyzed together for the input scenario $\mathtt{x}{\geq}0{\wedge}\mathtt{y}{<}0$.

The first group of relational assumptions, $(a^1_1) - (a^1_3)$, allows us to confirm a definite non-termination scenario, since we can use (a^1_3) to determine the unreachability of its post-predicate $\mathtt{U}^2_{\mathtt{po}}(\mathtt{x},\mathtt{y})$. By using the hypothesis $\mathtt{U}^2_{\mathtt{po}}(\mathtt{x},\mathtt{y}){\equiv}\mathtt{false}$ for both occurrences of the post-predicate $\mathtt{U}^2_{\mathtt{po}}(\mathtt{x},\mathtt{y})$ in (a^1_3), we can inductively determine the falsity (or unreachability) of $\mathtt{U}^2_{\mathtt{po}}(\mathtt{x},\mathtt{y})$. This hypothesis declares that post-predicate is unreachable (false) under the pre-condition $\mathtt{x}{\geq}0{\wedge}\mathtt{y}{\geq}0$. Note our use of inductive reasoning here which assumes the hypothesis that $\mathtt{U}^2_{\mathtt{po}}(\mathtt{x},\mathtt{y})$ is unreachable under pre-condition $\mathtt{x}{\geq}0{\wedge}\mathtt{y}{\geq}0$ (by the recursive call), in order to prove the same hypothesis.

The second group of relational assumptions, $(a^1_4) - (a^1_6)$, suggests us to prove the method's termination under the precondition $\mathtt{x}{\geq}0{\wedge}\mathtt{y}{<}0$ first, since its base case (captured by (a^1_4)) is possibly reachable under this condition. This termination scenario is confirmed, once we have derived a lexicographic ranking measure $[\mathtt{x}]$ that is bounded and would moreover decrease with each recursive invocation for the pre-predicate $\mathtt{U}^3_{\mathtt{pr}}(\mathtt{x},\mathtt{y})$ using (a^1_5).

As a summary of our combined analyses, we have effectively derived the following definitions for the two unknown predicates:

$$\begin{aligned}\mathtt{U_{pr}}(\mathtt{x},\mathtt{y}) \equiv\ & \mathtt{x}{<}0{\wedge}\mathtt{Term} \vee \mathtt{x}{\geq}0{\wedge}\mathtt{y}{<}0{\wedge}\mathtt{Term}[\mathtt{x}] \vee \mathtt{x}{\geq}0{\wedge}\mathtt{y}{\geq}0{\wedge}\mathtt{Loop}\\ \mathtt{U_{po}}(\mathtt{x},\mathtt{y}) \equiv\ & (\mathtt{x}{<}0 \Rightarrow \mathtt{true}) \wedge (\mathtt{x}{\geq}0{\wedge}\mathtt{y}{<}0 \Rightarrow \mathtt{true}) \wedge (\mathtt{x}{\geq}0{\wedge}\mathtt{y}{\geq}0 \Rightarrow \mathtt{false})\end{aligned}$$

Note how the unknown temporal predicates $\mathtt{U}^2_{\mathtt{pr}}(\mathtt{x},\mathtt{y})$ and $\mathtt{U}^2_{\mathtt{po}}(\mathtt{x},\mathtt{y})$ are being resolved to be `Loop` and an unreachable `false` for input scenario $\mathtt{y}{\geq}0$, respectively. In contrast, the unknown predicates $\mathtt{U}^3_{\mathtt{pr}}(\mathtt{x},\mathtt{y})$ and $\mathtt{U}^3_{\mathtt{po}}(\mathtt{x},\mathtt{y})$ are being resolved to be $\mathtt{Term}\,[\mathtt{x}]$ and a reachable `true` state for input scenario $\mathtt{y}{<}0$, respectively.

Using the inferred predicate definitions, we can construct the following case-based specification which fully captures termination and non-termination behaviors for method `foo`.

$$\begin{array}{l}\mathtt{case}\ \{\\ \quad \mathtt{x} < 0 \rightarrow \mathtt{requires\ Term\ ensures\ true};\\ \quad \mathtt{x} \geq 0 \rightarrow \mathtt{case}\ \{\\ \qquad \mathtt{y} < 0 \rightarrow \mathtt{requires\ Term}\,[\mathtt{x}]\ \mathtt{ensures\ true};\\ \qquad \mathtt{y} \geq 0 \rightarrow \mathtt{requires\ Loop\ ensures\ false};\ \}\}\end{array}$$

int Ack (int m, int n)	int Mc91 (int n)
requires true ensures res $\geq$ n+1; { if (m == 0) return n + 1; else if (n == 0) return Ack(m − 1, 1); else return Ack(m − 1,Ack(m, n − 1)); }	requires true ensures res $\geq$ 91; { if (n > 100) return n − 10; else return Mc91(Mc91(n + 11)); }
(a)	(b)

Figure 4-3. Functions with Nested Recursion: Ackermann function (a) and McCarthy 91 function (b)

4.2.1 Other Examples

Our termination and non-termination inference is completely automated. By allowing unknown temporal predicates into functional correctness specifications, our inference mechanism can freely leverage on prior infrastructures to *(i)* handle a wider class of programs, and to *(ii)* improve the accuracy of the inference results. Note that prior specifications for the analyzed methods might be manually given or be automatically derived by other inference mechanisms, but they are orthogonal to our current proposal.

We list below some interesting examples to demonstrate how our inference mechanism works with programs that already have some safety specifications.

Nested Recursion. Some knowledge about the output of methods with nested recursion, such as the Ackermann function and the McCarthy 91 function in Fig. 4-3, is crucial for the inference of their termination and non-termination properties. Without any specification, our inference mechanism returns incomplete summaries on the terminating and non-terminating behaviors of these two functions. The result for the

Ackermann function is summarized as:

$$\texttt{case}\ \{$$
$$\texttt{m}{=}0 \quad \rightarrow \texttt{requires Term ensures true};$$
$$\texttt{m}{<}0 \vee \texttt{n}{<}0 \quad \rightarrow \texttt{requires Loop ensures false};$$
$$\texttt{m}{>}0 \wedge \texttt{n}{\geq}0 \quad \rightarrow \texttt{requires MayLoop ensures true};\ \}$$

While the inference shows that this function is terminating when $\texttt{m}{=}0$ (base case) or non-terminating when $\texttt{m}{<}0 \vee \texttt{n}{<}0$, it cannot prove the termination of the function under the input scenario $\texttt{m}{>}0{\wedge}\texttt{n}{\geq}0$ since the value of the second argument in the last recursive call is unknown (or unbounded). However, with the stronger specification given in Fig. 4-3(a), with an upper bound $\texttt{res} \geq \texttt{n}{+}1$ on the function's returned value, denoted by `res`, our inference mechanism can replace `MayLoop` in scenario $\texttt{m}{>}0{\wedge}\texttt{n}{\geq}0$ by $\texttt{Term}\,[\texttt{m},\texttt{n}]$ where $[\texttt{m},\texttt{n}]$ is a valid lexicographic ranking function. Similarly, without specification, the inference only shows that the McCarthy 91 function terminates in it base case when $\texttt{n}{>}100$. However, with the specification given in Fig. 4-3(b), our inference can prove that the function terminates for all inputs.

While our termination inference mechanism does not directly infer bounded postconditions, it can be (and has been) made to work with other automated postcondition inference sub-systems, such as [69, 130]. Such postcondition inference sub-systems are orthogonal to our proposal, and can be leveraged to provide a more comprehensive solution for fully automated termination and non-termination inference.

Heap-Manipulating Programs. Our inference mechanism can be also integrated into existing verification frameworks (such as [38], or even shape inference system [97]) that reason about safety properties of heap programs via

data node { node next; }
 	
pred lseg(root, q, n) ≡ root=q∧n=0
∨ root↦node(p) ∗ lseg(p, q, n−1)
 	
pred cll(root, n) ≡
root↦node(p)∗lseg(p, root, n−1)
	void append (node x, node y)
	requires lseg(x, null, n)∧x≠null
	ensures lseg(x, y, n);
	requires cll(x, n) ensures true;
	{ if (x.next == null) x.next = y;
	else append(x.next, y); }

Figure 4-4. Specification with Implementation for append method of two linked lists

separation logic [133]. This extension could help prove the termination and non-termination of heap-manipulating programs.

For example, Fig. 4-4 shows the specification and implementation (for the verification) of the method append that concatenates two linked lists x and y. With the separation conjunction $*$ and the points-to operator $\mapsto$ of separation logic, the heap predicate lseg(root, q, n) represents a list segment from root to q with n elements. This predicate can then be used in the declarations of other predicates, such as cll(root, n) for circular lists. Using these predicates, we can capture two safety specifications of append in Fig. 4-4.

In the first scenario when the input x is a null-terminating list with size n, our inference mechanism is able to show that the method append always terminates with the ranking function [n]. In the second scenario where x is a circular linked list, our inference can show that append is definitely non-terminating, after confirming (by induction) that its postcondition can be strengthened to false. These examples highlight the modular nature of our non-termination and termination inference mechanism, which can be built on top of other inference mechanisms.

4.3 From Verification to Inference

So far we have illustrated a unified specification logic with three known temporal predicates: $\texttt{Term}\ [\overline{e}]$, $\texttt{Loop}$ and $\texttt{MayLoop}$. Semantically, these predicates can be defined using resource capacities (on lower and upper bounds) of execution length, *i.e.* $\texttt{Term}\ [\overline{e}] =_{df} \texttt{RC}\langle 0, f([\overline{e}])\rangle$, $\texttt{Loop} =_{df} \texttt{RC}\langle \infty, \infty\rangle$, and $\texttt{MayLoop} =_{df} \texttt{RC}\langle 0, \infty\rangle$. The resource predicate $\texttt{RC}\langle L, U\rangle$ specifies a resource capacity with a lower bound L and an upper bound U. It is satisfied by each program state whose resource capacity (l, u) is subsumed by (L, U), i.e. $l{\geq}L$ and $u{\leq}U$. Note that the function $f([\overline{e}])$ obtains a finite bound through an order-embedding of $[\overline{e}]$ into naturals.

Verification conditions involving these temporal predicates can be discharged by a resource consumption entailment $\vdash_t$, that is used to account for (lower and upper bound) resources that are utilized by each code fragment. Such entailment can be used to analyze termination or non-termination property for some given method via resource reasoning. Given the temporal constraint θ_a associated with the current program state ρ and the temporal resource constraint θ_c (of some code fragment that must be executed), the entailment $\rho \wedge \theta_a \vdash_t \theta_c \blacktriangleright \theta_r$ firstly checks whether the execution resource required by constraint θ_c can be met by the execution resource of constraint θ_a or not. In the former case, the entailment will return the remaining execution resource that is denoted by residue θ_r.

In terms of the actual execution capacity, this consumption entailment can be formalized by the following rule:

$$\frac{\rho \Rightarrow U_c{\leq}U_a \quad L_r = L_a{-_l}L_c \quad U_r = U_a{-_u}U_c \quad \rho \Rightarrow L_r{\leq}U_r}{\rho \wedge \texttt{RC}\langle L_a, U_a\rangle \vdash_t \texttt{RC}\langle L_c, U_c\rangle \blacktriangleright \texttt{RC}\langle L_r, U_r\rangle}$$

where two subtraction operators are designed to cater to an integer domain

extended with the ∞ value (*i.e.* $\mathtt{N}^\infty$):

$$L_1 -_l L_2 \equiv \mathit{min}\{r \in N^\infty \mid r + L_2 \geq L_1\}$$

$$U_1 -_u U_2 \equiv \mathit{max}\{r \in N^\infty \mid r + U_2 \leq U_1\},\ \mathit{if}\ U_1{\geq}U_2$$

These two operators are essentially integer subtraction operators, except that their results are never negative and such that $\infty{-_l}\infty = 0$ and $\infty{-_u}\infty = \infty$. They are formulated in this way to give the best (or largest) possible lower and upper bound values to denote the execution capacity of residue. In addition, the subtraction $U_a -_u U_c$ requires a check for upper bound execution capacity, namely $\rho \Rightarrow U_c{\leq}U_a$. This check is important to ensure that resource consumption is within the specified upper bound, and will also ensure that the residue is a valid resource capacity.

The resource implication operator $\Rightarrow_r$ on execution capacity, which is used earlier in the implication hierarchy of known temporal predicates, can be defined based on the following subsumption relation, in which the lower and upper bound exhibit dual (opposite) properties.

$$\frac{L_1{\leq}L_2 \quad U_2{\leq}U_1}{\mathtt{RC}\langle L_1, U_1\rangle \Rightarrow_r \mathtt{RC}\langle L_2, U_2\rangle}$$

From this definition, `MayLoop` is the strongest pre-predicate in the subsumption hierarchy since it has the maximum execution capacity $(0, \infty)$ to be able to subsume either `Loop` (with execution capacity (∞, ∞)) or `Term` $[\overline{e}]$ (with execution capacity $(0, f([\overline{e}]))$) predicates. Note that the implication operator $\Rightarrow_r$ is only weakly related to the resource consumption entailment operator, $\vdash_t$, as follows:

$$(\theta_a \Rightarrow_r \theta_c) \Rightarrow \exists \theta_r \cdot \theta_a \vdash_t \theta_c \blacktriangleright \theta_r$$

For termination and non-termination inference, we have introduced unknown predicates $\mathtt{U_{pr}}(\overline{v})$ for precondition and $\mathtt{U_{po}}(\overline{v})$ for postcondition for

each method, with $\mathtt{U_{pr}}(\overline{v})$ denoting some execution capacity, and $\mathtt{U_{po}}(\overline{v})$ specifying reachability of a method with a set of formal parameters $\overline{v}$. To support its inference, we will have to extend the resource entailment procedure to handle entailments between known and unknown temporal constraints.

The most general form of temporal entailment is $\rho \wedge \bigwedge_i \mathtt{U}^i_{\mathtt{po}}(\overline{v_i}) \wedge \theta_a \vdash_t \theta_c \blacktriangleright (\theta_r, \mathcal{R})$, where each $\mathtt{U}^i_{\mathtt{po}}(\overline{v_i})$ is an unknown post-predicate accumulated into the program state after a recursive method call. The temporal constraint θ_a in the antecedent of the entailment might be an unknown pre-predicate $\mathtt{U_{pr}}(\overline{v})$ or a known temporal predicate. The temporal constraint θ_c can be either an unknown post-predicate $\mathtt{U_{po}}(\overline{v})$ or a known predicate. The residue constraint θ_r denotes the residual capacity after entailment. Each relational assumption $\mathcal{R}$ for the unknown temporal predicates is a pre-requisite to ensure the validity of the entailment when either θ_a or θ_c is unknown. It is defined as below.

Definition 7 *The temporal relational assumption $\mathcal{R}$ in the residue of a temporal entailment $\rho \wedge \bigwedge_i \mathtt{U}^i_{\mathtt{po}}(\overline{v_i}) \wedge \theta_a \vdash_t \theta_c \blacktriangleright (\theta_r, \mathcal{R})$ can be defined as follows:*

(i) $\mathcal{R} \equiv \mathtt{true}$, *if both θ_a and θ_c are known predicates from* $\{\mathtt{Term}\ [\overline{e}], \mathtt{Loop}, \mathtt{MayLoop}\}$.

(ii) $\mathcal{R} \equiv \rho \wedge \bigwedge_i \mathtt{U}^i_{\mathtt{po}}(\overline{v_i}) \Rightarrow \theta_c$, *if θ_c is an unknown post-predicate.*

(iii) $\mathcal{R} \equiv \rho \wedge \theta_a \Rightarrow \theta_c$, *otherwise.*

This temporal entailment can be integrated into an entailment system with frame $\Psi \vdash \Phi \leadsto \Psi_r$, to obtain a new entailment procedure of the form $\Psi \vdash \Phi \leadsto (\Psi_r, \mathcal{S})$, that also captures in its residue the set of relational assumptions $\mathcal{S}$ generated by the temporal sub-entailments. The rules to discharge entailments

of logic formulas with disjunctions are:

$$\frac{\Psi = \bigvee_i \exists \overline{v_i} \cdot (\rho_i {\wedge} \bigwedge_j \theta_i^j) \qquad \forall i \cdot (\rho_i {\wedge} \bigwedge_j \theta_i^j) \vdash \Phi \leadsto (\Psi_r^i, \mathcal{S}_i)}{\Psi \vdash \Phi \leadsto (\bigvee_i \exists \overline{v_i} \cdot \Psi_r^i,\ \bigcup_i \mathcal{S}_i)}\ [\textbf{ENT-DISJ-LHS}]$$

$$\frac{\rho_a \vdash \rho_c \leadsto \rho_r \qquad \rho_r \wedge \bigwedge_i \mathtt{U}_{\mathtt{po}}^i(\overline{v_i}) \wedge \theta_a \vdash_t \theta_c \blacktriangleright (\theta_r, \mathcal{R})}{\rho_a {\wedge} \bigwedge_i \mathtt{U}_{\mathtt{po}}^i(\overline{v_i}) {\wedge} \theta_a \vdash \rho_c {\wedge} \theta_c \leadsto (\rho_r {\wedge} \bigwedge_i \mathtt{U}_{\mathtt{po}}^i(\overline{v_i}) {\wedge} \theta_r, \{\mathcal{R}\})}\ [\textbf{ENT-CONJ}]$$

4.4 Generation of Temporal Relational Assumptions

In this section, we show how our new entailment procedure is incorporated into Hoare logic to generate a set of relation assumptions over the unknown temporal constraints.

To support inference, Hoare judgment is formalized in the form $\vdash \{\Psi_{\text{Pre}}\}\ e\ \{\Phi_{\text{Post}}, \mathcal{S}\}$, where $\mathcal{S}$ is a generated set of temporal assumptions. For illustration, we show the new rule for method call:

$$\frac{\begin{array}{c} t_0\ mn(\overline{t\ v})\ (\Psi_{\text{Pre}}, \Phi_{\text{Post}})\ \{e\} \in Prog \\ \Psi \vdash \Psi_{\text{Pre}} \leadsto (\Phi, \mathcal{S}_1) \quad \Psi_r = \Phi * \Phi_{\text{Post}} \quad \mathcal{S}_2 = \mathtt{filter}(\mathcal{S}_1) \end{array}}{\vdash \{\Psi\}\ mn(\overline{v})\ \{\Psi_r, \mathcal{S}_2\}}\ [\textbf{FV-CALL}]$$

To facilitate the termination inference, at method calls, we collect only *nontrivial* assumptions of unknown temporal constraints. We list below trivial

relational assumptions, which will be removed by the function `filter` as shown in the rule [FV−CALL].

Firstly, the relational assumption $\rho \wedge \theta_a \Rightarrow \theta_c$ is trivial for any θ_a and θ_c if the context ρ is unsatisfiable. Secondly, the assumptions $\rho \wedge \texttt{Loop} \Rightarrow \theta_c$ and $\rho \wedge \texttt{MayLoop} \Rightarrow \theta_c$ are trivially valid for any program state ρ because the constraints `Loop` and `MayLoop` can accept any temporal constraints in the RHS. Finally, $\rho \wedge \theta_a \Rightarrow \texttt{Term}\ M$ is trivial if the callee n, whose termination is denoted by the temporal constraint $\texttt{Term}\ M$, and the caller m are *not* in the same strongly (mutually) connected component (scc for short) of the program's call graph.

Note that assumptions of the form $\rho \wedge \texttt{U}^1_{\texttt{pr}}(\overline{v_1}) \Rightarrow \texttt{U}^2_{\texttt{pr}}(\overline{v_2})$ are not trivial for any caller m and callee n. However, when m and n are in two different scc groups, this kind of assumptions can be avoided if we do a bottom-up verification and inference in which the (non-)termination of the callee n is inferred and the unknown $\texttt{U}^2_{\texttt{pr}}(\overline{v_2})$ is instantiated before the caller m is processed. This mechanism allows us to aggressively resolve trivial assumptions.

For each method declaration, we collect a set of relational assumptions $\mathcal{S}$ during the verification of its body, and another set of relational assumptions $\mathcal{T}$ at the method's exit points via the entailment for proving the post-condition, as shown in the rule below:

$$\frac{\vdash \{\Psi_{\text{Pre}}\}\ e\ \{\Psi, \mathcal{S}\} \quad \Psi \vdash \Phi_{\text{Post}} \leadsto (\Psi_r, \mathcal{T})}{t_0\ mn(\overline{t\ v})\ (\Psi_{\text{Pre}}, \Phi_{\text{Post}})\ \{e\}\ \{\mathcal{S}, \mathcal{T}\}}\ [\textbf{FV−METH}]$$

The termination and non-termination inference engine is invoked when a

```
1:  procedure solve(𝓜)
    // 𝓜 = {t_i mn_i(t v) (U^i_pr, U^i_po) {e} {𝒮_i, 𝒯_i} | 1≤i≤n}
2:      Θ ← {U^i_pr ≡ U^i_pr, U^i_po ≡ U^i_po | 1≤i≤n}
        // Initial defns for unknown pre/post predicates
3:      for each mn_i in 𝓜 do
4:          β ← syn_base(𝒮_i, 𝒯_i)
5:          Θ ← refine_base(Θ, U^i_pr, U^i_po, β)
6:      𝒮 ← ⋃𝒮_i; 𝒯 ← ⋃𝒯_i; iter ← 0
7:      𝒮 ← spec_relass(𝒮, Θ); 𝒯 ← spec_relass(𝒯, Θ)
8:      𝒢 ← reach_graph(𝒮)
        // Reachability graph for unknown pre-predicates
9:      for each scc in 𝒢 do
10:         (r, Θ) ← TNT_analysis(𝒢, scc, 𝒯, Θ)
11:         if ¬r ∧ iter < MAX_ITER then iter++; goto 7
12:         if iter ≥ MAX_ITER then break
13:         𝒯 ← spec_relass(𝒯, Θ)
14:         𝒢 ← graph_update(𝒢, Θ)
15:     return finalize(Θ)
```

Figure 4-5. Overall Inference Algorithm

whole group of mutually recursive methods are verified and their sets of relational assumptions are collected, as shown in the rule [FV−INF] below.

[FV−INF]

$$\frac{M_{scc} = \{t_i^0\ mn_i(\overline{t_i\ v_i})\ (\mathtt{U}^i_{\mathtt{pr}}(\overline{v_i}), \mathtt{U}^i_{\mathtt{po}}(\overline{v_i}))\ \{e\}\ \{\mathcal{S}_i, \mathcal{T}_i\} \mid 1{\le}i{\le}n\}}{M_{scc} \rightsquigarrow \mathtt{solve}(M_{scc})}$$

The `solve` procedure infers definitions for unknown temporal predicates and will be depicted in detail next.

4.5 A Mechanism for Termination and Non-Termination Inference

This section is devoted to the `solve` procedure used to infer the definitions for the unknown pre/post-predicates, based on the set of relational assumptions

generated by Hoare-style verification. The overall algorithm is shown in Fig. 4-5.

In this algorithm, Θ is used to store the set of definitions inferred thus far for the unknown temporal predicates. Since a key idea of our inference mechanism is *case analysis* that incrementally separates the terminating and non-terminating behaviors of the analyzed methods, the definition for each unknown predicate might be split into multiple scenarios, for which termination is either known or unknown.

Definition 8 (Unknown Temporal Predicates) *During the inference process, the definitions for a pair of unknown pre-predicate* $\mathtt{U}_{\mathtt{pr}}(\overline{v})$ *and post-predicate* $\mathtt{U}_{\mathtt{po}}(\overline{v})$ *are of the form* $\mathtt{U}_{\mathtt{pr}}(\overline{v}) \equiv \bigvee_i(\pi_i \wedge \theta^i_{\mathtt{pr}})$ *and* $\mathtt{U}_{\mathtt{po}}(\overline{v}) \equiv \bigwedge_i(\pi_i \Rightarrow \theta^i_{\mathtt{po}})$ *where each* $\theta^i_{\mathtt{pr}}$ *is either a known or unknown pre-predicate and* $\theta^i_{\mathtt{po}}$ *is either* $\mathtt{true}$, $\mathtt{false}$ *or an unknown post-predicate. The set of guards* $\{\pi_1, \ldots, \pi_n\}$ *must be (1)* feasible, *i.e.* $\forall i \cdot \mathtt{SAT}(\pi_i)$, *(2)* exclusive, *i.e.* $\forall i, j \cdot i{\neq}j \Rightarrow \mathtt{UNSAT}(\pi_i{\wedge}\pi_j)$, *and (3)* exhaustive, *i.e.* $\pi_1{\vee}\pi_2{\vee}\ldots{\vee}\pi_n \equiv \mathtt{true}$.

The initial form of each unknown predicate is the predicate itself with guard condition $\mathtt{true}$, e.g. $\mathtt{U}_{\mathtt{pr}}(\overline{v}) \equiv \mathtt{true}{\wedge}\mathtt{U}_{\mathtt{pr}}(\overline{v})$. At the end of the analysis, all $\theta^i_{\mathtt{pr}}$ and $\theta^i_{\mathtt{po}}$ become known.

The inference deals with two groups of temporal relational assumptions collected by rule [TNT−METH], namely

1. Pre-assumptions $\mathcal{S}$ collected when proving preconditions at method calls. They can be used to infer *(i)* ranking functions for termination proving, and *(ii) temporal reachability* graph that guides our search for proving termination vs. non-termination.

2. Post-assumptions $\mathcal{T}$ collected when proving postconditions contain

information about unknown post-predicates. They can be used to infer *(i)* termination base cases, *(ii)* inductive unreachability to prove non-termination or *(iii)* new conditions for the case analysis.

The algorithm in Fig. 4-5 first derives the base case of each analyzed method (line 4), and then refines the definitions of unknown temporal predicates in Θ with these newly inferred cases (line 5). After updating the set of relational assumptions (line 7), our algorithm (re-)builds the *temporal reachability* graph $\mathcal{G}$ from the latest $\mathcal{S}$ (line 8).

For each scc of the graph $\mathcal{G}$ in the bottom-up topological order, the analysis attempts to prove either termination or non-termination or to infer new cases for case-splitting and then updates the set Θ with the inferred result (line 10). If every unknown temporal predicate corresponding to the current scc is resolved into known predicates, the inference continues with the next sccs after updating the post-assumptions in $\mathcal{T}$ (line 13) and the graph $\mathcal{G}$ (line 14) with the new inferred known predicates. Otherwise, it restarts the core algorithm (line 11) with the updated Θ, whose elements have been refined into new sub-cases.

The algorithm halts when every unknown predicate has been resolved or the number of iterations reaches the maximum `MAX_ITER` pre-set by users. In the latter case, the remaining unknown predicates in Θ will be marked as `MayLoop` by an auxiliary procedure `finalize`. Next we will explain each inference step in some detail.

4.5.1 Inferring Base Case Termination

Identifying the conditions for base-case termination is an important first step before any other analyses. Formally:

Definition 9 (Base Case Pre-Condition) *Each base case termination precondition of a method must satisfy the following three conditions:*

(i) Its method's exit is reachable.

(ii) No mutually recursive method call is met in executions starting from this pre-condition.

(iii) All other method calls encountered from this pre-condition must have been proven to terminate.

While a syntactic-based approach that identifies base-case termination from its control-flow may be sufficient, we propose a semantics-based approach which infers a method's base case precondition from the two sets of assumptions $\mathcal{S}$ and $\mathcal{T}$ collected from the method, as follows:

$$\frac{\rho = \bigvee\{(\rho_i/\{\overline{v}\}) \mid \rho_i \wedge \mathtt{U_{pr}}(\overline{v}) \Rightarrow \theta_c^i \in \mathcal{S}\} \qquad \varrho = \bigvee\{(\beta_j/\{\overline{v}\}) \mid \beta_j \wedge \mathtt{true} \Rightarrow \mathtt{U_{po}}(\overline{v}) \in \mathcal{T}\}}{\mathtt{syn_base}(\mathcal{S}, \mathcal{T}) = \varrho \wedge \neg\rho}$$

where $\rho/\{\overline{v}\} \equiv \exists(\mathtt{FV}(\rho){-}\{\overline{v}\}) \cdot \rho$. Using our running example, we have $\mathcal{S} = \{a_2^0\}$ and $\mathcal{T} = \{a_1^0, a_3^0\}$:

(a_1^0) $\mathtt{x{<}0} \wedge \mathtt{true} \Rightarrow \mathtt{U_{po}}(\mathtt{x}, \mathtt{y})$

(a_2^0) $\mathtt{x{\geq}0} \wedge \mathtt{x'{=}x{+}y} \wedge \mathtt{y'{=}y} \wedge \mathtt{U_{pr}}(\mathtt{x}, \mathtt{y}) \Rightarrow \mathtt{U_{pr}}(\mathtt{x'}, \mathtt{y'})$

(a_3^0) $\mathtt{x{\geq}0} \wedge \mathtt{x'{=}x{+}y} \wedge \mathtt{y'{=}y} \wedge \mathtt{true} \wedge \mathtt{U_{po}}(\mathtt{x'}, \mathtt{y'}) \Rightarrow \mathtt{U_{po}}(\mathtt{x}, \mathtt{y})$,

Each post-assumption $\beta_j \wedge \mathtt{true} \Rightarrow \mathtt{U_{po}}(\overline{v}) \in \mathcal{T}$, whose antecedent does not contain any unknown post-predicate, capture a potential base-case termination condition. Due to over-approximation, the actual base-case condition (over the

method's parameters $\overline{v}$) must be formed by such conditions ($\bigvee \beta_j$), conjoined with the negation of contexts ($\neg\rho$) for the recursive calls. By identifying the base-case condition in $\{a_1^0\}$ and conditions for recursive pre-assumption in $\{a_2^0\}$, we can precisely infer $\texttt{syn_base}(\mathcal{S},\mathcal{T}) = x{<}0{\wedge}\neg(x{\geq}0)$.

With the inferred base case $\beta = \texttt{syn_base}(\mathcal{S},\mathcal{T})$ (line 4), we can now invoke the procedure `refine_base` (line 5) to refine (or specialize) the unknown predicates $\texttt{U}_{\texttt{pr}}(\overline{v})$ and $\texttt{U}_{\texttt{po}}(\overline{v})$, before updating their definitions in Θ (via the operator $\oplus$) as shown below.

$$\frac{\begin{array}{c}\bigvee \mu_i \equiv \neg\beta \\ \Delta_{\texttt{pr}} = (\texttt{U}_{\texttt{pr}}(\overline{v}) \equiv \bigvee(\mu_i \wedge \texttt{U}^i_{\texttt{pr}}(\overline{v})) \vee (\beta \wedge \texttt{Term})) \\ \Delta_{\texttt{po}} = (\texttt{U}_{\texttt{po}}(\overline{v}) \equiv \bigwedge(\mu_i \Rightarrow \texttt{U}^i_{\texttt{po}}(\overline{v}))) \\ \Omega = \bigcup\{\texttt{U}^i_{\texttt{pr}}(\overline{v}) \equiv \texttt{U}^i_{\texttt{pr}}(\overline{v}), \texttt{U}^i_{\texttt{po}}(\overline{v}) \equiv \texttt{U}^i_{\texttt{po}}(\overline{v})\}\end{array}}{\texttt{refine_base}(\Theta, \texttt{U}_{\texttt{pr}}(\overline{v}), \texttt{U}_{\texttt{po}}(\overline{v}), \beta) = \Theta \oplus (\{\Delta_{\texttt{pr}}, \Delta_{\texttt{po}}\} \cup \Omega)}$$

Since the method's termination status in the remaining condition $\mu = \neg\beta$ is unknown. In the new definitions of $\texttt{U}_{\texttt{pr}}(\overline{v})$ and $\texttt{U}_{\texttt{po}}(\overline{v})$, each pair of fresh predicates $\texttt{U}^i_{\texttt{pr}}(\overline{v})$ and $\texttt{U}^i_{\texttt{po}}(\overline{v})$ is associated with a disjunct μ_i in the disjunctive normal form of μ. For our running example, this refinement leads to:

$$\begin{array}{lllll}\texttt{U}_{\texttt{pr}}(\texttt{x},\texttt{y}) \equiv & \texttt{x}{<}\texttt{0} \wedge \texttt{Term} & \vee & \texttt{x}{\geq}\texttt{0} \wedge \texttt{U}^1_{\texttt{pr}}(\texttt{x},\texttt{y}) \\ \texttt{U}_{\texttt{po}}(\texttt{x},\texttt{y}) \equiv & \texttt{x}{<}\texttt{0} \Rightarrow \texttt{true} & \wedge & \texttt{x}{\geq}\texttt{0} \Rightarrow \texttt{U}^1_{\texttt{po}}(\texttt{x},\texttt{y})\end{array}$$

After the unknown predicates have been updated with base-case termination conditions, we transform the sets of relation assumptions by using the procedure `spec_relass` (line 7) described next.

4.5.2 Specializing Relational Assumptions

Whenever some unknown predicates in Θ receive new definitions, our inference algorithm will update its sets of relational assumptions with the procedure spec_relass. Its first parameter is a set of relational assumptions. Its second parameter Θ contains the definitions of unknown predicates.

For each relational assumption with unknown predicates, the procedure spec_relass finds the current definitions of these unknown predicates in Θ and substitutes them directly into the assumption. As the definition of each unknown predicate consists of exclusive and complete guards, we can further split each substituted assumptions into multiple specialized assumptions. We show below just one example where spec_relass is called with a new pre-assumption with two unknown predicates.

$$\frac{\mathtt{U}^1_{\mathtt{pr}}(\overline{v_1}) \equiv \bigvee_{i=1}^{n}(\rho_{1i} \wedge \theta^{1i}_{\mathtt{pr}}) \in \Theta \qquad \mathtt{U}^2_{\mathtt{pr}}(\overline{v_2}) \equiv \bigvee_{j=1}^{m}(\rho_{2j} \wedge \theta^{2j}_{\mathtt{pr}}) \in \Theta \\ \mathcal{C} = \{\rho \wedge \rho_{1i} \wedge \rho_{2j} \wedge \theta^{1i}_{\mathtt{pr}} \Rightarrow \theta^{2j}_{\mathtt{pr}} \mid 1 \leq i \leq n, 1 \leq j \leq m\}}{\mathtt{spec_relass}(\{\rho \wedge \mathtt{U}^1_{\mathtt{pr}}(\overline{v_1}) \Rightarrow \mathtt{U}^2_{\mathtt{pr}}(\overline{v_2})\} \cup \mathcal{S}, \Theta) = \\ \mathcal{C} \cup \mathtt{spec_relass}(\mathcal{S}, \Theta)}$$

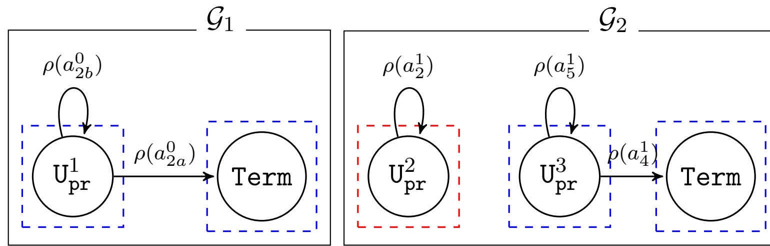
For our running example, the relational assumption (a^0_2) was specialized by its earlier partial definition into two more specialized assumptions: (a^0_{2a}) and (a^0_{2b}).

4.5.3 Resolving Temporal Reachability Graph

The core of our inference algorithm (in Fig. 4-5) incrementally resolves the unknown predicates present in the (specialized) relational pre-assumptions. If its attempt fails, it would also derive conditions for the next case analysis. This core algorithm uses a *reachability graph* $\mathcal{G}$, constructed from pre-predicates in

$\mathcal{S}$, to guide its proof search. Formally:

Definition 10 (Temporal Reachability Graph) *Given a set of pre-assumptions $\mathcal{S}$, a* temporal reachability graph $\mathcal{G} = (V, E)$ *is constructed from a set of vertices V and a set of labeled edges E, as follows. For each pre-assumption $\rho \wedge \theta_a \Rightarrow \theta_c \in \mathcal{S}$, we add two vertices θ_a and θ_c into V and an edge $(\theta_a, \rho, \theta_c)$ from θ_a to θ_c labeled by ρ into E.*



For example, the two graphs $\mathcal{G}_1$ and $\mathcal{G}_2$ are built for the inference of the running example. $\mathcal{G}_1$ is constructed from pre-assumptions (a^0_{2a}) and (a^0_{2b}) obtained after base case inference. The edges of $\mathcal{G}_1$ are labeled by $\rho(a^0_{2a})$ and $\rho(a^0_{2b})$, the contexts in (a^0_{2a}) and (a^0_{2b}) resp., e.g. $\rho(a^0_{2b}) \equiv (\mathtt{x}{\geq}0 \wedge \mathtt{x}'{=}\mathtt{x}{+}\mathtt{y} \wedge \mathtt{y}'{=}\mathtt{y} \wedge \mathtt{x}'{\geq}0)$. The self-loop edge on node $\mathtt{U}^1_{\mathtt{pr}}$ denotes the case when the latest values of program variables (*i.e.* $\mathtt{x}'{\geq}0$), are still in the same loop condition as their initial values ($\mathtt{x}{\geq}0$). The edge from $\mathtt{U}^1_{\mathtt{pr}}$ to `Term` indicates the base case is reached when $\mathtt{x}'{<}0$. Similarly, the graph $\mathcal{G}_2$ is constructed from pre-assumptions (a^1_4), (a^1_5) and (a^1_2) after a new case split $\mathtt{y}{\geq}0$ and $\mathtt{y}{<}0$ has been inferred.

Our core algorithm firstly partitions $\mathcal{G}$ into strongly connected components (scc), (*e.g.* dashed boxes in $\mathcal{G}_1$ and $\mathcal{G}_2$), whereby each unknown temporal predicate denotes an unknown behavior. Moreover, this unknown predicate is mutually dependent on the other predicates in the same scc. Using a bottom-up approach, the inference mechanism processes each scc in a topologically sorted

```
16: procedure TNT_analysis(G, scc, T, Θ)
17:     r ← true
18:     O ← scc_succ(scc, G)
19:     if O = {} then
20:         if scc has one node U_pr without cyclic edge then
21:             U_po ← the post-pred corresponding to U_pr
22:             Θ ← Θ ⊕ {U_pr ≡ Term, U_po ≡ true})
23:         else (r, Θ) ← prove_NonTerm(scc, T, Θ)
24:     else if ∀θ ∈ O · θ ≡ Term [ē] then
25:         (r, Θ) ← prove_Term(G, scc, Θ)
26:         if ¬r then (r, Θ) ← prove_NonTerm(scc, T, Θ)
27:     else (r, Θ) ← prove_NonTerm(scc, T, Θ)
28:     return (r, Θ)
```

Figure 4-6. Core TNT Inference Algorithm

order. With this approach, termination and non-termination proofs for *phase-change* programs [43] and that for mutual recursion are easily supported.

Definition 11 (scc's successors) *Given a graph $\mathcal{G}$, the outside successors of a scc in $\mathcal{G}$ is the set of all successors of any vertex in this scc but excluding the scc itself,*

$$\texttt{scc_succ}(scc, \mathcal{G}) = \bigcup\{\texttt{succ}(v, \mathcal{G}) \mid v \in scc\} \setminus scc$$

where $\texttt{succ}(v, \mathcal{G})$ *returns all successors of the vertex* v.

Our core algorithm, named TNT_analysis, for manipulating each scc is outlined in Fig. 4-6. After this analysis, if all vertices in the scc can be resolved as known temporal predicates, our procedure returns the result r=true. Otherwise, it returns r=false to allow inference mechanism to restart for the next iteration (from line 7 in Fig. 4-5). Moreover, upon termination of this procedure, some unknown pre- and post-predicates in store Θ, are updated with their new definitions.

Our procedure (Fig. 4-6) uses the set $\mathcal{O}$ of the scc's successors to determine whether termination proof (by sub-procedure prove_Term), or non-termination

```
29: procedure prove_Term(𝒢, scc, Θ)
30:     𝒞 ← {gen(e)|e≡(U^i_pr, ρ, U^j_pr)∈𝒢(E) ∧ U^i_pr, U^j_pr∈scc}
31:     Γ ← syn_rank(𝒞)
32:     if Γ ≠ {} then
33:         𝒟 ← subst_rank(scc, Γ)
34:         return (true, Θ ⊕ 𝒟)
35:     else  return (false, Θ)
```

Figure 4-7. Procedure for Proving Termination over a scc

proof (by sub-procedure `prove_NonTerm`), should be applied to resolve the unknown temporal predicates. Specifically, when the scc has only one unknown node $\mathtt{U}_{\mathtt{pr}}$ without any cyclic edge and successor (line 20), we resolve the unknown pre-predicate $\mathtt{U}_{\mathtt{pr}} \equiv \mathtt{Term}$ and its corresponding post-predicate $\mathtt{U}_{\mathtt{po}} \equiv \mathtt{true}$ for trivial termination (line 22). Moreover, when the set $\mathcal{O}$ is nonempty, the procedure invokes `prove_Term` with ranking function synthesis only if every element of $\mathcal{O}$ is a known $\mathtt{Term}\ [\bar{\mathtt{e}}]$ predicate (line 24-25).

For the running example, the procedure applies termination proofs for the left scc in $\mathcal{G}_1$ and the middle scc in $\mathcal{G}_2$. For the left scc in $\mathcal{G}_2$, it applies a non-termination proof directly. In the next sub-sections, we present the sub-procedures for proving termination and non-termination over a scc.

4.5.4 Inferring Ranking Function

For proving termination on a scc, we implement the procedure `prove_Term` (sketched in Fig. 4-7) to find a linear ranking function for each unknown pre-predicate in this scc by using a constraint-based technique [69,136] with Farkas' lemma [137].

Initially, we create a unique ranking function template for each unknown pre-predicate $\mathtt{U}_{\mathtt{pr}}(v_1, .., v_n) \in scc$ by the procedure `gen_rank`, defined as

$$\mathtt{gen_rank}(\mathtt{U}_{\mathtt{pr}}(v_1, \ldots, v_n)) = c_0 + \textstyle\sum_{i=1}^{n} c_i v_i$$

where $c_0, c_1, \ldots, c_n$ are unknown coefficients of the ranking function. Next, we generate a set of constraints over these ranking functions from every edge in $\mathcal{G}$ that connects two nodes in the scc (line 30). That is, given an edge $e \equiv (\mathtt{U}^{i}_{\mathtt{pr}}(\overline{v_i}), \rho, \mathtt{U}^{j}_{\mathtt{pr}}(\overline{v_j})) \in \mathcal{G}(E)$ s.t. $\mathtt{U}^{i}_{\mathtt{pr}}(\overline{v_i}), \mathtt{U}^{j}_{\mathtt{pr}}(\overline{v_j}) \in scc$, the constraint generated from it is

$$\frac{r_i(\overline{v_i}){=}\mathtt{gen_rank}(\mathtt{U}^{i}_{\mathtt{pr}}(\overline{v_i})) \quad r_j(\overline{v_j}){=}\mathtt{gen_rank}(\mathtt{U}^{j}_{\mathtt{pr}}(\overline{v_j}))}{\mathtt{gen}(e) = \forall \overline{v_i}, \overline{v_j} \cdot \rho \Rightarrow (r_i(\overline{v_i}) > r_j(\overline{v_j}) \wedge r_i(\overline{v_i}) \geq 0)}$$

This constraint indicates that the ranking function $r_i(\overline{v_i})$ is bounded and decreasing across a (mutually) recursive method call under the call context ρ. For example, the constraint generated from the middle scc in $\mathcal{G}_2$ is

$$\forall \mathtt{x}, \mathtt{y} \cdot \mathtt{x}{\geq}0 \wedge \mathtt{x}'{=}\mathtt{x}{+}\mathtt{y} \wedge \mathtt{y}'{=}\mathtt{y} \wedge \mathtt{x}'{\geq}0 \wedge \mathtt{y}{<}0 \Rightarrow$$
$$r(\mathtt{x}, \mathtt{y}){>}r(\mathtt{x}', \mathtt{y}') \wedge r(\mathtt{x}, \mathtt{y}){\geq}0$$

which is then solved by syn_rank to obtain the ranking function $r(\mathtt{x}, \mathtt{y}) = \mathtt{x}$. The method syn_rank (line 31) solves the generated constraints by applying Farkas' lemma on them to obtain another set of constraints over their unknown coefficients, which can be solved by a nonlinear solver, such as [88], to get the actual values of these unknowns. The result is a substitution Γ which maps each unknown coefficient to its actual value.

If the ranking function synthesis succeeds, we update each unknown pre-predicate in this scc into Term with an actual ranking function (line 34 in Fig. 4-7). Otherwise, we prove non-termination on this scc (line 26 in Fig. 4-6). The ranking function for a pre-predicate can be obtained by applying the substitution Γ to its ranking function template, as shown below. Note that

```
36: procedure prove_NonTerm(scc, T, Θ)
37:     for each U^i_pr ∈ scc do
38:         T_i ← filter_rel(T, U^i_pr)
39:         C_i ← ⋃{abd_inf(t) | t ∈ T_i}
40:     r ← ⋀_i(C_i ≠ {} ∧ ∀c ∈ C_i · (c ≡ true))
41:     if r then D ← {U^i_pr ≡ Loop, U^i_po ≡ false | U^i_pr∈scc}
42:     else D ← ⋃_i subst_unk(C_i, U^i_pr, U^i_po)
43:     return (r, Θ ⊕ D)
```

Figure 4-8. Proc. for Proving Non-Termination over a scc

$\mathtt{subst_rank}(\{\},\Gamma) = \{\}$.

$$\frac{\mathtt{r} = \Gamma(\mathtt{gen_rank}(\mathtt{U_{pr}}(\overline{v}))) \quad \mathtt{U_{pr}}(\overline{v}) \equiv \mathtt{Term}\,[\mathtt{r}] \quad \mathtt{U_{po}}(\overline{v}) \equiv \mathtt{true}}{\mathtt{subst_rank}(\{\mathtt{U_{pr}}(\overline{v})\} \cup \mathcal{U}, \Gamma) = \{\mathtt{U_{pr}}(\overline{v}), \mathtt{U_{po}}(\overline{v})\} \cup \mathtt{subst_rank}(\mathcal{U}, \Gamma)}$$

We also support the synthesis of lexicographic ranking functions, details are omitted for simplicity of presentation.

4.5.5 Inferring Inductive Unreachability

Procedure $\mathtt{prove_NonTerm}(scc, \mathcal{T}, \Theta)$ finds non-termination on a scc by unreachability of its post-predicates in $\mathcal{T}$. For each $\mathtt{U_{pr}}(\overline{v}){\in}scc$, the method $\mathtt{filter_rel}(\mathcal{T}, \mathtt{U_{pr}})$ selects a set of post-assumptions $\mathcal{T}_s \subseteq \mathcal{T}$ such that their RHS post-predicate is the corresponding $\mathtt{U_{po}}(\overline{v})$. The general form of such post-assumptions is either:

1. $\rho{\wedge}\mathtt{true} \Rightarrow (\mu{\Rightarrow}\mathtt{U_{po}}(\overline{v}))$, or
2. $\rho{\wedge} \bigwedge(\eta_i{\Rightarrow}\mathtt{false}) \wedge \bigwedge(\mu_j{\Rightarrow}\mathtt{U^j_{po}}(\overline{v_j})) \Rightarrow (\mu{\Rightarrow}\mathtt{U_{po}}(\overline{v}))$.

These post-assumptions capture possible non-termination of its method due to predicate $\mathtt{U_{po}}(\overline{v})$ being unknown, under the condition $\rho \wedge \mu$ where the context ρ is satisfiable. The first post-assumption describes a base-case

scenario. In order to ensure unreachability of its post-predicate, we must check that $\rho \wedge \mu$ is unsatisfiable. The second post-assumption shows that we can meet a non-terminating method call (with the postcondition `false`) if the condition η_i is satisfied by $\rho\wedge\mu$. In addition, we can meet a (mutually) recursive call whose termination is unknown if μ_j is satisfied, and thus the respective pre-predicate of $\mathtt{U}^j_{\mathtt{po}}(\overline{v_j})$ also belongs to the analyzed scc. We call the conditions η_i, μ_j and μ *potential non-termination conditions* as they could lead to an actual non-termination.

By induction, we prove that a caller is definitely non-terminating under a condition μ, assuming that one of its callee is definitely non-terminating under the same condition. Given a set of post-assumptions $\mathcal{T}_s$, we prove that if each unknown post-predicate in their LHS is `false` then every unknown post-predicate in their RHS is also `false`. This is done by the procedure `abd_inf` (line 39).

- For $t \equiv \rho\wedge\mathtt{true} \Rightarrow (\mu\Rightarrow\mathtt{U}_{\mathtt{po}}(\overline{v}))$, $\mathtt{U}_{\mathtt{po}}(\overline{v})\equiv\mathtt{false}$ iff $\rho\wedge\mu$ is unsatisfiable. So the proof succeeds and $\mathtt{abd_inf}(t)$ returns $\{\mathtt{true}\}$ if $\vdash \rho\wedge\mu\Rightarrow\mathtt{false}$.
- For $t\equiv\rho\wedge\bigwedge(\eta_i\Rightarrow\mathtt{false})\wedge\bigwedge(\mu_j\Rightarrow\mathtt{U}^j_{\mathtt{po}}(\overline{v_j}))\Rightarrow(\mu\Rightarrow\mathtt{U}_{\mathtt{po}}(\overline{v}))$, given that $\forall j\cdot\mathtt{U}^j_{\mathtt{po}}(\overline{v_j})\equiv\mathtt{false}$, we have $\mathtt{U}_{\mathtt{po}}(\overline{v})\equiv\mathtt{false}$ iff $\rho\wedge\mu\Rightarrow\bigvee\eta_i\vee\bigvee\mu_j$, this means that under the precondition μ, at least one of the callees' non-termination conditions is satisfied, so that the caller is also non-terminating. The proof succeeds and $\mathtt{abd_inf}(t)$ returns $\{\mathtt{true}\}$ if $\vdash \rho\wedge\mu\Rightarrow\bigvee\eta_i\vee\bigvee\mu_j$.

If the proof succeeds for all pre-predicates in scc (signified by r in line 40), we mark the unknown termination status as definitely non-terminating. This procedure thus refines, where possible, each unknown pre-predicate as $\mathtt{U}_{\mathtt{pr}}(\overline{v}) \equiv \mathtt{Loop}$ and its post-predicate as $\mathtt{U}_{\mathtt{po}}(\overline{v}) \equiv \mathtt{false}$ (line 41) and updates Θ

before returning $(\texttt{true}, \Theta)$.

For our running example, (a_1^0) and (a_3^0) from $\mathcal{T}$ would cause $\texttt{prove_NonTerm}(scc, \mathcal{T}, \Theta)$ to return false, but provide an abductive condition $\texttt{y}{\geq}\texttt{0}$ that facilitates case-splitting (see next sub-section). In contrast, (a_3^1) would be used to show that $\texttt{U}^2_{\texttt{po}}(\texttt{x}, \texttt{y})$ is inductively $\texttt{false}$ (or unreachable).

4.5.6 Abductive Case-Splitting

If non-termination proving fails, the method $\texttt{abd_inf}$ abductively infers new sub-conditions from the failed proof to refine the potential non-termination condition by case-split.

In the case $t \equiv \rho{\wedge}\texttt{true} \Rightarrow (\mu{\Rightarrow}\texttt{U}_{\texttt{po}}(\overline{v}))$, if the proof fails, *i.e.* $\nvdash \rho \wedge \mu \Rightarrow \texttt{false}$, $\texttt{abd_inf}(t)$ simply returns $\{\}$ as any condition that makes the entailment to hold would contradict with the antecedent $\rho{\wedge}\mu$.

If $t{\equiv}\rho{\wedge}\bigwedge(\eta_i{\Rightarrow}\texttt{false}){\wedge}\bigwedge(\mu_j{\Rightarrow}\texttt{U}^j_{\texttt{po}}(\overline{v_j})){\Rightarrow}(\mu{\Rightarrow}\texttt{U}_{\texttt{po}}(\overline{v}))$, and the proof fails, *i.e.* $\nvdash \rho{\wedge}\mu{\Rightarrow}\bigvee\eta_i{\vee}\bigvee\mu_j$, $\texttt{abd_inf}(t)$ returns a set of conditions $\mathcal{C}_t$ such that: for each $\beta_k \in \{\eta_i\}{\cup}\{\mu_j\}$ s.t. $\rho{\wedge}\eta{\wedge}\beta_k$ is satisfiable, there exists $\alpha_k{\in}\mathcal{C}_t$ such that *(i)* $\rho{\wedge}\eta{\wedge}\alpha_k$ is satisfiable and *(ii)* $\vdash \rho{\wedge}\eta{\wedge}\alpha_k{\Rightarrow}\beta_k$. That is, if the potential non-termination condition μ of the caller is strengthened by α_k then the (potential) non-termination condition β_k of a callee is satisfied.

For each condition β_k, the solution $\alpha_k \equiv \beta_k$ is a trivial but the weakest solution for α_k. For a more effective case-split, we aim to derive a stronger abductive condition α_k. By the same constraint-based approach used for the ranking function synthesis, we assume the template $\alpha_k \equiv c_0{+}\sum_{i=1}^{n} c_i v_i{\geq}0$, where $v_1, .., v_n{\equiv}\overline{v}$ and $c_0, .., c_n$ are unknown coefficients. We might solve these unknown coefficients with additional optimal constraints, *e.g.* the number of zero-coefficients is maximum, so that we can obtain a better solution with

minimum number of program variables.

Given a set of collective abductive conditions $\mathcal{C}$, the procedure $\texttt{subst_unk}$ (line 42) refines the pair of $(\texttt{U}_{\texttt{pr}}(\overline{v}), \texttt{U}_{\texttt{po}}(\overline{v}))$ with these new sub-cases for the update of Θ.

$$\frac{\begin{array}{c}\texttt{split}(\mathcal{C}) = \{\mu_j\}_{j=1}^{m} \quad \mu_{m+1} = \neg\mu_1 \wedge \ldots \wedge \neg\mu_m \\ \Delta_1{=}(\texttt{U}_{\texttt{po}}(\overline{v}){\equiv} \bigwedge(\mu_j{\Rightarrow}\texttt{U}^j_{\texttt{po}}(\overline{v}))) \;\; \Delta_2{=}(\texttt{U}_{\texttt{pr}}(\overline{v}){\equiv} \bigvee(\mu_j{\wedge}\texttt{U}^j_{\texttt{pr}}(\overline{v}))) \\ \Omega = \bigcup\{\texttt{U}^j_{\texttt{pr}}(\overline{v}) \equiv \texttt{U}^j_{\texttt{pr}}(\overline{v}), \texttt{U}^j_{\texttt{po}}(\overline{v}) \equiv \texttt{U}^j_{\texttt{po}}(\overline{v})\}\end{array}}{\texttt{subst_unk}(\mathcal{C}, \texttt{U}_{\texttt{pr}}(\overline{v}), \texttt{U}_{\texttt{po}}(\overline{v})) = \{\Delta_1, \Delta_2\} \cup \Omega}$$

As the conditions in $\mathcal{C}$ might be overlapping, we use the function $\texttt{split}$ defined below to partition these conditions into the new set of mutually exclusive conditions $\{\mu_j\}_{j=1}^{m}$ such that $\bigvee\mathcal{C} \equiv \bigvee\{\mu_j\}$. We also add into the new set the condition $\mu_{m+1} = \neg\mu_1 \wedge \ldots \wedge \neg\mu_m$, if it is satisfiable, to cover the missing case, so that $\{\mu_j\}_{j=1}^{m+1}$ is complete. Note $\texttt{split}(\{\}) = \{\}$.

$$\frac{\begin{array}{c}\mathcal{C}_2{=}\texttt{split}(\mathcal{C}_1) \quad \mathcal{C}_3{=}\{c_i \mid c_i{\in}\mathcal{C}_2{\wedge}\texttt{UNSAT}(c_i{\wedge}c_1)\} \\ \mathcal{C}_4 = \{c_i \mid c_i{\in}\mathcal{C}_2{\wedge}\texttt{SAT}(c_i{\wedge}c_1)\} \quad c = c_1{\wedge}\bigwedge\{\neg c_i \mid c_i{\in}\mathcal{C}_4\} \\ \mathcal{C}_5 = \{c_i{\wedge}c_1 \mid c_i{\in}\mathcal{C}_4\} \cup \{c_i{\wedge}\neg c_1 \mid c_i{\in}\mathcal{C}_4{\wedge}\texttt{SAT}(c_i{\wedge}\neg c_1)\}\end{array}}{\texttt{split}(\{c_1\}{\cup}\mathcal{C}_1) = \text{if } \texttt{SAT}(c) \text{ then } \{c\} \cup \mathcal{C}_3 \cup \mathcal{C}_5 \text{ else } \mathcal{C}_3 \cup \mathcal{C}_5}$$

4.6 Experiments

We have built the proposed inference mechanism on top of the existing verification system HIPTNT that can also verify both termination and non-termination specifications given by users. Our inference system, namely HIPTNT+ can be run online at

Benchmark	crafted					crafted-lit					numeric					memory-alloca					Total			
	Y	N	U	T/O	Time	Y	N	U	T/O	Time	Y	N	U	T/O	Time	Y	N	U	T/O	Time	Y	N	U	T/O
AProVE	19	0	17	4	169.7	104	0	24	22	1913.1	68	0	0	0	511.8	69	0	10	7	3385.9	260	0	51	33
ULTIMATE	21	15	3	1	561.7	112	17	14	7	1080.6	56	0	9	3	382.5	41	6	7	32	3073.2	230	38	33	43
HIPTNT+	19	13	8	0	39.2	114	19	17	0	109.6	61	0	7	0	29.1	66	6	14	0	201.5	260	38	46	0

Figure 4-9. Termination Outcomes on SV-COMP 2015's C Programs.

http://loris-7.ddns.comp.nus.edu.sg/~project/hiptnt/plus/

To evaluate our approach, we compare our system against two state-of-the-art systems, ULTIMATE [74] and AProVE [64]. The latter is a recent winner for several categories of problems in the annual Termination Competition 2014. We made our preliminary comparison based on a set of numerical and pointer-based C programs selected from four benchmarks used for the termination category of the Competition on Software Verification (SV-COMP) 2015 [2]. These benchmarks were largely contributed by the teams of AProVE and ULTIMATE. We have excluded 55 programs with arrays and strings from the total 399 programs in 4 benchmarks, since these two aspects[1] have not yet been handled by our specification inference and verification system. The experiments were performed on a machine with the Intel Xeon X5650 (2.67GHz) processor and 24GB of RAM.

In Fig. 4-9, we report the number of programs whose `main` methods' termination or non-termination were proven successfully in columns labeled by Y (for termination) or N (for non-termination), respectively. The columns U (*i.e.* unknown) show the number of programs in which the tools cannot decide whether they are definitely terminating or non-terminating. The number of unsuccessful cases in which the tools give no answer after a timeout is provided in the columns T/O. As in the competition, we set a wall-clock timeout of 300 seconds for the proving process on each program. Finally, the last column in each benchmark presents the total time (excluding timeouts)

[1] These are orthogonal to termination and non-termination reasoning.

```
void loop(int x, int m) {          void main() {
  if (x ≠ m) {                       int x; int m;
    if (x > m) x = 0;                assume (m ≥ 0);
    else x = x + 1;                  loop(x, m);
    loop(x, m); }}                   loop(x, −m); }
```

Figure 4-10. A simple non-terminating program adapted from [48]

each tool took to prove the termination and non-termination of the whole benchmark. In this evaluation, we only report the wall-clock time instead of the consumed CPU time of all the verifier's processes because CPU time of tools executing jobs in parallel, such as AProVE, would be much higher otherwise.

The overall result shows that our HIPTNT+ can efficiently (without any timeout) infer more (non-)termination properties than the other tools. Note that all answers (specifications inferred) that were returned by our tool have been successfully re-verified by an underlying automated verification system. Thus, our tool does not have any false positive nor negative for this set of benchmarks. For the other two analyzers, we also manually check their answers and see that they also do not have any unsoundness.

To illustrate the advantages of our approach, let us highlight two small but challenging examples next.

4.6.1 Tricky While Program

In Fig. 4-10, we use a simple program adapted from [48] to demonstrate how our inference mechanism handles programs with complex termination behavior. Given the input $\mathtt{m}{\geq}0$, the approach in [48] failed to prove the termination of `loop` with a lexicographic ranking function. However, by using a Ramsey-based disjunctive well-founded termination argument [46], their T2 prover can show that this method always terminates under the given precondition. On the other

hand, the tool can also prove that this method does not terminate when $\mathtt{m}<0$. However, it cannot prove the non-termination of `main` under the assumption $\mathtt{m}\geq 0$.

For this example, our modular inference mechanism would construct the following comprehensive summary on the termination and non-termination scenarios of the method `loop`.

$$
\begin{array}{l}
\mathtt{case}\ \{ \\
\quad \mathtt{x} \leq \mathtt{m} \rightarrow \mathtt{requires}\ \mathtt{Term}\ [1, \mathtt{m}-\mathtt{x}]\ \mathtt{ensures}\ \mathtt{true}; \\
\quad \mathtt{x} > \mathtt{m} \rightarrow \mathtt{case}\ \{ \\
\qquad \mathtt{m} < 0 \rightarrow \mathtt{requires}\ \mathtt{Loop}\ \mathtt{ensures}\ \mathtt{false}; \\
\qquad \mathtt{m} \geq 0 \rightarrow \mathtt{requires}\ \mathtt{Term}\ [2]\ \mathtt{ensures}\ \mathtt{true};\ \}\}
\end{array}
$$

This summary can later be used to prove either termination or non-termination of `loop` under different preconditions, not just $\mathtt{m}\geq 0$. Therefore, it facilitates the non-termination proof for `main`. With case-splitting, our inference mechanism segments the terminating behavior of `loop` into two distinct phases: the case when $\mathtt{x}>\mathtt{m} \wedge \mathtt{m}\geq 0$ and its successor phase $\mathtt{x}\leq\mathtt{m}$. In addition, our approach is able to show that this method never terminates when $\mathtt{x}>\mathtt{m} \wedge \mathtt{m}<0$. Such nontermination precondition cannot be derived by other approaches, such as [35, 70], as the `loop`'s body has a disjunctive transition relation.

4.6.2 Phase Change Recursive Programs

Our inference method can handle not only simple loops but also mutually recursive programs with complex recursion patterns. Moreover, with case analysis, phase-change programs are naturally recognized by our approach, even if they were expressed in mutual-recursive form, as shown in Fig. 4-11,

```
void f(int x, int y) {
  if (x < 0) return;   else g(x + y, y + 1); }

void g(int x, int y) {
  if (x < 0) return;   else f(x, y − 2); }
```

Figure 4-11. A phase-change recursive program

which is inspired from the phase-change loop:

$\texttt{while}\ (\mathtt{x}{\geq}0)\ \{\mathtt{x} = \mathtt{x} + \mathtt{y};\ \mathtt{y} = \mathtt{y} - 1;\}$

from [43], in which the value of y is decreasing toward 0 before the value of x starts to decrease. However, unlike this loop, the value of y in our example is not continuously decreasing so that the supporting condition $\mathtt{y}{<}0$ cannot be observed easily. Moreover, since the approach proposed in [43] is designed only for the while loops with transition relation, it cannot handle this recursive program. The size-change approach [7, 102] can show that y is decreasing across the recursive chain but it is unable to prove program termination because a necessary supporting condition $\mathtt{y}{<}0$ cannot be derived by such size-change analysis.

The inferred termination specification of the method f and g by our approach is:

```
void f(int x, int y)
case {
  x < 0 → requires Term ensures true;
  x ≥ 0 → case {
    y < 0 → requires Term [1, 2x − 1] ensures true;
    y ≥ 0 → requires Term [2, 2y + 2] ensures true; }}
```

$$
\begin{array}{l}
\texttt{void}\ \texttt{g}(\texttt{int}\ \texttt{x}, \texttt{int}\ \texttt{y}) \\
\texttt{case}\ \{ \\
\quad \texttt{x} < 0 \rightarrow \texttt{requires}\ \texttt{Term}\ \texttt{ensures}\ \texttt{true}; \\
\quad \texttt{x} \geq 0 \rightarrow \texttt{case}\ \{ \\
\quad\quad \texttt{y} \leq 0 \rightarrow \texttt{requires}\ \texttt{Term}\ [1, 2\texttt{x}]\ \texttt{ensures}\ \texttt{true}; \\
\quad\quad \texttt{y} > 0 \rightarrow \texttt{requires}\ \texttt{Term}\ [2, 2\texttt{y}{-}1]\ \texttt{ensures}\ \texttt{true}; \}\}
\end{array}
$$

Here, the termination behaviors of these two methods are also partitioned into distinct phases, numbered by the prefix 1 and 2, with suitable ranking function for each phase.

4.7 Discussion

Over the last decade, there has been a large body of work on proving program termination. Most of these termination provers, such as TERMINATOR [46] and its successor T2 [29, 48], ARMC [129], TAN [92] and ULTIMATE [74], either show that a program terminates for all (given) inputs or return a counterexample to termination upon the failure of termination proofs. However, due to the incompleteness of termination-based techniques, these provers cannot guarantee that every returned counterexample (from failed termination proofs) leads to a definitely non-terminating execution. Thus, each tool might deploy a separate non-termination proving technique to prove that the counterexample is feasible. Also, each such counterexample is only an under-approximation of its program execution, so that it does not capture the wider scenarios for non-terminating behaviors of the analyzed program.

We have also seen much related work on proving program non-termination, e.g. [11, 30, 35, 70, 94, 123, 146]. Non-termination provers, such as TNT [70] and INVEL [146], attempt to disprove program termination by searching for

some initial configurations that act as witnesses for non-termination. To find a wider class of non-termination bugs, these approaches attempt to discover sufficient pre-conditions for non-termination. Nevertheless, since non-termination proving techniques are also incomplete, the analyzed program is not guaranteed to terminate under the complement of the inferred pre-condition for non-termination.

The dual problem of conditional termination, first addressed in [43], identifies initial configurations that ensure termination. In [43], such termination preconditions are derived from potential ranking functions, which are bounded but not decreasing. Later, the tools FLATA [26] and ACABAR [61] infer the sufficient precondition for termination from (the negation on over-approximation of) the set of initial states from which the program might not terminate. However, FLATA differs from ACABAR by limiting itself to classes of loops with restricted forms in which the precise non-termination conditions are definable.

Chapter 5

A Proof Slicing Framework for Program Verification

In the context of program verification, we propose a *formal framework* for *proof slicing* that can aggressively reduce the size of proof obligations as a means of performance improvement. In particular, each large proof obligation may be broken down into smaller proofs, for which the overall processing cost can be greatly reduced, and be even more effective under *proof caching*, since multiple instances of smaller sub-proofs of a large original proofs usually occur frequently. Our proposal is built on top of existing automatic provers, including the state-of-the-art prover Z3, and can also be viewed as a re-engineering effort in proof decomposition that attempts to avoid large-sized proofs for which these provers may be particularly inefficient. In our approach, we first develop a calculus that formalizes a *complete proof slicing* procedure, which is followed by the development of an *aggressive proof slicing* method. Retaining completeness is important, and thus in our experiments the complete method serves as a backup for the cases when the aggressive procedure fails. The foundations of the aggressive slicing procedure are based on a novel

lightweight annotation scheme that captures *weak links* between sub-formulas of a proof obligation; the annotations can be inferred automatically in practice, and thus both methods are fully automated.

5.1 Overview

A significant challenge in the area of program verification is posed by the ever increasing number and complexity of proof obligations that need to be discharged by automated theorem provers. To overcome this challenge, a number of previous investigations have considered the approach of "shrinking" the generated proof obligations as a means of speeding up the solvers. [107] splits the proof obligations based on control flow to get smaller proofs. [113, 138, 139] detect and discard information that is not relevant to the problem at hand, thus streamlining the proof process. When this streamlining is performed aggressively, the size of the resulting proof obligations may be greatly reduced, leading to opportunities for significant performance improvement. In this context, an important technique is that of *proof caching* [83], which reuses proof results when multiple instances of the same sub-formulas are encountered. While the idea of *proof slicing* is not new in the context of automatic theorem provers, we believe that the procedure is more effectively carried out in the larger scope of program verification. In this regards, we make new contributions in three key directions, namely *(i)* the development of a *formal foundation* for proof slicing mechanisms, *(ii)* a general application of proof slicing that is *prover-independent* and tailored to *program verification*, and *(iii)* an *annotation scheme* that allows a more aggressive application of the mechanism, leading to improved performance.

A formal foundation in proof slicing is important for providing an avenue

towards a more rigorous investigation into the field. To that end, we first develop a *complete* calculus for automatic slicing, which serves as a foundation for the implementation of our tool. Importantly, apart from completeness, this calculus also enjoys properties of convergence and completeness, which are crucial for its trustworthiness, and its potential for efficient implementation.

One important application area is that of program verification, whereby a typical approach is to employ a program verifier that processes the code of interest, annotated with pre/post-conditions, in order to produce a set of proof obligations that are subsequently passed on to off-the-shelf theorem prover. These proof obligations are fundamentally of the form $P \Rightarrow Q$, whereby each P is an antecedent that captures some current program state, while Q is a goal (or assertion) that has to be proven. Since proof slicing remains complete only when the antecedent is satisfiable, and since satisfiability checks typically add a non-negligible overhead, existing state-of-the-art theorem provers, with formula reduction techniques such as relevancy propagation [51], or labelled splitting [59], do not employ this mechanism. However, with our slicing mechanism placed in-between the verifier and the theorem prover, we ensure that the satisfiability checks of antecedents are *incremental* and with low overhead, which is key to good performance.

As a further improvement, we designed an *annotation scheme* that captures *constraint linking properties*, that is, variable-sharing dependencies between interpreted atoms (*i.e.*, constraints) of a proof obligation; this scheme enables an *aggressive slicing* procedure. We believe that such an approach allows proof slicing to be viewed as a modular and extensible mechanism, rather than as a black box with limited functionality. This point is particularly poignant, as a good annotation scheme is also the basis for effective *annotation inference*

mechanisms. These mechanisms can, in general, be completely automatic; several examples can be found in the experimental results section.

5.2 Proof Slicing for Program Verification

Depending on the context, we shall use the term "slicing" to denote either formula slicing or proof slicing. Formula slicing is the partitioning of a formula into "slices" – sub-formulas that group together related constraints. Two slices are said to be *disjoint* if they do not share any common variables, otherwise they are said to be *overlapping*. Proof slicing is the partitioning of a proof obligation into smaller sub-proofs to reduce the proof's complexity, thus improving performance of discharging proofs.

In the context of program verification, there are typically two major kinds of proof obligations, namely: *(i) Entailment checking*, of the form $P \vdash Q$ and *(ii) Unsatisfiability checking*, of the form $\mathit{UNSAT}(P)$ or $P \vdash \texttt{false}$. For unsatisfiability checking, the proof slicing mechanism partitions the initial formula P into a set of disjoint slices $\{P_1, \ldots, P_n\}$ whereby $P \leftrightarrow P_1 \wedge \cdots \wedge P_n$, and then incrementally applies unsatisfiability checks on some of these slices, *i.e.*, the slices that have been recently modified since the last unsatisfiability checks.

For entailment checking, proof slicing is the division of an initial, large entailment formula into smaller ones, obtained by slicing the original formula's antecedent with respect to each of its consequent. Given an antecedent P and a conjunctive consequent $Q_1 \wedge \cdots \wedge Q_n$, we partition P into possibly overlapping slices $\{P_1, \ldots, P_n\}$ such that each slice P_i is sufficient to prove the corresponding consequent Q_i. That is, the original entailment is replaced by a set of smaller entailments $\{P_i \vdash Q_i\}_{i=1}^{n}$. Importantly, this slicing step assumes

that the sequent's antecedent is satisfiable, *i.e.*, it has been subjected to a prior unsatisfiability check. Loss of completeness occurs when weakening an unsatisfiable antecedent into a satisfiable one, and is the main reason for the limited adoption of this optimization in mainstream theorem provers.

Let consider the implication checks of the form $P_1 \wedge \cdots \wedge P_n \Rightarrow Q_1 \wedge \cdots \wedge Q_m$. Without proof slicing, a theorem prover needs to prove the unsatisfiability of $P_1 \wedge \cdots \wedge P_n \wedge (\neg Q_1 \vee \cdots \vee \neg Q_m)$. Due to the possibility of $P_1 \wedge \cdots \wedge P_n$ being unsatisfiable, the prover could not drop any constraint of the antecedents, unless it is willing to risk a loss of precision. By explicitly distinguishing between two kinds of proof obligations, our framework can avoid this problem by a prior unsatisfiable checking of the antecedents. Moreover, this distinction also allows us to exploit more aggressive pruning of irrelevant constraints from the antecedents with a novel annotation scheme (see Sec. 5.5).

Let us demonstrate how proof slicing can be applied to help with verifying the code snippet in Fig. 5-1(a). The pre- and post-conditions are provided by the `assume` and `assert` statements, respectively. To prove the total correctness of this program, we use the loop invariant $\mathtt{x{=}2y} \wedge \mathtt{n{\geq}0}$ for partial correctness proof, and the variant `n` as a well-founded measure for termination proof. The set of generated verification conditions are shown in Fig. 5-1(b). Observe that in these verification conditions, the constraints of `x` and `y` and the constraints of `n` are disjoint. As a result, they can be proven independently by the proof slicing mechanism, resulting in simpler proof obligations. For example, the verification condition $\mathtt{VC_4}$ can be split into two separate entailments

$$\mathtt{VC_{4a}} : \mathtt{x{=}2y} \vdash \mathtt{x{+}2{=}2(y{+}1)} \quad \mathtt{VC_{4b}} : \mathtt{n{\geq}0} \wedge \mathtt{n{>}0} \wedge \mathtt{n{=}N_0} \vdash \mathtt{n{-}1{\geq}0} \wedge \mathtt{n{-}1{<}N_0}$$

by partitioning the antecedent into two slices *(i)* $\mathtt{x{=}2y}$ and

1:	$\texttt{assume(n} \geq \texttt{0);}$		$\texttt{Inv(x,y,n)} \equiv \texttt{x=2y} \land \texttt{n}\geq\texttt{0}$
2:	$\texttt{x = 0; y = 0;}$		
3:	$\texttt{while (n > 0) \{}$	$\texttt{VC}_1:$	$\texttt{x=0} \land \texttt{y=0} \land \texttt{n}\geq\texttt{0} \vdash \texttt{Inv(0,0,n)}$
4:	$\quad \texttt{x = x + 2;}$	$\texttt{VC}_2:$	$\texttt{Inv(x,y,n)} \land \neg\texttt{(n>0)} \vdash \texttt{x=2y} \land \texttt{n=0}$
5:	$\quad \texttt{y = y + 1;}$	$\texttt{VC}_3:$	$\texttt{Inv(x,y,n)} \land \texttt{n>0} \vdash \texttt{n}\geq\texttt{0}$
6:	$\quad \texttt{n = n - 1; \}}$	$\texttt{VC}_4:$	$\texttt{Inv(x,y,n)} \land \texttt{n>0} \land \texttt{n=N}_0$
7:	$\texttt{assert(x = 2*y} \land \texttt{n=0);}$		$\vdash \texttt{Inv(x+2,y+1,n-1)} \land \texttt{n-1<N}_0$
	(a)		(b)

Figure 5-1. A code snippet and its verification conditions for total correctness proof

(ii) $\texttt{n}\geq\texttt{0} \land \texttt{n>0} \land \texttt{n=N}_0$. Prior to the entailment checks, each new antecedent is subjected to a satisfiability check, if its slice has changed when compared to an earlier program point. We note that only formula slice *(ii)* has changed, with its invariant strengthened by the extra constraints $\texttt{n>0} \land \texttt{n=N}_0$. Thus, for $\texttt{VC}_4$, we only need to check the satisfiability of the slice *(ii)*, instead of the whole antecedent.

In summary, the division of proof obligations into two classes, of entailments and unsatisfiability checks, both of which benefit in performance from proof slicing, distinguishes our work from the techniques employed in current theorem provers. In entailment checks, the size of the antecedent can be greatly reduced when subjected to a prior unsatisfiability check. A similar mechanism is used for unsatisfiability checks, where only changed slices need be re-checked. Without this early analysis on the potential satisfiability of antecedents, current theorem provers would have to process much larger sets of constraints[1] when discharging proof obligations produced by a verification system.

[1] A theorem prover might group relevant constraints into classes, such as congruence classes in the theory of equality, or classes of different theories in the Nelson-Oppen theory combination, or more generally, classes of constraints which share some common symbols.

5.3 A Framework for Proof Slicing

The starting point of our formalization is that of entailment or unsatisfiability obligations whose left hand side is an unquantified conjunction of constraints and uninterpreted predicates. For reasons of simplicity, we shall confine our presentation to unquantified formulas; the system is, nevertheless, capable of handling quantifiers. Informally, the slicing mechanism will preprocess the input by always floating outwards the constraints that appear under quantifiers but are independent of the corresponding quantified variables, and treat the remaining quantified constraints as atomic.

$$(\wedge N)\frac{X_{i_0}{=}X'_{j_0}}{\bigwedge_i X_i \vee \bigwedge_j X'_j \hookrightarrow X_{i_0}\wedge(\bigwedge_{i\neq i_0} X_i \vee \bigwedge_{j\neq j_0} X'_j)}$$

$$(\wedge R)\ \frac{P \vdash Q_1 \quad P \vdash Q_2}{P \vdash Q_1 \wedge Q_2} \qquad (\vee L)\ \frac{P_1 \vdash Q \quad P_2 \vdash Q}{P_1 \vee P_2 \vdash Q}$$

Consequently, we consider a first-order language with equality and interpreted function symbols. The atoms of the language are formed in the usual way, and denote *constraints*, *i.e.*, predicates that have a fixed interpretation with respect to an external automated reasoning tool. Sequents are denoted by $P \vdash Q$, where P and Q are formulas. Our slicing mechanism is specified by the rules in Fig. 5-2, and works by taking in a sequent, and outputting a set of sliced sequents that are meant to be discharged by off-the-shelf provers. However, the input sequent must first undergo a pre-processing stage with the beside rewrite rule $(\wedge N)$ and two structural rules $(\wedge R)$ and $(\vee L)$, which yields a set of sequents in a form where the effect of the slicing rules in Fig. 5-2 is maximized, while retaining completeness. The result

$$\textbf{[SPLIT-E1]}\quad \frac{}{\textsc{Split}(\{\}) = \{\}}$$

$$\textbf{[SPLIT-E2]}\quad \frac{\textsc{Split}(P) = R \qquad P_1 = \{Q \in R \mid \; \exists\beta \in Q.\textsc{SameSlice}(\alpha,\beta)\} \qquad P_2 = \{Q \in R \mid \neg\exists\beta \in Q.\textsc{SameSlice}(\alpha,\beta)\}}{\textsc{Split}(\{\alpha\} \cup P) = P_2 \cup \{\{\alpha\} \cup \bigcup_{X \in P_1} X\}}$$

$$\textbf{[GETCTR-E1]}\quad \frac{}{\textsc{GetCtr}_0(Q, PS) = \{\}}$$

$$\textbf{[GETCTR-E2]}\quad \frac{\{S \in PS \mid \textsc{IsRelevant}(Q,S)\} = \{\}}{\textsc{GetCtr}_n(Q, PS) = \{\}}$$

$$\textbf{[GETCTR-E3]}\quad \frac{S_1 = \{S \in PS \mid \; \textsc{IsRelevant}(Q,S)\} \qquad R = \bigcup_{X \in S_1} X \qquad R' = \textsc{GetCtr}_{n-1}(R, PS \setminus S_1)}{\textsc{GetCtr}_n(Q, PS) = R \cup R'}$$

$$\textbf{[P-ENTAIL]}\quad \frac{\textsc{Split}(\{P_i\}_{i=0}^{m}) = PS \qquad \textsc{GetCtr}_n(Q, PS) \Rightarrow Q}{\bigwedge_{i=0}^{m} P_i \vdash Q}$$

$$\textbf{[P-UNSAT]}\quad \frac{\textsc{Split}(\{P_i\}_{i=0}^{m}) = PS \qquad \exists X \in PS \cdot \textsc{GetCtr}_n(X, PS) \Rightarrow \texttt{false}}{\mathit{UNSAT}(\bigwedge_{i=0}^{m} P_i)}$$

Figure 5-2. Framework for Proof Slicing Mechanisms

of this decomposition is a set of sequents whose LHS is a conjunctive formula and RHS is either a disjunctive or atomic formula. However, to avoid increasing the number of sub-sequents when these rules are applied, that may lead to some performance loss, rule $(\wedge N)$ should take precedence over rules $(\wedge R)$ and $(\vee L)$, if applicable, and rule $(\wedge R)$ can be stopped early if the pair of conjunctive consequents in the RHS share the same set of variables.

We distinguish between two calculi: a *complete slicing* calculus, and an *aggressive slicing* calculus. Both calculi formalize mechanisms for partitioning the conjuncts of a sequent, yielding sets of smaller sequents whose discharge is sufficient for establishing the proof of the original sequent. The assumption here is that the total effort of proving the set of smaller sequents by means of external provers is, in general, lighter than the effort of proving the original sequent by the same means. In the optimal case, the application of slicing decomposes the entailment $P_1 \wedge \ldots \wedge P_n \models Q$ into several sub-formulas, of the form $\bigwedge_{P \in X_i} P \models Q$, such that the sets X_i satisfy three properties: (i)

inclusion: $\forall i. X_i \subseteq \{P_1, \ldots, P_n\}$, (ii) *relevance*: all X_i constraints are relevant to Q, *i.e.*, $\forall R. R \in X_i \rightarrow \bigwedge_{P \in X_i \setminus \{R\}} P \nvDash Q$ and (iii) *correlation*: for each pair of constraints $P, P' \in X_i$, there exists a chain $P = P_1, \ldots, P_k = P'$ such that every two consecutive constraints P_j, P_{j+1} are overlapping. Similarly, an unsatisfiability check for a formula $P_1 \wedge \ldots \wedge P_n$ is sliced into several unsatisfiability checks for $\bigwedge_{P \in X_i} P$ such that X_i satisfies the inclusion and correlation properties.

Unfortunately, this formulation is not practical, as even establishing the relevance for a given slice is costly, let alone discovering the slices. Our proposal relies on a more syntactic formulation for the relevance and correlation properties, by using two meta-predicates, ISRELEVANT and SAMESLICE, as approximations of the relevance and correlation tests. The actual definitions dictate the slicing strategies each calculus uses. In the following sections, we expand more on their formulation and usage.

The complete and aggressive slicing calculi share the set of rules given in Fig. 5-2, which we shall call the *slicing framework* and differ in the definitions used for the two meta-predicates. Specifically, to obtain the *complete* (or *aggressive*) slicing calculus, we add the rules in Fig. 5-3 (or in Fig. 5-7, resp.) to the framework. We shall discuss the framework in the remainder of this section, and we shall devote Sec. 5.4 and 5.6 to each of the two calculi.

The conjunct partitioning procedure SPLIT calculates PS, a set of slices, from a set of conjuncts. Each slice is either extended with a new conjunct or not, in accordance with the SAMESLICE meta-predicate. This meta-predicate's role is to establish if two conjuncts should be kept in the same slice or not. Intuitively, it works by checking how information is shared between its two arguments. The result of applying the SPLIT relation to a formula P is a set of

sets of constraints that represent the partitioning into *slices* of P. Each set of constraints can be interpreted as a formula that is formed by a conjunction of its constraints. Propertywise, we have:

$$\bigcup \textsc{Split}(P){=}P \wedge (\forall X, Y{\in}\textsc{Split}(P){\cdot}X{\neq}Y \rightarrow X{\cap}Y{=}\{\})$$

The formulation of [SPLIT−E2] allows for arbitrary slicing decisions from the picking of α. Nevertheless, the slicing mechanism needs to be *convergent*, that is, to yield the same set of sliced sequents upon termination. Slicing convergence can be ensured by requiring the rewrite system formed by [SPLIT] to be confluent. In the following sections, we shall investigate convergence properties for the complete and aggressive slicing calculi.

Another operation of interest is the computation of relevant slices for a given formula from a set of slices. [GETCTR−E3] and [GETCTR−E2] describe a family $\textsc{GetCtr}_n$ of such functions that differ only in the exhaustiveness of the relevance computation. All start by picking the slices that are in the IsRelevant relation with the input formula Q. This step can be repeated using each of the previously selected slices as input for the next iteration. Such a refinement is important because, depending on the actual definition used for SameSlice, a single step might not be sufficient to gather all relevant constraints[2]. The default GetCtr function to use is $\textsc{GetCtr}_1$, but we can gradually increase its coverage through $\textsc{GetCtr}_2$, $\textsc{GetCtr}_3$, ..., if needed. This family of operators satisfies the following two properties

$$\textit{(i)}\ \textsc{GetCtr}_n(Q, PS) \subseteq PS \qquad \textit{(ii)}\ \textsc{GetCtr}_n(Q, PS) \subseteq \textsc{GetCtr}_{n+1}(Q, PS)$$

[2]Such is the case for the *aggressive slicing calculus* with an *annotation scheme* that will be introduced later.

Continuing on with the description of the slicing rules in Fig. 5-2, the rule [P−UNSAT] defines slicing for unsatisfiability obligations. The formula P is first partitioned, and then a search is performed for an unsatisfiable slice. Each slice is considered together with its relevant counterparts as computed by GETCTR_n. The $\Rightarrow$ notation signifies the invocation of an external prover.

Similarly, [P−ENTAIL] defines the treatment of entailment obligations. The rule prescribes partitioning of the antecedent and the consequent, pairing consequent slices with relevant antecedent slices, and enforcing the implication relation on the resulting pairs. The [P−ENTAIL] rule corresponds to the conjunction introduction rules of Gentzen's sequent calculus [33]. Intuitively, a sequent with conjunctions on the right hand side can be split into separate sequents, each retaining one conjunct. Similarly, sequents with conjunctions on the left hand side can have any number (desirably, all but one) of conjuncts discarded. We state the lemma for soundness as follows, where its proof can be found in Appendix B.

Lemma 4 (Soundness) *All sequents proven using the rules of the slicing framework are true.*

5.4 Complete Proof Slicing

In this section we introduce a completely automatic slicing mechanism. This mechanism uses the slicing framework rules given in Fig. 5-2, together with the meta-predicates SAMESLICE and ISRELEVANT given in Fig. 5-3. Essentially, this mechanism produces slices whose sets of free variables are disjoint. This is based on the idea that if a hypothesis and the conclusion of a proof obligation have disjoint sets of free variables, then the hypothesis cannot be directly contributing to the proof of the conclusion, and can thus be discarded.

$$[\underline{\textsc{as-correlation}}]$$
$$\textsc{SameSlice}(P_1, P_2) = \mathcal{V}(P_1) \cap \mathcal{V}(P_2) \neq \{\}$$

$$[\underline{\textsc{as-relevance}}]$$
$$\textsc{IsRelevant}(Q, P) = \mathcal{V}(Q) \cap \mathcal{V}(P) \neq \{\}$$

Figure 5-3. Complete Slicing Mechanism

Whenever two conjuncts of the hypothesis share free variables, we say that they are *correlated*, and under the current slicing scheme, they should belong to the same slice. This is reflected in the rule [AS−CORRELATION], where the meta-predicate SAMESLICE is defined to keep two conjuncts together if their sets of free variables are correlated. Here, the symbol $\mathcal{V}$ denotes a function that returns the set of free variables from its input.

Similarly, if a conjunct in the hypothesis shares variables with the consequent, we say that the conjunct is *relevant* to proving the conclusion. The definition of the meta-predicate ISRELEVANT given in the rule [AS−RELEVANCE] captures precisely this idea. We have taken the approach of utilizing these two rules to make our proof slicing framework more general. In the next section, we shall define a new variant of our proof slicing framework with annotation guidance, by simply redefining these two rules, without having to change any of the rules in Fig. 5-2.

In the previous section, we mentioned that [SPLIT] rules are expected to be convergent. This can be ensured by the convergence of our calculi. The following lemma substantiates this claim.

Lemma 5 [SPLIT] *with* [AS−CORRELATION] *is confluent.*

An important property of the complete slicing mechanism is that it does not alter the level of completeness of the underlying solver. The slicing mechanism

converts provable sequents into new sequents that are still provable in the same logic, provided that the antecedent of the sequent at hand is satisfiable. To formalize this claim, we assume that the underlying prover is formalized as a calculus LK^T, obtained from Gentzen's calculus LK [33], augmented with a theory T capable of handling the interpreted symbols of the language. Moreover, we assume that the axioms of T do not discharge sequents of the form $P \vdash Q$ when $\mathcal{V}(P) \cap \mathcal{V}(Q) = \{\}$.

Lemma 6 (Relative completeness) *Let $P' \vdash Q$ be the sequent obtained by applying the complete slicing rules to the sequent $P \vdash Q$, where Q is atomic. Let LK^T be a sequent calculus obtained from LK by augmenting it with rules from a theory T that can handle the interpreted symbols of our formulas. If $P \vdash Q$ is provable, and P is satisfiable in LK^T, then $P' \vdash Q$, is also provable in LK^T.*

5.5 An Annotation Scheme for Proof Slicing

The complete proof slicing mechanism is particularly effective in the case of formulas that can be neatly partitioned into disjoint slices. It is, however, not as effective in the presence of constraints that seemingly link together sub-formulas that would otherwise be disjoint; for such cases, slicing needs to be applied more aggressively. To highlight this need, let us now consider a more expressive logic, capable of specifying and verifying heap-manipulating programs, with the possibility of generating more complex proof obligations. Consider the definitions of a binary tree node and an inductive predicate that specifies an AVL tree rooted at its first argument and height-balanced in Fig. 5-4.

```
data node { int val; node left; node right; }
pred avl(root, n, h, B) ≡ root=null∧n=0∧h=0∧B={}
 ∨ ∃v, p, q, n1, n2, h1, h2 · root↦node(v, p, q)
  ∗ avl(p, n1, h1, B1) ∗ avl(q, n2, h2, B2)
  ∧ n=1+n1+n2 ∧ h=1+max(h1, h2)∧−1≤h1−h2≤1
  ∧ B={v}∪B1∪B2∧(∀a∈B1·a<v)∧(∀b∈B2·v≤b)
 inv n≥0 ∧ h≥0 ∧ n≥h;
```

Figure 5-4. The avl heap predicate

This predicate captures four aspects of the AVL tree property. Parameter root is a pointer to the root of the tree, whereas n, h, and B (and their subscripted variants) capture, respectively, numbers of nodes in trees, their heights, and their sets of values. The constraint $-1 \leq h_1 - h_2 \leq 1$ states that the tree is nearly height-balanced, whereas the quantified set constraint $(\forall a \in B_1 \cdot a < v) \wedge (\forall b \in B_2 \cdot v \leq b)$ enforces the binary search tree property. Recall that the formula specified after the **inv** keyword denotes the invariant property that holds for all instances of the predicate. Moreover, the *separating conjunction* operator $*$ (cf. [118]) is used to concisely capture the memory disjointness property.

To prove an invariant of the AVL predicate (*e.g.*, $n \geq 0$), the entailment proof (*e.g.*, $\mathtt{avl}(x, n, h, B) \vdash n \geq 0$, resp.) can be discharged inductively by applying the definition of the predicate avl. For example, the below LHS is the resulting proof obligations (after each points-to $\mapsto$ is approximated by a non-null constraint, and each predicate is approximated by its invariant) while RHS is the same two entailments after applying *complete* proof slicing. For brevity, we use $n_i, h_i \geq 0$ to denote the conjunction $n_i \geq 0 \wedge h_i \geq 0$.

Though sound, the second (sliced) entailment is unnecessarily verbose due to the presence of constraints $n_1 \geq h_1$ and $n_2 \geq h_2$ which act to link the constraints relating to size and height for the avl predicate. We refer to such

$$\texttt{x=null} \wedge \texttt{n=0} \wedge \texttt{h=0} \wedge \texttt{B=\{\}} \vdash \texttt{n}\geq 0 \qquad\qquad \texttt{n=0} \vdash \texttt{n}\geq 0$$

$$\begin{array}{ll}
\texttt{x}\neq\texttt{null} & \\
\wedge(\texttt{n}_1,\texttt{h}_1\geq 0\wedge\texttt{n}_1\geq\texttt{h}_1)\wedge(\texttt{n}_2,\texttt{h}_2\geq 0\wedge\texttt{n}_2\geq\texttt{h}_2) & (\texttt{n}_1,\texttt{h}_1\geq 0\wedge\texttt{n}_1\geq\texttt{h}_1)\wedge(\texttt{n}_2,\texttt{h}_2\geq 0\wedge\texttt{n}_2\geq\texttt{h}_2) \\
\wedge\ \texttt{n=1+n}_1\texttt{+n}_2 & \wedge\ \texttt{n=1+n}_1\texttt{+n}_2 \\
\wedge\ \texttt{h=1+max(h}_1,\texttt{h}_2)\wedge -1\leq\texttt{h}_1-\texttt{h}_2\leq 1 & \wedge\ \texttt{h=1+max(h}_1,\texttt{h}_2)\wedge -1\leq\texttt{h}_1-\texttt{h}_2\leq 1 \\
\wedge\ \texttt{B=\{v\}}\cup\texttt{B}_1\cup\texttt{B}_2 & \\
\wedge\ (\forall \texttt{a}\in\texttt{B}_1\cdot\texttt{a<v})\wedge(\forall\texttt{b}\in\texttt{B}_2\cdot\texttt{v}\leq\texttt{b}) & \\
\vdash \texttt{n}\geq 0 & \vdash \texttt{n}\geq 0
\end{array}$$

constraints as *weakly linking* constraints, and propose to deploy a more aggressive proof slicing mechanism that can selectively disregard the relationship between variables occurring in such linkages. Though this decision may suffer from a risk of losing completeness, it would allow for a more aggressive application of the slicing mechanism. Applying this mechanism, we are able to obtain the following more compact entailment proof (*e.g.*, $\texttt{n}_1\geq 0 \wedge \texttt{n}_2\geq 0\wedge \texttt{n=1+n}_1\texttt{+n}_2 \vdash \texttt{n}\geq 0$). To provide a systematic way to deal with weakly linking constraints, we propose the following annotation scheme.

Informal Definition 1 (Weakly Linking Constraint) *A constraint ϕ can be annotated as a* weakly linking *constraint $\phi\#$ if it is a weak constraint, such as inequality constraint (e.g., $\leq$ or $\neq$), that links together multiple variables from disjoint properties.*

In addition, for proving the invariant $\texttt{n}\geq\texttt{h}$ of the AVL predicate, our annotated proof slicing mechanism would keep the constraints related to both the size and the height properties and their weakly linking constraints, as follows:

$$\begin{array}{l}
\texttt{n}_1,\texttt{n}_2\geq 0 \wedge \texttt{h}_1,\texttt{h}_2\geq 0 \wedge (\texttt{n}_1\geq\texttt{h}_1)\# \wedge (\texttt{n}_2\geq\texttt{h}_2)\# \\
\wedge\ \texttt{n=1+n}_1\texttt{+n}_2 \wedge \texttt{h=1+max(h}_1,\texttt{h}_2) \wedge -1\leq\texttt{h}_1-\texttt{h}_2\leq 1 \vdash \texttt{n}\geq\texttt{h}
\end{array}$$

Aside from weakly linking constraints, we propose to support two additional

kinds of weak linkages, namely:

Informal Definition 2 (Weakly Linking Variable) *A variable occurrence v can be annotated as a* weakly linking *variable $v\#$ if it does not belong to any particular property, but appears in the constraints of multiple distinct properties.*

Informal Definition 3 (Weakly Linking Expression) *An expression e can be annotated as a* weakly linking *expression $e\#$ if its definition has been captured by another variable, in a constraint such as $v{=}e$. This variable (or property) is only weakly linked with variables inside the linking expression.*

We note here that each weakly linking annotation is added only once (mostly in predicate definitions and specifications), with the intent of being used across the entire program verification process.

In summary, the key points on the use of weakly linking annotations in support of more aggressive proof slicing are: *(i)* Proof obligations containing multiple weakly linked properties are commonly generated from richer specifications. *(ii)* The use of weakly linking annotations leads to loosely connected partitions that can be split when necessary, thus easily regaining the performance benefits of proof slicing. *(iii)* Multiple instances of the same (small) slice are frequently encountered in practice, which are shown in our experiments; thus, the use of proof caching would yield further performance gains.

Moreover, in a goal driven approach, it is possible to select only a small set of (loosely connected) partitions that have a higher chance of being relevant for the current proof obligation. Should this attempt fail, the algorithm can retry with a broader set of partitions, preserving the precision of the approach. Since failure rate is small in practice, this aggressive approach yields a significant

$$\begin{array}{ll} \pi & ::= \alpha_{\mathcal{L}} \mid \neg\alpha_{\mathcal{L}} \mid \pi_1 \wedge \pi_2 \\ \alpha_{\mathcal{L}} & ::= \alpha \mid (\alpha)\# \qquad v_{\mathcal{L}} \ ::= \ v \mid v\# \\ \alpha & ::= \texttt{true} \mid f_{\mathcal{L}}(v^*_{\mathcal{L}}) \mid v_{\mathcal{L}}{=}f_{\mathcal{L}}(v^*_{\mathcal{L}}) \mid v_{\mathcal{L}1}{=}v_{\mathcal{L}2} \\ f_{\mathcal{L}}(v^*_{\mathcal{L}}) & ::= f(v^*_{\mathcal{L}}) \mid (f(v^*_{\mathcal{L}}))\# \\ \textit{where} & \# \textit{ is the annotated slicing label}; \\ & \alpha \textit{ denotes atomic predicates}; \\ & \pi \textit{ denotes pure formulas};\ v \textit{ is a variable}; \\ & v_{\mathcal{L}} \textit{ is a variable with or without \# label}; \\ & f_{\mathcal{L}} \textit{ is an interpreted symbol, possibly labeled}; \end{array}$$

Figure 5-5. Support Logic with Annotation Scheme

improvement in efficiency. In our experiments, we have obtained multi-fold reductions in prover execution times.

5.6 Aggressive Proof Slicing

In this section, we propose a novel *annotation* mechanism, capable of pinpointing locations where proof slicing can be applied more aggressively.

5.6.1 Annotation Scheme

As mentioned in Sec. 5.3, the target of our framework is a first-order language with equality and interpreted function symbols. This language, more precisely described in Fig. 5-5, imposes no restrictions on the versatility of our framework. Without loss of generality we can safely assume that the annotations described in Sec. 5.5 will be transparently translated into annotations in our target language.

5.6.2 Annotation Reduction

To simplify the formulation of our core calculus, we shall restrict our annotations for proof slicing to only weakly linking variables. Through a preprocessing step, we can transform each weakly linking constraint and each

$$
\begin{array}{lcl}
red_\beta(\pi_1 \wedge \pi_2) & \Rightarrow & red_\beta(\pi_1) \wedge red_\beta(\pi_2) \\
red_\beta(\neg \alpha_\mathcal{L}) & \Rightarrow & \neg red_\beta(\alpha_\mathcal{L}) \\
red_\beta((\alpha)\#) & \Rightarrow & red_{\texttt{true}}(\alpha) \\
red_\beta(\texttt{true}) & \Rightarrow & \texttt{true} \\
red_\beta(f(v^*_\mathcal{L})) & \Rightarrow & f(red_\beta(v^*_\mathcal{L})) \\
red_\beta((f(v^*_\mathcal{L}))\#) & \Rightarrow & f(red_{\texttt{true}}(v^*_\mathcal{L})) \\
red_\beta(f_\mathcal{L}(v^*_\mathcal{L})) & \Rightarrow & f_\mathcal{L}(red_\beta(v_\mathcal{L})^*) \\
red_\beta(v_\mathcal{L}{=}f_\mathcal{L}(v^*_\mathcal{L})) & \Rightarrow & red_\beta(v_\mathcal{L}){=}f_\mathcal{L}(red_\beta(v_\mathcal{L})^*) \\
red_\beta(v_{\mathcal{L}1}{=}v_{\mathcal{L}2}) & \Rightarrow & red_\beta(v_{\mathcal{L}1}){=}red_\beta(v_{\mathcal{L}2}) \\
red_\beta(v\#) & \Rightarrow & v\# \\
red_{\texttt{true}}(v) & \Rightarrow & v\# \\
red_{\texttt{false}}(v) & \Rightarrow & v
\end{array}
$$

Figure 5-6. Rules for Annotation Reduction

weakly linking expression into weakly linking variables, by transferring the weakly linking annotation to the free variables of a linking constraint or linking expression. Such a translation, named red, is formalized in Fig. 5-6.

With this translation scheme, the free variable set of each constraint is divided into two disjoint sets, namely *weakly* and *strongly linking* variables. The set of *weakly linking* variables of a constraint can be computed by a simple function $\mathcal{V}_\mathcal{W}$ over the structure of the constraint α that picks up all (weakly) annotated variables, $\mathcal{V}_\mathcal{W}(v\#) = \{v\}$ while the set of *strongly linking* variables of a constraint α is its complement, namely $\mathcal{V}_\mathcal{S}(\alpha) = \mathcal{V}(\alpha) \setminus \mathcal{V}_\mathcal{W}(\alpha)$, where $\mathcal{V}(\alpha)$ returns the free variable set (without annotation) of the constraint α.

The translation scheme described above converts away all non-variable annotations. Nevertheless, a weakly linking constraint can still be distinguished from a constraint with weakly linking expressions or a constraint with a mix of weakly and strongly linking variables. At this point, we can make the following general observations: *(i)* a strongly linking constraint expresses knowledge specific to one property, and does not have any weakly linking variables; *(ii)* a weakly linking constraint encodes only weakly linking

$$\boxed{\textbf{FS-CORRELATION}}$$

$$\text{SameSlice}(P_1, P_2) = \begin{array}{l} \mathcal{V}_{\mathcal{W}}(P_1) = \mathcal{V}_{\mathcal{W}}(P_2) \wedge \\ \mathcal{V}_{\mathcal{S}}(P_1) \cap \mathcal{V}_{\mathcal{S}}(P_2) \neq \{\} \end{array}$$

$$\boxed{\textbf{FS-RELEVANCE}}$$

$$\text{IsRelevant}(Q, P) = \begin{array}{l} (\mathcal{V}(Q) \cap \mathcal{V}_{\mathcal{S}}(P) \neq \{\}) \vee \\ (\mathcal{V}_{\mathcal{S}}(P) = \{\} \wedge \mathcal{V}_{\mathcal{W}}(P) \subseteq \mathcal{V}(Q)) \end{array}$$

Figure 5-7. Annotated Slicing Mechanism

information, and thus has an empty set of strongly linking variables; *(iii)* constraints with weakly linking expressions or some weakly linking variables will express some relation between weakly linking entities and some other variables; thus neither set of weakly or strongly linking variables is empty. These observations allow us to support a uniform way of handling different kinds of linkages using a simpler variable-only annotation scheme.

5.6.3 Slicing Criterion

To take advantage of weakly connected components, our aggressive slicing mechanism will create partitions (or slices) by ignoring links that are due to solely weakly linking variables. This is achieved by allowing two constraints to be in the same slice if they satisfy the following two conditions: *(i)* they share one or more strongly linking variables, and *(ii)* they have the same set of weakly linking variables. These two conditions are captured in a new definition for the SAMESLICE meta-predicate in Fig. 5-7. According to this definition, each weakly linking constraint will be kept as a separate slice. Furthermore, two constraints that share the same set of weakly linking variables will only be kept in the same slice if they share one or more strongly linking variables.

The following lemma establishes the convergence of our splitting procedure

in the presence of the new meta-predicate.

Lemma 7 [SPLIT] *with* [FS–CORRELATION] *is convergent.*

5.6.4 Relevance Criterion

In the case of complete proof slicing, the constraints referring to a given property are spread across multiple slices. To have a good balance between precision and efficiency, we should ideally find the smallest set of hypotheses that ensure the success of the entailment check, whenever possible. To properly exploit the weakly linking annotations, we propose a two-step approach to finding relevant hypotheses. First, we employ aggressive slicing, which uses GETCTR_2, in order to obtain constraints that are most closely linked to the given goal. In case this first step fails, we may apply a subsequent exhaustive search step in order to identify additional constraints using a higher-level operator GETCTR_n, where n is the cardinality of our set of slices. Using n as a limit, our aggressive proof slicing mechanism has a similar behavior to that of complete proof slicing. We can formalize these two steps as instances of the slicing framework defined in Sec. 5.3.

Given a goal Q, the aggressive slicing mechanism would consider a slice *relevant* if either of the following holds:

1. It contains strongly linking variables that overlap with the free variables of Q.

2. It contains weakly linking constraints whose set of variables are entirely subsumed by the set of free variables of Q.

In order to collect these two categories of constraints, the calculus need only use GETCTR_2 in the aggressive search mechanism. The formalization of the

aggressive search relevance check is given by $\lfloor$FS–RELEVANCE$\rceil$ in Fig. 5-7. The condition $\mathcal{V}_S(P) = \{\}$ in the meta-predicate ISRELEVANT indicates that P is a slice of a weakly linking constraint.

5.7 Experiments

We have integrated the proposed proof slicing mechanisms into a separation logic-based program verification system [116], where proof obligations are soundly approximated by formulas in heap-free pure logic that can be discharged by off-the-shelf back-end theorem provers. The theorem provers used in our current evaluation are the Omega Calculator [132], MONA [90], Reduce/Redlog [56] and Z3 [52]. The proof slicing mechanisms are implemented as intermediate layers between the verifier and the theorem provers, effectively acting as prover-independent pre-processors for the back-end. In our measurements, we were careful to quantify the sole effect of applying the slicing procedures on the running time of the theorem provers (including overheads of the proof slicing mechanisms, if any) and show the relative comparison (on percentage) of timings by charts. For brevity, we use NS, CS and AS to indicate no, complete or aggressive proof slicing mechanism, respectively.

We used several benchmarks for evaluating the resulting system. The first benchmark includes a set of heap-manipulating programs, implementing typical operations for singly and doubly linked lists, as well as more complex tree data structures such as AVL and Red-Black trees. The benchmark also includes the BigInt program, which uses linked list to implement infinite precision integers and their arithmetic operations as well as the Karatsuba's fast multiplication method. The program is verified with non-linear constraints, which currently

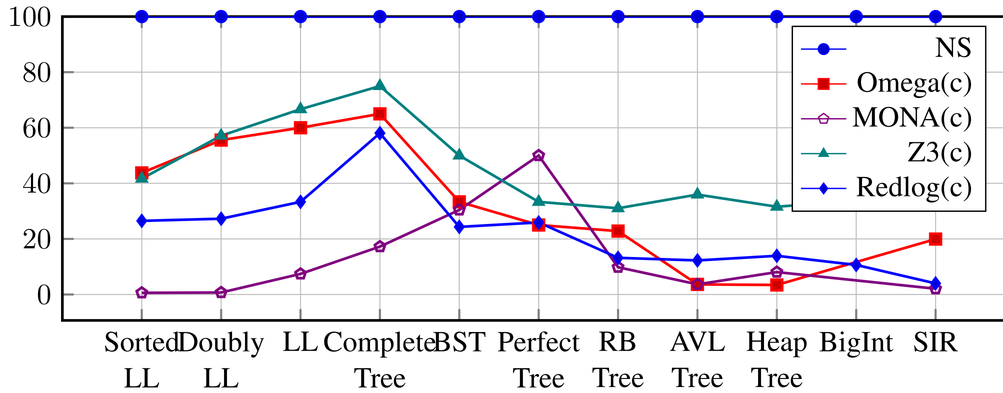


Figure 5-8. Relative Comparison (%) of CS over NS with various theorem provers.

can only be handled by the Redlog prover. The second benchmark consists of programs taken from the SIR/Siemens test suite [55] with some data structures mentioned above and arrays.

Fig. 5-8 shows the comparison on percentage between the time spent on each underlying prover plus slicing overhead when CS is on (indicating by the prover name with the postfix (c)) and the time spent on the same prover without proof slicing mechanism (NS) for the first two benchmarks. [3] As can be seen, CS benefits all provers in general, especially on complex programs (*e.g.*, BigInt and SIR) with over 60% reduction. Moreover, on less scalable provers like Omega, MONA or Redlog, CS helps to reduce about 90% of the total prover time (or 10x faster). Those significant improvements come from the reduction on proof size for both unsatisfiability and entailment proofs by the effect of proof slicing. For Z3, the total reduction on the prover time is about 60% despite its own optimization mechanisms (*e.g.*, the relevancy propagation technique). Because our proof slicing mechanisms focus on the *higher level* tasks of checking entailments and detecting unsatisfiability, they are able to filter out irrelevant constraints more effectively whenever the relationships

[3] We did not pay attention to the verification overhead because it is almost constant across different provers with and without proof slicing.

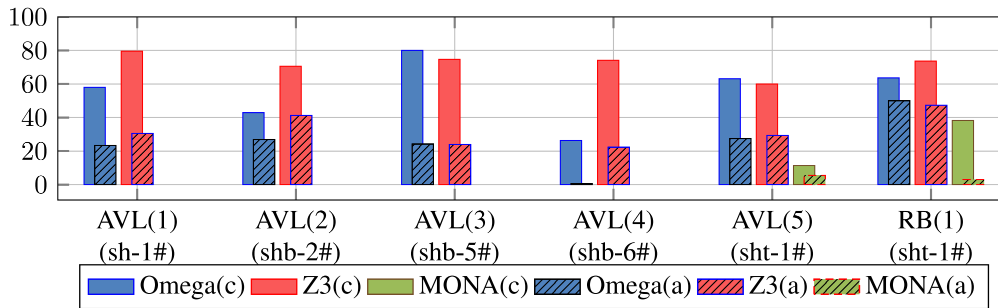


Figure 5-9. Comparison of CS (c) and AS (a) over NS on examples with Weakly Linking Components (s: size, h: height, b: balance factor, t: sets, n#: number of (annotated) weakly linking components)

between constraints are preserved. Moreover, with proof slicing, the unsatisfiability checks on the antecedents of entailment proofs are performed incrementally and non-redundantly, thus bringing more performance gains.

The next set of experiments concerns annotated formulas, and the application of AS. The inductive predicates of data structures used in this benchmark are augmented with additional *linking constraints* that enhance their precision to move towards verification of full functional correctness but also greatly increase the complexity of the derived proof obligations. Annotations for those linking constraints are inferred automatically, via a number of heuristics. For example, each parameter of a heap predicate is regarded as an independent property, unless it is mutually-dependent on another parameter, leading to an approach where every constraint between two distinct properties is always marked as *weakly linking*. Fig. 5-9 illustrates the performance benefits of AS over CS in the relative comparison with NS. It shows that in the presence of more complex specifications, AS performs better than its complete counterpart. In these examples, proof obligations with set constraints are discharged by MONA.

The fourth benchmark, called *Spaguetti*, came from the SLP tool [115]. It includes a set of heap-based test cases; each of them comprises 1000

randomly-generated, parameterized by the number of heap variables, UNSAT checks of the form $\mathtt{F} \vdash \mathtt{false}$ with the success rate about 50%. The SLP tool is an optimized paramodulation prover, hardwired to support only the list segment predicate, together with equality and disequality constraints on heap addresses and thus yielding a very good performance (under 3 seconds for each Spaguetti test case). With the help of AS together with a simple heuristic that automatically marks each disequality as a weakly linking constraint, our general-purpose separation logic-based prover is expected to achieve comparable performance while allowing a much more expressive specification language.

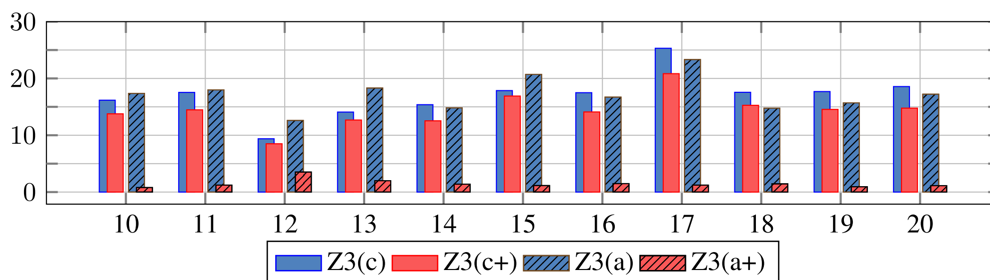


Figure 5-10. Comparison (%) of CS and AS over NS on the Spaguetti Benchmark with the number of heap variables from 10 to 20 (+ indicates caching used)

Unfortunately, as shown in Fig. 5-10, while the use of CS helps reduce the prover times with Z3 (by about 76.2% in total), AS has only little extra effect due to high numbers of (smaller) proofs generated. To obtain further improvements, we have augmented our proof slicing framework with a simple *proof caching mechanism* that memoizes on string representations of normalized proof obligations. This brought about over 90% reduction (after including overheads of both caching and slicing) when AS is used; thus the performance is now comparable to the SPL tool. This outcome is supported by a much higher hit rate (over 99%) from caching of smaller proofs generated by AS, as compared to the hit rate from the combination of proof caching and CS.

This effective result highlights the synergistic interplay between the proof caching and AS although the idea of proof caching is not new. Moreover, with the help of AS, an obsolete prover like Omega can catch up the performance of the advanced prover Z3 because the number of disequalities, which are expensively handled by Omega, is considerably reduced.

To investigate the portability of our proof slicing mechanisms, we have equipped AS for the Frama-C verification system [50]. For evaluation, we designed a family of contrived procedures, parameterized by the number of their parameters, that do computation on these independent variables, so as to illustrate the potential of AS. A version comprising two parameters is shown in Fig 5-11. Our AS (without proof caching) is interposed between the Frama-C verifier and the default Alt-Ergo prover. AS is supported by an annotation heuristic marking simple constraints of the form v=2 as weakly linking constraints. As can be seen from Fig. 5-12, the use of AS achieved good performance gains in conjunction with the default prover. We have also evaluated our proof slicing mechanism on a set of 20 small examples obtained from the Frama-C distribution, on which the use of proof slicing did not yield any noticeable gain. It remains our thesis that larger, more complex examples would, in general, benefit more from our proof slicing methods.

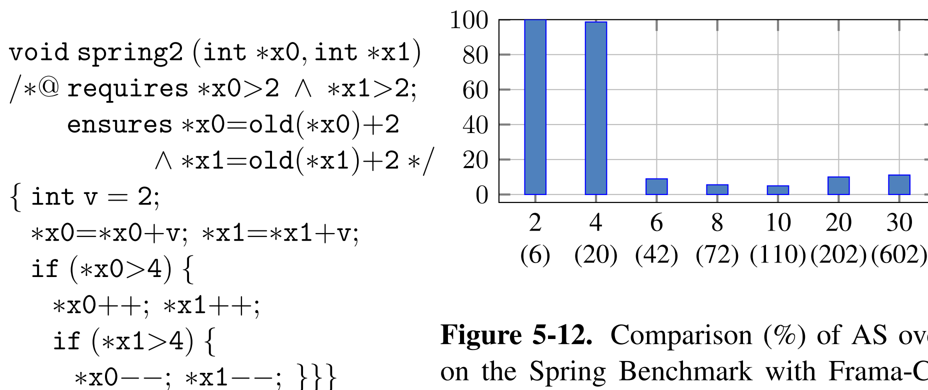


Figure 5-11. A simple contrived procedure

Figure 5-12. Comparison (%) of AS over NS on the Spring Benchmark with Frama-C. The number of parameters ranges from 2 to 30 and the number of generated proof obligations are given in the parentheses.

5.8 Discussion

The problem of filtering irrelevant information has been studied under different guises in several research areas. In [93], the authors focus on filtering out non-relevant information in knowledge bases. They discuss the concept of free variable independence for a conservative partitioning scheme and the concept of forgetting constraints, by which they eliminate irrelevant variables and produce the strongest consequent of the initial formula containing only relevant variables. However, the lack of an aggressive slicing mechanism (which in our case was supported by annotating weak links between distinct properties) leads to higher overheads in both the elimination and the solving phases.

Huang et al. [83] focus on slicing proofs for the infeasibility of counterexamples generated from a model checking process. The insight of this work is that global proofs can be sliced into independent proofs of atomic predicates, and memoization can be used to store the smaller proofs. While the general slicing technique has also been refined via a myriad of proposals (such as combined with abstract interpretation [138]), no mechanism has been proposed to allow a more flexible tradeoff of effectiveness versus conservatism in the slicing process.

Yet another direction of related research focuses on conservatively slicing formulas in connected components in order to simplify the satisfiability and entailment checks. In [6], Amir et al. introduce a methodology for representing large knowledge bases, namely sets of axioms, as trees of loosely connected partitions. They also define a message passing mechanism for reasoning over individual partitions. This has the effect of maintaining the linking information, but leading to higher overheads.

Simpler schemes, *e.g.*, conservative partitioning, have been proposed for

SAT solvers. The benefits of an union-find approach over the depth first search in identifying partitions are emphasized in [24]. In [143], a hypergraph cut method partitions the problem, then checks individual partitions and corroborates the results based on the assignments of the linking variables. In [121], SAT solvers are employed for each subproblem while delaying the assignments of linking variables to reduce the search space. In contrast to these methods, our approach refrains from converting implication checks into SAT checks, thus doing a better job at identifying weak linking constraints, and consequently yielding smaller proof slices. We also introduce customizable formula slicing capabilities that facilitate the exploration of new strategies. Our experiments shows that the approach is capable of speed gains without loss of completeness.

Finally, we mention Craig interpolation-based approaches, such as [75], that use interpolation to infer relevant predicates as a way of implementing abstraction refinement more efficiently. In these approaches, the notion of relevance is encoded in entailments and detected by an interpolating prover [112]. In contrast, relevance detection in our approach is largely syntactic, allowing the development of a generic proof slicing framework for automated program verification that would be effective for a broad range of off-the-shelf theorem provers used as back-end.

Chapter 6

Conclusions

The main aim of this thesis was to enhance the expressiveness and the scalability of total correctness proofs in program verification, especially termination and non-termination analysis. These enhancements would benefit the verification of large software systems with various complex program properties.

The first contribution of this study is a resource-based logic for termination and non-termination reasoning. Although termination reasoning has been intensively studied in the past, it remains a challenge for the technology developed there to keep up with improvements to specification logic infrastructure, and vice versa. We propose an approach that would combine the two areas more closely together, through a tightly coupled union. Our unique contribution is to embed both termination and non-termination reasoning directly into specification logics, and to do so with the help of a temporal entailment, developed in the form of resource reasoning. We show how the termination properties can be captured by a resource logic based on execution capacity, and how they could be abstracted into a flow-insensitive temporal logic. This approach has several benefits. Its expressiveness is immediately enhanced by any improvement to the underlying logics. It can also benefit from

infrastructures that have been developed for the underlying logics, including those that are related to program analysis. In particular, this approach allows us to use the safety specifications as a basis for termination specification inference. Last, but not least, it has placed termination and non-termination reasoning as a first-class concept, much like what was originally envisioned by Hoare's logic for total correctness.

The second contribution of this study is a modular inference framework for program termination and non-termination. By incorporating unknown pre/post temporal predicates into the specification logic for program termination/non-termination, our framework employs a Hoare-style forward verification to collect a set of relational assumptions to help soundly discover termination and non-termination properties. One major advantage of our technique is that it analyzes program termination and non-termination at the same time, and constructs a summary of termination behaviors for each method. This enables better modularity and reuse for our proving processes. Furthermore, it is integrated with a verification system allowing us to use partial correctness specification and to re-check our inference outcome. We have tested our implementation on a set of benchmark programs from a recent termination competition, and found our approach to compare favorably against the state-of-the-art termination analyzers.

The third contribution of this study is a formal proof slicing framework that allows the development of modular and extensible proof slicing mechanisms to enhance the scalability of verification systems. Relying on this general framework, two different proof slicing mechanisms have been constructed:

– the complete proof slicing mechanism, which is totally automated and

– the aggressive proof slicing mechanism, which is based on a lightweight

annotation scheme to achieve an even better performance. This mechanism can be automated with the development of annotation inference.

These mechanisms have been shown to be efficient and scalable by multi-fold reductions in verification times for each of the state-of-the-art theorem provers used as back-end of verification systems. The proof slicing framework is believed to be important for automated verification systems that are geared towards full functional correctness, where generated proof obligations are not only large and complex but may also be highly intertwined.

6.1 Future Work

For future work, we shall expand the existing verification infrastructure for broader application of the research presented in this thesis. There are several interesting directions for further investigation:

A possible avenue for future work is a verification framework for programs with real numbers and/or non-linear arithmetic, on top of which we develop a specification logic and inference mechanism for proving termination and non-termination of such programs. This framework would facilitate the application of our proposal to new areas, such as hybrid systems. With the recent progress on non-linear SMT solvers [62, 88], the construction of a verification system supporting nonlinear arithmetic would be feasible and promising.

Another interesting area for future work is the extension of current termination and non-termination analysis to array- and string-manipulating programs with or without pointer arithmetic. This extension is needed to complete our experiment with the SV-COMP's benchmarks. Like other

aspects, our starting point would be a verification and specification inference system for programs with arrays and strings. Based on the current infrastructure for separation logic in HIP/SLEEK system, we believe that the support for string and array would be straightforward and feasible once appropriate pure domain and segmented heap structures are suitably supported.

Lastly, the generation of counterexamples to termination should be investigated in future. However, we aim to provide a general framework that can generate and verify counterexamples to not only termination but also several other safety properties, such as memory safety. Based on the point of view that counterexamples are under-approximations which will be consumed when errors occur, the development of this feature in HIP/SLEEK would be useful for highlighting real bugs and how they were triggered.

Bibliography

[1] The Coq Proof Assistant - Version 8.4pl4. http://coq.inria.fr/, 2014.

[2] The Competition on Software Verification (SV-COMP), 2015.

[3] Andreas Abel. Type-based termination of generic programs. *Sci. Comput. Program.*, 74(8):550–567, 2009.

[4] Wolfgang Ahrendt, Thomas Baar, Bernhard Beckert, Richard Bubel, Martin Giese, Reiner Hhnle, Wolfram Menzel, Wojciech Mostowski, Andreas Roth, Steffen Schlager, and Peter H. Schmitt. The KeY tool. *Software and Systems Modeling*, 4(1):32–54, 2005.

[5] Elvira Albert, Puri Arenas, Samir Genaim, German Puebla, and Damiano Zanardini. COSTA: Design and Implementation of a Cost and Termination Analyzer for Java Bytecode. In *Int. Symposium on Formal Methods for Components and Objects*, pages 113–132, 2007.

[6] Eyal Amir and Sheila McIlraith. Partition-based logical reasoning for first-order and propositional theories. *Artif. Intell.*, 162:49–88, 2005.

[7] Hugh Anderson and Siau-Cheng Khoo. Affine-Based Size-Change Termination. In *Asian Symposium on Programming Languages and Systems*, pages 122–140, 2003.

[8] Andrew W. Appel and Sandrine Blazy. Separation Logic for Small-Step C minor. In *Int. Conf. on Theorem Proving in Higher Order Logics*, pages 5–21, 2007.

[9] Krzysztof R. Apt. Ten Years of Hoare's Logic: A Survey - Part I. *ACM Trans. Program. Lang. Syst.*, 3(4):431–483, 1981.

[10] David Aspinall, Lennart Beringer, Martin Hofmann, Hans-Wolfgang Loidl, and Alberto Momigliano. A program logic for resources. *Theor. Comput. Sci.*, 389(3):411–445, 2007.

[11] Mohamed Faouzi Atig, Ahmed Bouajjani, Michael Emmi, and Akash Lal. Detecting Fair Non-termination in Multithreaded Programs. In *Int. Conf. on Computer-Aided Verification*, pages 210–226, 2012.

[12] Robert Atkey. Amortised resource analysis with separation logic. *Logical Methods in Computer Science*, 7(2):85–103, 2011.

[13] Michael Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In *Int. Symposium on Formal Methods for Components and Objects*, pages 364–387, 2005.

[14] Clark W. Barrett and Sergey Berezin. CVC Lite: A New Implementation of the Cooperating Validity Checker Category B. In *Int. Conf. on Computer-Aided Verification*, pages 515–518, 2004.

[15] Clark W. Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In *Int. Conf. on Computer-Aided Verification*, pages 171–177, 2011.

[16] Clark W. Barrett and Cesare Tinelli. CVC3. In *Int. Conf. on Computer-Aided Verification*, pages 298–302, 2007.

[17] Patrick Baudin, Pascal Cuoq, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. ACSL: ANSI/ISO C Specification Language Version 1.8. *URL http://frama-c.com/acsl.html*, 2013.

[18] J. Berdine, C. Calcagno, and P. W. O'Hearn. Smallfoot: Modular Automatic Assertion Checking with Separation Logic. In *Int. Symposium on Formal Methods for Components and Objects*, pages 115–137, 2006.

[19] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. Symbolic Execution with Separation Logic. In *Asian Symposium on Programming Languages and Systems*, pages 52–68, 2005.

[20] Josh Berdine, Aziem Chawdhary, Byron Cook, Dino Distefano, and Peter W. O'Hearn. Variance analyses from invariance analyses. In *ACM Symposium on Principles of Programming Languages*, pages 211–224, 2007.

[21] Josh Berdine, Byron Cook, Dino Distefano, and Peter W. O'Hearn. Automatic Termination Proofs for Programs with Shape-Shifting Heaps. In *Int. Conf. on Computer-Aided Verification*, pages 386–400, 2006.

[22] Josh Berdine, Byron Cook, and Samin Ishtiaq. SLAyer: Memory Safety for Systems-Level Code. In *Int. Conf. on Computer-Aided Verification*, pages 178–183, 2011.

[23] Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. The Software Model Checker Blast: Applications to Software Engineering. *Int. J. Softw. Tools Technol. Transf.*, 9(5):505–525, 2007.

[24] Armin Biere and Carsten Sinz. Decomposing SAT problems into connected components. *J. on Satisfiability, Boolean Modeling and Computation*, 2(1-4):201–208, 2006.

[25] Richard Bornat, Cristiano Calcagno, Peter W. O'Hearn, and Matthew J. Parkinson. Permission accounting in separation logic. In *ACM Symposium on Principles of Programming Languages*, pages 259–270, 2005.

[26] Marius Bozga, Radu Iosif, and Filip Konečný. Deciding conditional termination. In *Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, pages 252–266, 2012.

[27] Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. Linear Ranking with Reachability. In *Int. Conf. on Computer-Aided Verification*, pages 491–504, 2005.

[28] Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. The Polyranking Principle. In *Int. Colloquium on Automata, Languages and Programming*, pages 1349–1361, 2005.

[29] Marc Brockschmidt, Byron Cook, and Carsten Fuhs. Better termination proving through cooperation. In *Int. Conf. on Computer-Aided Verification*, pages 413–429, 2013.

[30] Marc Brockschmidt, Thomas Ströder, Carsten Otto, and Jürgen Giesl. Automated Detection of Non-termination and NullPointerExceptions for Java Bytecode. In *Int. Conf. on Formal Verification of Object-Oriented Software*, pages 123–141, 2011.

[31] Stephen Brookes. A semantics for concurrent separation logic. *Theor. Comput. Sci.*, 375(1-3):227–270, 2007.

[32] James Brotherston, Richard Bornat, and Cristiano Calcagno. Cyclic proofs of program termination in separation logic. In *ACM Symposium on Principles of Programming Languages*, pages 101–112, 2008.

[33] Samuel R. Buss. An introduction to proof theory. In *Handbook of Proof Theory, Studies in Logic and the Foundations of Mathematics*, pages 1–78. Elsevier, 1998.

[34] Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. Compositional shape analysis by means of bi-abduction. In *ACM Symposium on Principles of Programming Languages*, pages 289–300, 2009.

[35] Hong-Yi Chen, Byron Cook, Carsten Fuhs, Kaustubh Nimkar, and Peter O'Hearn. Proving nontermination via safety. In *Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, pages 156–171, 2014.

[36] Wei-Ngan Chin, Cristina David, Huu Hai Nguyen, and Shengchao Qin. Multiple pre/post specifications for heap-manipulating methods. In *Int. Symposium on High Assurance Systems Engineering*, pages 357–364, 2007.

[37] Wei-Ngan Chin, Cristina David, Huu Hai Nguyen, and Shengchao Qin. Enhancing modular OO verification with separation logic. In *ACM Symposium on Principles of Programming Languages*, pages 87–99, 2008.

[38] Wei-Ngan Chin, Cristina David, Huu Hai Nguyen, and Shengchao Qin. Automated verification of shape, size and bag properties via user-defined predicates in separation logic. *Sci. Comput. Program.*, 77(9):1006–1036, 2012.

[39] Edmund M. Clarke and Jeannette M. Wing. Formal Methods: State of the Art and Future Directions. *ACM Comput. Surv.*, 28(4):626–643, 1996.

[40] Edmund M. Clarke, Jr., Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, 1999.

[41] Michael Colon and Henny Sipma. Synthesis of Linear Ranking Functions. In *Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, pages 67–81, 2001.

[42] Michael Colon and Henny Sipma. Practical Methods for Proving Program Termination. In *Int. Conf. on Computer-Aided Verification*, pages 442–454, 2002.

[43] Byron Cook, Sumit Gulwani, Tal Lev-Ami, Andrey Rybalchenko, and Mooly Sagiv. Proving Conditional Termination. In *Int. Conf. on Computer-Aided Verification*, pages 328–340, 2008.

[44] Byron Cook, Daniel Kroening, Philipp Rümmer, and Christoph M. Wintersteiger. Ranking Function Synthesis for Bit-Vector Relations. In *Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, pages 236–250, 2010.

[45] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Abstraction refinement for termination. In *Int. Symposium on Static Analysis*, pages 87–101, 2005.

[46] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Termination proofs for systems code. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 415–426, 2006.

[47] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Proving thread termination. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 320–330, 2007.

[48] Byron Cook, Abigail See, and Florian Zuleger. Ramsey vs. Lexicographic Termination Proving. In *Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, pages 47–61, 2013.

[49] Patrick Cousot. Proving Program Invariance and Termination by Parametric Abstraction, Lagrangian Relaxation and Semidefinite Programming. In *Int. Conf. on Verification, Model Checking, and Abstract Interpretation*, pages 1–24, 2005.

[50] Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-C - A Software Analysis Perspective. In *Int. Conf. on Software Engineering and Formal Methods*, pages 233–247, 2012.

[51] Leonardo de Moura and Nikolaj Bjørner. Relevancy propagation. Technical report, Microsoft Research, 2007.

[52] Leonardo de Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In *Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, 2008.

[53] Edsger W. Dijkstra. The humble programmer. *Commun. ACM*, 15(10):859–866, 1972.

[54] Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. A Local Shape Analysis Based on Separation Logic. In *Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, pages 287–302, 2006.

[55] Hyunsook Do, Sebastian Elbaum, and Gregg Rothermel. Supporting Controlled Experimentation with Testing Techniques: An Infrastructure and its Potential Impact. *Empirical Softw. Engg.*, 10(4):405–435, 2005.

[56] Andreas Dolzmann and Thomas Sturm. REDLOG: Computer Algebra meets Ccomputer Logic. *SIGSAM Bulletin*, 31:2–9, 1997.

[57] Kamil Dudka, Petr Peringer, and Tomáš Vojnar. Predator: A Practical Tool for Checking Manipulation of Dynamic Data Structures Using Separation Logic. In *Int. Conf. on Computer-Aided Verification*, pages 372–378, 2011.

[58] Stephan Falke and Deepak Kapur. A term rewriting approach to the automated termination analysis of imperative programs. In *Int. Conf. on Automated Deduction*, pages 277–293, 2009.

[59] Arnaud Fietzke and Christoph Weidenbach. Labelled splitting. In *Int. Joint Conf. on Automated Reasoning*, pages 459–474, 2008.

[60] Robert W. Floyd. Assigning meanings to programs. In *Mathematical Aspects of Computer Science*, volume 19, pages 19–32, 1967.

[61] Pierre Ganty and Samir Genaim. Proving termination starting from the end. In *Int. Conf. on Computer-Aided Verification*, pages 397–412, 2013.

[62] Sicun Gao, Soonho Kong, and Edmund M Clarke. dReal: An SMT solver for nonlinear theories over the reals. In *Int. Conf. on Automated Deduction*, pages 208–214, 2013.

[63] Cristian Gherghina, Cristina David, Shengchao Qin, and Wei-Ngan Chin. Structured specifications for better verification of heap-manipulating programs. In *Int. Symposium on Formal Methods*, pages 386–401, 2011.

[64] Jürgen Giesl, Marc Brockschmidt, Fabian Emmes, Florian Frohn, Carsten Fuhs, Carsten Otto, Martin Plücker, Peter Schneider-Kamp, Thomas Ströder, Stephanie Swiderski, and René Thiemann. Proving Termination of Programs Automatically with AProVE. In *Int. Joint Conf. on Automated Reasoning*, pages 184–191, 2014.

[65] Jürgen Giesl, Peter Schneider-Kamp, and René Thiemann. AProVE 1.2: Automatic Termination Proofs in the Dependency Pair Framework. In *Int. Joint Conf. on Automated Reasoning*, pages 281–286, 2006.

[66] Arnaud Gotlieb and Matthieu Petit. Towards a theory for testing non-terminating programs. In *Int. Computer Software and Applications Conf.*, pages 160–165, 2009.

[67] Alexey Gotsman, Josh Berdine, Byron Cook, Noam Rinetzky, and Mooly Sagiv. Local reasoning for storable locks and threads. In *Asian Symposium on Programming Languages and Systems*, pages 19–37, 2007.

[68] Sumit Gulwani, Krishna K. Mehra, and Trishul M. Chilimbi. SPEED: precise and efficient static estimation of program computational

complexity. In *ACM Symposium on Principles of Programming Languages*, pages 127–139, 2009.

[69] Sumit Gulwani, Saurabh Srivastava, and Ramarathnam Venkatesan. Program Analysis As Constraint Solving. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 281–292, 2008.

[70] Ashutosh Gupta, Thomas A. Henzinger, Rupak Majumdar, Andrey Rybalchenko, and Ru-Gang Xu. Proving non-termination. In *ACM Symposium on Principles of Programming Languages*, pages 147–158, 2008.

[71] William R. Harris, Akash Lal, Aditya V. Nori, and Sriram K. Rajamani. Alternation for Termination. In *Int. Symposium on Static Analysis*, pages 304–319, 2010.

[72] Eric C. R. Hehner. Termination is timing. In *Int. Conf. on Mathematics of Program Construction*, pages 36–47, 1989.

[73] Eric C. R. Hehner. Specifications, programs, and total correctness. *Sci. Comput. Program.*, 34(3):191–205, 1999.

[74] Matthias Heizmann, Jochen Hoenicke, and Andreas Podelski. Termination Analysis by Learning Terminating Programs. In *Int. Conf. on Computer-Aided Verification*, pages 797–813, 2014.

[75] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Kenneth L. McMillan. Abstractions from proofs. In *ACM Symposium on Principles of Programming Languages*, pages 232–244, 2004.

[76] Charles Antony Richard Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[77] Charles Antony Richard Hoare and He Jifeng. *Unifying Theories of Programming*. Prentice-Hall, 1998.

[78] Aquinas Hobor, Andrew W. Appel, and Francesco Zappa Nardelli. Oracle semantics for concurrent separation logic. In *European Symposium on Programming Languages and Systems*, pages 353–367, 2008.

[79] Aquinas Hobor and Cristian Gherghina. Barriers in Concurrent Separation Logic: Now With Tool Support! *Logical Methods in Computer Science*, 8(2), 2012.

[80] Aquinas Hobor and Jules Villard. The ramifications of sharing in data structures. In *ACM Symposium on Principles of Programming Languages*, pages 523–536, 2013.

[81] Jan Hoffmann and Martin Hofmann. Amortized resource analysis with polynomial potential. In *European Symposium on Programming Languages and Systems*, pages 287–306, 2010.

[82] Gerard J Holzmann. The model checker SPIN. *IEEE Transactions on software engineering*, 23(5):279–295, 1997.

[83] Hai Huang, Wei-Tek Tsai, and Raymond A. Paul. Proof slicing with application to model checking web services. In *IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, pages 292–299, 2005.

[84] Samin S. Ishtiaq and Peter W. O'Hearn. BI as an assertion language for mutable data structures. In *ACM Symposium on Principles of Programming Languages*, pages 14–26, 2001.

[85] Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java. In *NASA International Symposium on Formal Methods*, pages 41–55, 2011.

[86] Jonas B. Jensen, Nick Benton, and Andrew Kennedy. High-level separation logic for low-level code. In *ACM Symposium on Principles of Programming Languages*, pages 301–314, 2013.

[87] Cliff B. Jones. Balancing Expressiveness in Formal Approaches to Concurrency. Technical Report CS-TR-1394, Newcastle University, 2013.

[88] Dejan Jovanovic and Leonardo Mendonça de Moura. Solving non-linear arithmetic. In *Int. Joint Conf. on Automated Reasoning*, pages 339–354, 2012.

[89] Matt Kaufmann and J Strother Moore. An ACL2 tutorial. In *Int. Conf. on Theorem Proving in Higher Order Logics*, pages 17–21, 2008.

[90] Nils Klarlund and Anders Møller. MONA Version 1.4 - User Manual. BRICS Notes Series, 2001.

[91] Stephen C. Kleene. *Mathematical logic*. Wiley, 1967.

[92] Daniel Kroening, Natasha Sharygina, Aliaksei Tsitovich, and Christoph M. Wintersteiger. Termination Analysis with Compositional Transition Invariants. In *Int. Conf. on Computer-Aided Verification*, pages 89–103, 2010.

[93] Jôme Lang, Paolo Liberatore, and Pierre Marquis. Propositional independence: formula-variable independence and forgetting. *J. Artif. Int. Res.*, 18:391–443, 2003.

[94] Daniel Larraz, Kaustubh Nimkar, Albert Oliveras, Enric Rodriguez-Carbonell, and Albert Rubio. Proving Non-termination Using Max-SMT. In *Int. Conf. on Computer-Aided Verification*, pages 779–796, 2014.

[95] Daniel Larraz, Albert Oliveras, Enric Rodriguez-Carbonell, and Albert Rubio. Proving termination of imperative programs using Max-SMT. In *Conf. on Formal Methods in Computer-Aided Design*, pages 218–225, 2013.

[96] Kim G Larsen, Paul Pettersson, and Wang Yi. Uppaal in a nutshell. *Int. J. Softw. Tools Technol. Transf.*, 1(1):134–152, 1997.

[97] Quang Loc Le, Cristian Gherghina, Shengchao Qin, and Wei-Ngan Chin. Shape Analysis via Second-Order Bi-Abduction. In *Int. Conf. on Computer-Aided Verification*, pages 52–68, 2014.

[98] Ton Chanh Le, Cristian Gherghina, Aquinas Hobor, and Wei-Ngan Chin. A resource-based logic for termination and non-termination proofs. In *Int. Conf. on Formal Engineering Methods*, pages 267–283, 2014.

[99] Ton Chanh Le, Cristian Gherghina, Razvan Voicu, and Wei-Ngan Chin. A Proof Slicing Framework for Program Verification. In *Int. Conf. on Formal Engineering Methods*, pages 53–69, 2013.

[100] Ton Chanh Le, Shengchao Qin, and Wei-Ngan Chin. Termination and non-termination specification inference. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation*, 2015. to appear.

[101] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: A behavioral interface specification language for Java. *ACM SIGSOFT Software Engineering Notes*, 31(3):1–38, 2006.

[102] Chin Soon Lee, Neil D. Jones, and Amir M. Ben-Amram. The size-change principle for program termination. In *ACM Symposium on Principles of Programming Languages*, pages 81–92, 2001.

[103] Oukseh Lee, Hongseok Yang, and Rasmus Petersen. Program Analysis for Overlaid Data Structures. In *Int. Conf. on Computer-Aided Verification*, pages 592–608, 2011.

[104] Wonchan Lee, Bow-Yaw Wang, and Kwangkeun Yi. Termination analysis with algorithmic learning. In *Int. Conf. on Computer-Aided Verification*, pages 88–104, 2012.

[105] Wonyeol Lee and Sungwoo Park. A proof system for separation logic with magic wand. In *ACM Symposium on Principles of Programming Languages*, pages 477–490, 2014.

[106] K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In *Int. Conf. on Logic for Programming, Artificial Intelligence, and Reasoning*, pages 348–370, 2010.

[107] K. Rustan M. Leino, Michal Moskal, and Wolfram Schulte. Verification condition splitting. Technical report, Microsoft Research, 2008.

[108] Jacques-Louis Lions et al. Ariane 5 Flight 501 failure, 1996.

[109] Stephen Magill, Ming-Hsien Tsai, Peter Lee, and Yih-Kuen Tsay. THOR: A Tool for Reasoning about Shape and Arithmetic. In *Int. Conf. on Computer-Aided Verification*, pages 428–432, 2008.

[110] Stephen Magill, Ming-Hsien Tsai, Peter Lee, and Yih-Kuen Tsay. Automatic numeric abstractions for heap-manipulating programs. In *ACM Symposium on Principles of Programming Languages*, pages 211–222, 2010.

[111] Claude Marché and Hans Zantema. The termination competition. In *Int. Conf. on Term Rewriting and Applications*, pages 303–313, 2007.

[112] Kenneth L. McMillan. An interpolating theorem prover. In *Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, pages 16–30, 2004.

[113] Jia Meng and Lawrence C. Paulson. Lightweight relevance filtering for machine-generated resolution problems. *Journal of Applied Logic*, 7(1):41–57, 2009.

[114] Keiko Nakata and Tarmo Uustalu. A Hoare logic for the coinductive trace-based big-step semantics of While. In *European Symposium on Programming Languages and Systems*, pages 488–506, 2010.

[115] Juan Antonio Navarro Pérez and Andrey Rybalchenko. Separation logic + superposition calculus = heap theorem prover. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 556–566, 2011.

[116] Huu Hai Nguyen, Cristina David, Shengchao Qin, and Wei-Ngan Chin. Automated verification of shape and size properties via separation logic. In *Int. Conf. on Verification, Model Checking, and Abstract Interpretation*, pages 251–266, 2007.

[117] Peter W. O'Hearn. Resources, concurrency, and local reasoning. *Theor. Comput. Sci.*, 375(1-3):271–307, 2007.

[118] Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. In *Int. Workshop on Computer Science Logic*, pages 1–19, 2001.

[119] Peter W. O'Hearn, Hongseok Yang, and John C. Reynolds. Separation and information hiding. *ACM Trans. Program. Lang. Syst.*, 31(3), 2009.

[120] Carsten Otto, Marc Brockschmidt, Christian von Essen, and Jürgen Giesl. Automated Termination Analysis of Java Bytecode by Term Rewriting. In *Int. Conf. on Term Rewriting and Applications*, pages 259–276, 2010.

[121] Tai Joon Park and Allen Van Gelder. Partitioning methods for satisfiability testing on large formulas. In *Int. Conf. on Automated Deduction*, pages 748–762, 1996.

[122] Matthew J. Parkinson and Gavin M. Bierman. Separation Logic, Abstraction and Inheritance. In *ACM Symposium on Principles of Programming Languages*, pages 75–86, 2008.

[123] Étienne Payet and Fausto Spoto. Experiments with Non-Termination Analysis for Java Bytecode. *Electr. Notes Theor. Comput. Sci.*, 253(5):83–96, 2009.

[124] Charles Sanders Peirce. *Collected papers of Charles Sanders Peirce*. Harvard University Press., 1958.

[125] Ruzica Piskac, Thomas Wies, and Damien Zufferey. Automating Separation Logic with Trees and Data. In *Int. Conf. on Computer-Aided Verification*, pages 711–728, 2014.

[126] Andreas Podelski and Andrey Rybalchenko. A Complete Method for the Synthesis of Linear Ranking Functions. In *Int. Conf. on Verification, Model Checking, and Abstract Interpretation*, pages 239–251, 2004.

[127] Andreas Podelski and Andrey Rybalchenko. Transition Invariants. In *IEEE Symposium on Logic in Computer Science*, pages 32–41, 2004.

[128] Andreas Podelski and Andrey Rybalchenko. Transition predicate abstraction and fair termination. In *ACM Symposium on Principles of Programming Languages*, pages 132–144, 2005.

[129] Andreas Podelski and Andrey Rybalchenko. ARMC: The Logical Choice for Software Model Checking with Abstraction Refinement. In *Int. Symposium on Practical Aspects of Declarative Languages*, pages 245–259, 2007.

[130] Corneliu Popeea and Wei-Ngan Chin. Inferring Disjunctive Postconditions. In *Asian Computing Science Conf.*, pages 331–345, 2006.

[131] Corneliu Popeea and Wei-Ngan Chin. Dual analysis for proving safety and finding bugs. In *ACM Symposium on Applied Computing*, pages 2137–2143, 2010.

[132] William Pugh. The Omega test: a fast and practical integer programming algorithm for dependence analysis. In *Supercomputing*, pages 4–13, 1991.

[133] John C. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *IEEE Symposium on Logic in Computer Science*, pages 55–74, 2002.

[134] John C. Reynolds. An overview of separation logic. In *Int. Conf. on Verified Software: Theories, Tools, Experiments*, pages 460–469. 2008.

[135] Alan JA Robinson and Andrei Voronkov. *Handbook of automated reasoning*, volume 2. Elsevier, 2001.

[136] Andrey Rybalchenko. Constraint Solving for Program Verification: Theory and Practice by Example. In *Int. Conf. on Computer-Aided Verification*, pages 57–71, 2010.

[137] Alexander Schrijver. *Theory of Linear and Integer Programming*. John Wiley & Sons, Inc., New York, NY, USA, 1986.

[138] Hyoung Seok Hong, Insup Lee, and Oleg Sokolsky. Abstract Slicing: A New Approach to Program Slicing Based on Abstract Interpretation and Model Checking. In *IEEE Int. Workshop on Source Code Analysis and Manipulation*, pages 25–34, 2005.

[139] Uffe Sørensen. Slicing for Uppaal. Technical report, AALBORG University, 2008.

[140] Aaron Stump, Clark W. Barrett, and David L. Dill. CVC: A Cooperating Validity Checker. In *Int. Conf. on Computer-Aided Verification*, pages 500–504, 2002.

[141] Kasper Svendsen, Lars Birkedal, and Matthew Parkinson. Modular Reasoning About Separation of Concurrent Data Structures. In *European Symposium on Programming Languages and Systems*, pages 169–188, 2013.

[142] Gang Tan. A Collection of Well-Known Software Failures. http://www.cse.lehigh.edu/~gtan/bug/softwarebug.html, 2009.

[143] Jose Torres-Jimenez, Luis Vega-Garcia, CA Coutino-Gomez, and FJ Cartujano-Escobar. SSTP: An Approach to Solve SAT Instances Through Partition. *WSEAS Trans. on Computer*, 3:1482–1487, 2004.

[144] Viktor Vafeiadis. Concurrent separation logic and operational semantics. *Electron. Notes Theor. Comput. Sci.*, 276:335–351, 2011.

[145] Viktor Vafeiadis and Matthew J. Parkinson. A Marriage of Rely/Guarantee and Separation Logic. In *Conf. on Concurrency Theory*, pages 256–271, 2007.

[146] Helga Velroyen and Philipp Rümmer. Non-termination checking for imperative programs. In *Int. Conf. on Tests and Proofs*, pages 154–170, 2008.

[147] Jules Villard, Étienne Lozes, and Cristiano Calcagno. Tracking Heaps That Hop with Heap-Hop. In *Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, pages 275–279, 2010.

[148] Hongwei Xi. Dependent types for program termination verification. In *IEEE Symposium on Logic in Computer Science*, pages 231–242, 2001.

[149] Hongseok Yang, Oukseh Lee, Josh Berdine, Cristiano Calcagno, Byron Cook, Dino Distefano, and Peter W. O'Hearn. Scalable shape analysis for systems code. In *Int. Conf. on Computer-Aided Verification*, pages 385–398, 2008.

Appendix A

A.1 Proofs for Lemmas

Lemma 1 (Resource Entailments) *Given resource assertions ρ, ρ_1 and ρ_2, $\rho \vdash \rho_1 \blacktriangleright \rho_2$ iff $\forall s, h, r, r_1 \cdot$ if $(s, h, r) \models \rho$ and $(s, h, r_1) \models \rho_1$ then $(s, h, r \ominus r_1) \models \rho_2$.*

Proof. We have

$$\begin{aligned}
& \rho \vdash \rho_1 \blacktriangleright \rho_2 \\
\equiv\ & \forall s, h, r \cdot \text{ if } (s, h, r) \models \rho \text{ then } (s, h, r) \models \rho_1 \blacktriangleright \rho_2 \\
& \hfill \text{(Defn. of logical entailment)} \\
\equiv\ & \forall s, h, r \cdot \text{ if } (s, h, r) \models \rho \text{ then } \forall r_1 \cdot \\
& \qquad \text{if } (s, h, r_1) \models \rho_1 \text{ then } (s, h, r \ominus r_1) \models \rho_2 \text{ (Semantics of } \rho_1 \blacktriangleright \rho_2) \\
\equiv\ & \forall s, h, r, r_1 \cdot \text{ if } (s, h, r) \models \rho \text{ and } (s, h, r_1) \models \rho_1 \text{ then } (s, h, r \ominus r_1) \models \rho_2 \ \square
\end{aligned}$$

Lemma 2 *If the termination of a program can be proven by a given lexicographic termination measure, then for each call tree τ of the program, every element of the termination measure applied to the program states corresponding to the nodes in the call tree τ is bounded.*

Proof. As the program can be proven to terminate by the measure $[X_n, X_{n-1}, \ldots, X_0]$, the corresponding evaluation call tree is finite. The reason

is that if the call tree is infinite then by König's lemma [91], there is an infinite evaluation path, which means that the program is non-terminating.

Let S_i be the set of evaluating values of X_i and N be the finite number of the tree's nodes. Then, for all i, the cardinality $|S_i| \leq N$ or S_i is finite. As a result, the set $S = \bigcup S_i$ is also finite. The maximum value k of S is the upper bound of every element of the given termination measure. □

Lemma 3 *For all* $x_n, \ldots, x_0, y_n, \ldots, y_0 \in \mathbb{N}$ *such that* $\forall i \in \{0..n-1\} \cdot x_i, y_i < b$, $[x_n, \ldots, x_0] >_l [y_n, \ldots, y_0]$ *iff* $\mathcal{D}([x_n, \ldots, x_0]) > \mathcal{D}([y_n, \ldots, y_0])$, *where* $>_l$ *is the lexicographic ordering.*

Proof. ($\Rightarrow$) From the premise, we have $\exists i \in \{0..n\} \cdot x_n = y_n \wedge \ldots \wedge x_{i+1} = y_{i+1} \wedge x_i > y_i$. Consequently, $x_i - y_i \geq 1$. Moreover, because $\forall i \cdot 0 \leq x_i, y_i < b$, we also have $1-b \leq x_i - y_i \leq b-1$. Let consider

$$\begin{aligned} & \mathcal{D}([x_n, x_{n-1}, \ldots, x_0]) - \mathcal{D}([y_n, y_{n-1}, \ldots, y_0]) \\ = \; & (x_i - y_i) * b^i + (x_{i-1} - y_{i-1}) * b^{i-1} + \ldots + (x_0 - y_0) \\ \geq \; & b^i + (1-b) * (b^{i-1} + \ldots + 1) = b^i + (1 - b^i) = 1 > 0 \end{aligned}$$

Thus, $\mathcal{D}([x_n, \ldots, x_0]) > \mathcal{D}([y_n, \ldots, y_0])$.

($\Leftarrow$) By contradiction, assume that $[x_n, \ldots, x_0] <_l [y_n, \ldots, y_0]$. Similarly to the above proof, we have $\mathcal{D}([y_n, \ldots, y_0]) > \mathcal{D}([x_n, \ldots, x_0])$, which is a contradiction. Moreover, if $\forall i \cdot x_i = y_i$ then $\mathcal{D}([x_n, \ldots, x_0]) = \mathcal{D}([y_n, \ldots, y_0])$. As a result, $[x_n, \ldots, x_0] >_l [y_n, \ldots, y_0]$. □

A.2 Soundness Proofs

Our goal here is to prove the soundness of our resource-aware Hoare logic for execution lengths. First, we outline an operational semantics for the verified

strict imperative language. Second, we define a Hoare triple with respect to this operational semantics and prove the soundness of our Hoare rules, *i.e.*, the operational semantics would get stuck on executions starting in states that falsify the resource assertions.

Operational semantics. We have modified a standard small-step operational semantics to incorporate the execution capacity. In Fig. A-1, we list only the method call and return steps; the other steps do not interact with the execution capacity in any interesting way. As mentioned previously, our core language does not have loops. Therefore, execution capacity is only consumed at method calls.

The formulation of the method call step ensures that *at least one* execution step is still allowed by the current execution capacity, via the capacity subtraction $r \ominus (1, 1)$, corresponding to the first entailment in the verification rule $\lfloor \textsc{fv-call} \rfloor$ in Fig. 3-6. As a result, the semantics will not allow (*e.g.*, eventually get stuck on) executions which requires more resource than the available resource upper bound in the initial states. For example, the semantics will not allow infinite executions from states in which the capacity has finite values.

The return operational rule ensures that executions do not finish if the resource lower bound has not been consumed all. That is, the operational semantics prohibits the return step if the call stack has height 1 and the execution capacity has a non zero lower bound, which would equate with a return from the outermost method before all the required steps have been taken.

Hoare Triples. We define the Hoare triple in a continuation-passing style as in Appel and Blazy [8]. A *configuration* is a pair of code k and state σ. We say

$$\frac{t_0\ mn((t\ v)^*)\ \{code\} \in Prog}{<(s,h,r), mn(w^*)> \hookrightarrow <([v \mapsto s[w]]^*{:}s, h, r \ominus (1,1)), code>}$$

$$\frac{}{<(s_t{:}s,h,r), \texttt{return}\ v> \hookrightarrow <s\ |_{res \mapsto s_t[v]}, h, r, \mathrm{nop}>}$$

$$\frac{r = (0, _)}{<([s_t], h, r), \texttt{return}> \hookrightarrow <[], h, r, \mathrm{nop}>}$$

Figure A-1. Key Rules in Operational Semantics

a configuration is *safe*, written $\mathsf{safe}(k, \sigma)$, if all reachable states are safely halted or can continue to step:

$$\begin{aligned}\mathsf{safe}(k,\sigma) \ \equiv\ & \forall k', \sigma' \cdot\ <\sigma, k> \hookrightarrow^* <(s', h', r'), k'> \rightarrow \\ & ((k' = \mathrm{nop} \wedge s' = [] \wedge r' = (0, _) \ \vee\ \exists \sigma'', k'' \cdot <\sigma', k'> \hookrightarrow <\sigma'', k''>)\end{aligned}$$

We say that a formula P guards code k, written $\mathsf{guards}(P, k)$ when the code k is safe on any state accepted by P:

$$\mathsf{guards}(P, k) \ \equiv\ \forall \sigma \cdot\ \sigma \models P \ \rightarrow\ \mathsf{safe}(k, \sigma)$$

We now define the Hoare triple $\{\Psi\}c\{\Phi\}$ in a continuation passing style using guards:

$$\{\Psi\}c\{\Phi\} \ \equiv\ \forall k \cdot\ \mathsf{guards}(\Phi, k) \ \rightarrow\ \mathsf{guards}(\Psi, c; k)$$

Note that we dramatically simplified Appel and Blazy's Hoare tuple to include just enough detail to indicate how the temporal assertions fit into the setup without overwhelming the presentation. We conclude by stating the key soundness theorems.

Theorem 2 (Safety) *If* $\vdash \{\Psi\}\, c\, \{\Phi\}$ *then* $\forall \sigma \cdot \sigma \models \Psi \rightarrow \mathsf{safe}(c, \sigma)$.

Proof. For all σ, $\mathsf{safe}(\texttt{nop}, \sigma)$, so for all Φ, $\mathsf{guards}(\Phi, \texttt{nop})$. If we instantiate $k = \texttt{nop}$ in the Hoare triple definition then safety follows immediately. □

In addition, by guaranteeing that the Hoare tuple $\{\Psi\}c\{\Phi\}$ holds, the safety theorem also implies that the postcondition holds after the execution of the code c. More precisely, this style of Hoare tuple implies the expected soundness property for any decidable postcondition.

Proposition 3 *If* $\vdash \{\Psi\}\, c\, \{\Phi\}$ *then* Φ *holds after the execution of the code* c*.*

Proof. To show that if the Hoare tuple $\{\Psi\}c\{\Phi\}$ holds then the postcondition Φ holds, we design the continuation k as a "tester" program that tests the resulting state and gets stuck if the test fails, otherwise does nothing. For example, consider a postcondition $\Phi = x{>}3$, we can use a continuation k:

$$k = \texttt{if}\ (\texttt{x} > 3)\ \texttt{then}\ \texttt{skip}\ \texttt{else}\ \texttt{get_stuck}$$

such that k will be safe iff the state of the machine after c's execution satisfies Φ. Thus, we know Φ guards k. We can feed that fact into our Hoare tuple to get Ψ guards $c; k$. Therefore we know that either: (i) c does not terminate, or (ii) c does terminate, and the resulting state is enough to make k safe, which implies (by k's construction) that Φ holds after c terminates. □

Theorem 4 *The standard Hoare rules (*e.g.*, assignment, conditional, sequential composition) are sound with respect to the semantics of our Hoare judgment.*

Proof. In [8], it is proven sound a set of Hoare rules very similar to ours for a language that has many of the same features, *e.g.*, load/ store/ assignment/ conditional, making the proofs of these features very similar. □

Theorem 5 *The Hoare rules for method call and return are sound.*

Proof. The proof of the return rule is standard, except in the case of returning from a method requiring the resource assertion $\mathtt{RC}\langle l, u\rangle$ whose the lower bound l is larger than the actual execution length of the method. In this case, operational semantics must get stuck. The Hoare rule for return requires that the $\mathsf{CheckMin}$ predicate holds, meaning that return is not executed with any such precondition. That is, the Hoare rule prohibits the execution of a return from a program state with $r = (l, _)$ where $l > 0$, which describes a superset of the states in which the operational semantics would block when executing a return step. Specifically, a proper Hoare derivation guarantees that non-terminating code never returns since the lower bound $l = \infty$ has never been consumed all.

The proof for the method call rule hinges on the proof that the precondition guarantees that there exists an execution capacity with a smaller upper bound that suffices for the callee. The resource-enhanced entailment from Sec. 3.3.3 and the Hoare rule for method call guarantee exactly this, meaning that a proper Hoare derivation guarantees that all function calls requires smaller upper bounds in their execution capacities than the available resource in the current program state. Hence, this guarantees the upper bound requirement of the resource assertion is never violated. □

We have used an operational semantics enriched with execution counters to show that a proper Hoare derivation guarantees that the operational semantics never blocks in accordance with the resource specifications. However, the execution counters do not have a counterpart in a "real machine" as modelled by a standard operational semantics. Below we will outline one such standard, erased semantics and show that our enriched semantics is a strict subset of the erased semantics. Thus the soundness results for our resource logic with

regards to the enriched semantics simply carry to the erased semantics.

$$\frac{t_0\ mn((t\ v)^*)\ \{code\} \in Prog}{<(s, h), mn(w^*)> \hookrightarrow <([v \mapsto s[w]]^*{:}s, h), code>}$$

$$\frac{}{<(s_t{:}s, h), \texttt{return}\ v> \hookrightarrow <(s\ |_{res \mapsto s_t[v]}, h), \mathtt{nop}>}$$

Theorem 6 (Erasure) *The set of executions allowed by the enriched operational semantics is a subset of the set of executions allowed by the erased operational semantics.*

Proof. Each rule in the enriched operational semantics directly corresponds to a rule in the erased operational semantics that has precisely a subset of its premises. Since the enriched state never affects the erased state (except for perhaps making the machine get stuck more often), any execution (sequence of operational steps) in the enriched semantics corresponds directly to an execution in the erased semantics. □

Appendix B

Lemma 4 (Soundness) *All sequents proven using the rules of the slicing framework are true.*

Proof. Rule [P–UNSAT] is a syntactic conversion of a unsatisfiability obligation into an implication obligation. Rule [P–ENTAIL] is an instance of conjunction introduction rule of the sequent calculus [33]. Thus, every proof of the slicing framework is a proof of the sequent calculus, and consequently, the slicing framework rules are sound. □

Lemma 5 [SPLIT] *with* [AS–CORRELATION] *is confluent.*

Proof. Firstly, due to the set intersection operator being symmetric, the [AS–CORRELATION] relation is symmetric as well. Secondly, note that the [SPLIT] rule considers every constraint in the initial constraint set. The only possibility for the outcomes to be different is if the order is important. However due to the symmetry of the [AS–CORRELATION] and the fact that $P_1 \cup P_2$ covers all the elements in the partially constructed slicing R, the partitioning ensures that all previously considered constraints that are in the [AS–CORRELATION] relation with the current constraint will be part of the same slice. □

Lemma 6 (Relative completeness) *Let $P' \vdash Q$ be the sequent obtained by*

applying the complete slicing rules to the sequent $P \vdash Q$, where Q is atomic. Let LK^T be a sequent calculus obtained from LK by augmenting it with rules from a theory T that can handle the interpreted symbols of our formulas. If $P \vdash Q$ is provable, and P is satisfiable in LK^T, then $P' \vdash Q$, is also provable in LK^T.

Proof. The slicing mechanism will first convert P into the conjunction $P' \wedge P''$, where $\mathcal{V}(P'') \cap \mathcal{V}(Q) = \{\}$. It can then be decided that P'' can be discarded, and $P' \vdash Q$ is retained as a viable proof obligation. At this point, we have to make use of the statement that a sequent $R_1 \wedge R_2 \vdash R$ can be reduced to $R_1 \vdash R$ if $\mathcal{V}(R_2) \cap \mathcal{V}(R) = \{\}$, and $R_1 \wedge R_2$ is satisfiable. This statement can be proved by structural induction on the proof tree of $R_1 \wedge R_2 \vdash R$. Based on this statement, repeated eliminations of irrelevant hypotheses would not change the LK^T provability of $P' \vdash Q$, which establishes the original claim. □