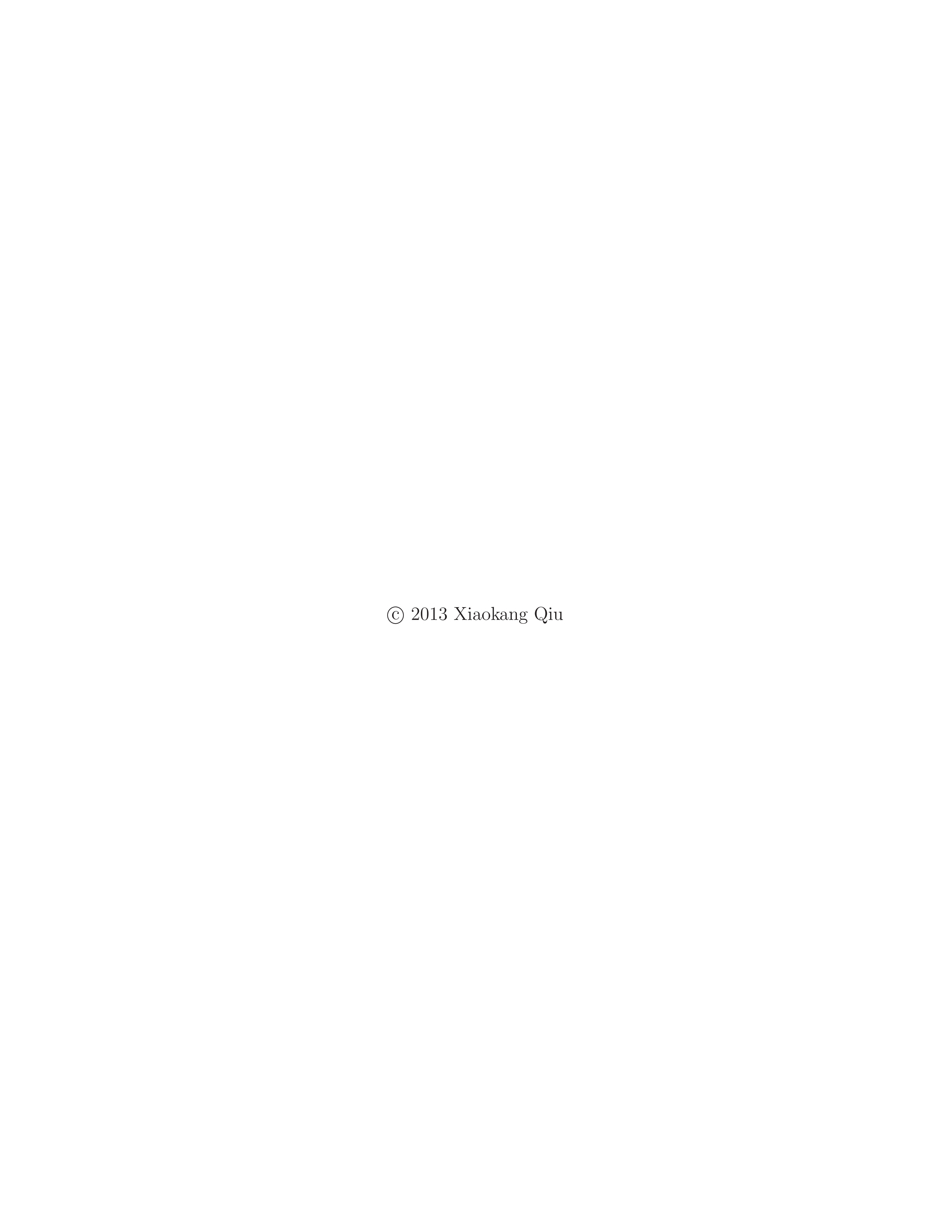


© 2013 Xiaokang Qiu

# AUTOMATIC TECHNIQUES FOR PROVING CORRECTNESS OF HEAP-MANIPULATING PROGRAMS

BY

XIAOKANG QIU

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2013

Urbana, Illinois

Doctoral Committee:

Associate Professor Madhusudan Parthasarathy, Chair
Zisman Family Professor Rajeev Alur, University of Pennsylvania
Associate Professor Grigore Roşu
Associate Professor Mahesh Viswanathan

# ABSTRACT

Reliability is critical for system software, such as OS kernels, mobile browsers, embedded systems and cloud systems. The correctness of these programs, especially for security, is highly desirable, as they should provide a trustworthy platform for higher-level applications and the end-users. Unfortunately, due to its inherent complexity, the verification process of these programs is typically manual/semi-automatic, tedious, and painful. Automating the reasoning behind these verification tasks and decreasing the dependence on manual help is one of the greatest challenges in software verification.

This dissertation presents two logic-based automatic software verification systems, namely STRAND and DRYAD, that help in the task of verification of heap-manipulating programs, which is one of the most complex aspects of modern software that eludes automatic verification. STRAND is a logic that combines an expressive heap-logic with an arbitrary data-logic and admits several powerful decidable fragments. The general decision procedures can be used in not only proving programs correct but also in software analysis and testing. DRYAD is a family of logics, including $\text{DRYAD}_{\text{tree}}$ as a first-order logic for trees and $\text{DRYAD}_{\text{sep}}$ as a dialect of separation logic. Both the two logics are amenable to automated reasoning using the *natural proof* strategy, a radically new approach to software verification. DRYAD and the natural proof techniques are so far the most efficient logic-based approach that can verify the full correctness of a wide variety of challenging programs, including a large number of programs from various open-source libraries. They hold promise of hatching the next-generation automatic verification techniques.

*To Ping.*

# ACKNOWLEDGMENTS

The six-year scholarly pursuit of a Ph.D. in Illinois has been the greatest challenge in my life. It is my privilege to thank all the people that have influenced this endeavor.

I am deeply indebted to my advisor, Madhusudan Parthasarathy, who has been a consistent source of research guidance and financial support. As a venerable mentor, he has taught me virtually all I know of being a good researcher. I am extremely admired for his incredible enthusiasm, deep thinking, broad knowledge, and sense of responsibility.

I thank other members of my doctoral committee, Rajeev Alur, Grigore Roşu and Mahesh Viswanathan. They have all eagerly provided me with advice on both my research and career. My research collaborator Gennaro Parlato has been an "older brother" of mine, and taught me important and practical lessons on conducting research, writing papers, and surviving the Ph.D. I also thank my fellow students, including Pranav Garg, Edgar Pek, Shambwaditya Saha, Francesco Sorrentino and Andrei Ştefănescu. They have been a source for great conversation and feedback on my research.

I am grateful for the great staff in the Department of Computer Science. Elaine Wilson handled my travel arrangements and reimbursements with extraordinary efficiency. Shirley Finke and Jennifer Dittmar made sure I got paid every semester. Rhonda McElroy, Mary Beth Kelly and Kara MacGregor helped me get visas, write and sign important supporting letters, and meet the coursework requirements on time.

I owe much thanks to my parents, Jianxiang Qiu and Minguang Shen. They have unceasingly supported and encouraged me for coming to America and being an academic. I am also very thankful to Elder Wei-Laung Hu and brothers and sisters in the Cornerstone Fellowship, for their prayer, sharing and encouragement in my spiritual journey in the Midwest.

Finally, I especially thank my wife, Ping. I would not have finished the degree without her love, support and patience. She has been with me, through thick and thin. For all those times, this dissertation is lovingly dedicated to her.

# TABLE OF CONTENTS

LIST OF TABLES . . . viii

LIST OF FIGURES . . . ix

LIST OF ABBREVIATIONS . . . x

CHAPTER 1 INTRODUCTION . . . 1
- 1.1 Summary of Contributions . . . 3
- 1.2 Organization . . . 4

CHAPTER 2 PRELIMINARIES . . . 6
- 2.1 Satisfiability Modulo Theories and Z3 . . . 6
- 2.2 Monadic Second-Order Theory and MONA . . . 7
- 2.3 Separation Logic . . . 8

CHAPTER 3 STRAND LOGIC . . . 11
- 3.1 Overview . . . 12
- 3.2 Motivating Examples and Logic Design . . . 13
- 3.3 Recursively Defined Data-Structures . . . 17
- 3.4 The Logic . . . 23
- 3.5 Program Verification Using STRAND . . . 27

CHAPTER 4 DECISION PROCEDURES . . . 39
- 4.1 Overview . . . 39
- 4.2 Satisfiability-Preserving Embeddings . . . 43
- 4.3 A Semantically Defined Fragment . . . 48
- 4.4 A Syntactically Defined Fragment . . . 54
- 4.5 Experimental Evaluation . . . 64
- 4.6 Related Work . . . 69

CHAPTER 5 NATURAL PROOFS . . . 71
- 5.1 The $\text{DRYAD}_{\text{tree}}$ Logic . . . 75
- 5.2 Deriving the Verification Condition . . . 82
- 5.3 A Decidable Fragment of $\text{DRYAD}_{\text{tree}}$ . . . 100
- 5.4 Formula Abstraction . . . 109

5.5 Experiments . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 115
5.6 Related Work . . . . . . . . . . . . . . . . . . . . . . . . . . . . 119

CHAPTER 6 COMBINING SEPARATION AND RECURSION . . . 123
6.1 Logic Design . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 124
6.2 Syntax . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 127
6.3 Semantics . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 129
6.4 Examples . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 137
6.5 Translating to A Logic over the Global Heap . . . . . . . . . . . 139

CHAPTER 7 NATURAL PROOFS FOR STRUCTURE, DATA, AND SEPARATION . . . . . . . . . . . . . . . . . . . . . . . . . . . . 147
7.1 Programs and Hoare-triples . . . . . . . . . . . . . . . . . . . . . 149
7.2 Generating the Verification Condition . . . . . . . . . . . . . . . 151
7.3 Unfolding Across the Footprint . . . . . . . . . . . . . . . . . . . 158
7.4 Formula Abstraction . . . . . . . . . . . . . . . . . . . . . . . . . 162
7.5 Case Study . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 166
7.6 Experimental Evaluation . . . . . . . . . . . . . . . . . . . . . . . 171
7.7 Related Work . . . . . . . . . . . . . . . . . . . . . . . . . . . . 177
7.8 Annotation Synthesis . . . . . . . . . . . . . . . . . . . . . . . . 179

CHAPTER 8 CONCLUSIONS . . . . . . . . . . . . . . . . . . . . . . . 191
8.1 Conclusions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 191
8.2 A Look Ahead . . . . . . . . . . . . . . . . . . . . . . . . . . . . 192

REFERENCES . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 194

# LIST OF TABLES

4.1 Results of program verification using STRAND . . . . . . . . . . 68

5.1 Results of program verification using $\text{DRYAD}_{\text{tree}}$ . . . . . . . . . 120

6.1 Domain-exact property and Scope function . . . . . . . . . . . . 141

7.1 Results of verifying data-structure algorithms . . . . . . . . . . 173

7.2 Results of verifying open-source libraries . . . . . . . . . . . . . 175

# LIST OF FIGURES


3.1 A list with head and tail . . . . . . . . . . . . . . . . . . . . . . . 14
3.2 $l'$ inherits data values from $l$ . . . . . . . . . . . . . . . . . . . . . . 15
3.3 A binary tree example represented in $\mathcal{R}_{bt}$ . . . . . . . . . . . . . 20
3.4 Definition of the $tailor_X$ function . . . . . . . . . . . . . . . . . . 22
3.5 Syntax of STRAND . . . . . . . . . . . . . . . . . . . . . . . . . . 24
3.6 Predicates defined for non-updating statements. . . . . . . . . . 33
3.7 Predicates defined for updating statements. . . . . . . . . . . . . 34
3.8 A syntactic transformation for conditions. . . . . . . . . . . . . 35

4.1 Definition of the *interpret* function . . . . . . . . . . . . . . . . . 49
4.2 Definition of the *tailor* function . . . . . . . . . . . . . . . . . . . 51
4.3 Syntax of STRAND$^{syn}_{dec}$ . . . . . . . . . . . . . . . . . . . . . . . . 55
4.4 A valid subset $X$ that falsifies $\beta$ . . . . . . . . . . . . . . . . . . . 58

5.1 Syntax of DRYAD$_{\text{tree}}$ . . . . . . . . . . . . . . . . . . . . . . . . . 77
5.2 Recursive definitions for red black trees . . . . . . . . . . . . . . 80
5.3 AVL-find routine . . . . . . . . . . . . . . . . . . . . . . . . . . . 100
5.4 Pre/post conditions and recursive definition for AVL-find . . . 100
5.5 Expanding the symbolic heap and generating the formulas . . 101
5.6 Syntax of local formulas $\varphi_p(x)$ . . . . . . . . . . . . . . . . . . . 103
5.7 Syntax of DRYAD$^{\text{dec}}_{\text{tree}}$ . . . . . . . . . . . . . . . . . . . . . . . 104
5.8 Inductive definition of $map(\varphi)$ . . . . . . . . . . . . . . . . . . . . 106

6.1 Syntax of DRYAD$_{\text{sep}}$ . . . . . . . . . . . . . . . . . . . . . . . . 128
6.2 The *pure* predicate for terms/formulas . . . . . . . . . . . . . . 131
6.3 Translation of DRYAD$_{\text{sep}}$ terms . . . . . . . . . . . . . . . . . . 142
6.4 Translation of DRYAD$_{\text{sep}}$ formulas . . . . . . . . . . . . . . . . 142

7.1 Syntax of programs . . . . . . . . . . . . . . . . . . . . . . . . . 149
7.2 Case study: Heapify . . . . . . . . . . . . . . . . . . . . . . . . . 166

# LIST OF ABBREVIATIONS

| | |
|---|---|
| APF | Array Property Fragment |
| BST | Binary Search Tree |
| FOL | First-Order Logic |
| ITE | If-Then-Else |
| LCA | Least Common Ancestor |
| MSO | Monadic Second-Order |
| MSOL | Monadic Second-Order Logic |
| RDDS | Recursively Defined Data-Structure |
| SL | Separation Logic |
| SMT | Satisfiability Modulo Theories |
| SPE | Satisfiability-Preserving Embedding |
| VC | Verification Condition |
| WS1S | Weak monadic Second-order theory of 1 Successor |
| WS2S | Weak monadic Second-order theory of 2 Successors |

# CHAPTER 1

# INTRODUCTION

As software systems have become indispensable in our daily lives, their reliability has grown to be one of the most concerned issues, especially when deployed for critical applications and services. This includes complex embedded software in avionics, vehicles and medical equipments, system software such as operating systems and browsers, as well as today's emerging cloud software.

Numerous approaches have been proposed to build reliability-critical software to satisfy its complex correctness requirements. A focus of intense research in this area has been program verification using theorem provers, utilizing manually provided proof annotations, such as pre- and post-conditions for functions and loop invariants. Automatic theory solvers (e.g. SMT solvers) that handle a variety of quantifier-free theories including arithmetic, uninterpreted functions, Boolean logic, etc., serve as effective tools that automatically discharge the validity checking of many verification conditions [18].

A key area that has eluded the above paradigm of specification and verification is heap analysis: the verification of programs that dynamically allocate memory and manipulate them using pointers, maintaining structural invariants (e.g. "the nodes form a tree"), aliasing invariants, and invariants on the data stored in the locations (e.g. "the keys of a list are sorted"). Heap-manipulating programs are pervasive in lower-level computer systems: garbage collectors, OS kernels, device drivers, mobile browsers, etc. The functional correctness of these programs is highly desirable, as they should provide a trustworthy platform for higher-level applications. Unfortunately, due to its inherent complexity, the verification process of these programs is typically manual/semi-automatic, tedious and painful. It usually eludes all existing automatic techniques and tools, and poses one of the greatest challenges in software verification.

Dynamically allocated heaps are difficult to reason with for several reasons. First, the specification of proof annotations itself is hard, as the annotation needs to talk about several intricate properties of an unbounded heap, often requiring quantification and reachability predicates, and needs to specify aliasing as well as structural properties of the heap. Also, in experiences with manual verification, it has been observed that pre- and post-conditions get unduly complex, including large formulas that say how the frame of the heap that is *not* touched by the program remains the same across a function. expressing such properties naturally and succinctly in a logical formalism has been challenging, and reasoning with them automatically even more so.

To this end, this dissertation aims at building automatic software verification systems, with a focus on heap-manipulating programs. An underlying theme of this work is that for logical reasoning to succeed, two important directions must be pursued:

1. Consider more expressive logics to allow the programmer to easily specify complex data structures; and

2. Develop decision procedures that can reason efficiently about these sophisticated logics.

The two directions cannot be considered in isolation. On the one hand, when the logic becomes more powerful, e.g., separation logic with inductive algebraic definitions, the analysis is usually manual or semi-automatic, the latter being usually sound, incomplete, and non-terminating, and proceeds by heuristically searching for proofs using a proof system, unrolling recursive definitions arbitrarily. Typically, such tools can find simple proofs if they exist, but are unpredictable, and cannot robustly produce counterexamples. On the other hand, when the verification becomes completely automatic, e.g., Lisbq [46] and CSL [15], the logics are often constrained heavily on expressivity. They are often not sufficiently expressive to state complex properties of the heap (e.g. the balancedness of an AVL tree, or that the set of keys stored in a heap does not change across a program).

The main goal of this work is to develop new program logics and methodologies that strike a nice balance between *expressiveness* and *verifiability* in the area of verifying heap-manipulating programs. In particular, we present two logics, one called Strand, and the other one called Dryad. Strand is

a logic that combines an expressive heap-logic with an arbitrary data-logic and admits several powerful decidable fragments. It is one of the most powerful decidable logics for complex properties combining heap structures and data. DRYAD is a family of logics that support recursive definitions and our novel proof strategy called *Natural Proofs*. Tools are built based on our logical mechanism and successfully verified the full partial correctness of a wide variety of challenging programs, including a large number of programs from various open source libraries. We explain our contributions in details as follows.

## 1.1 Summary of Contributions

The main contributions of this dissertation are highlighted as follows:

1. The STRAND logic that expresses constraints involving heap structures and the data they contain; this logic allows quantification in limited form and is capable of expressing complex properties of common data structures.

2. Two decidable fragments of STRAND, one semantically defined ($\text{STRAND}_{dec}^{sem}$) and one syntactically defined ($\text{STRAND}_{dec}^{syn}$); experiments show that complex verification conditions generated from heap-manipulating programs can be handled efficiently using the decision procedures for STRAND.

3. A novel proof strategy that calls *Natural Proofs*. Natural proofs are a subclass of proofs that are amenable to completely automated reasoning; it is a systematic methodology that provides sound but incomplete procedures, and that captures common reasoning tactics in program verification.

4. Two variants of the DRYAD logic that are amenable to natural proofs: $\text{DRYAD}_{\text{tree}}$ extends first-order logic to support recursive definitions for trees; $\text{DRYAD}_{\text{sep}}$ is a dialect of Separation Logic that disallows explicit quantification but permits powerful recursive definitions. The salient feature of $\text{DRYAD}_{\text{sep}}$ is that it admits a *quantifier-free, deterministic*

translation to a classical logic with free set variables, which can be handled using modern SMT solvers.

5. Exploit the Natural Proofs strategy to verify a wide variety of open-source programs from real world; these programs include low-level C routines from Glib, OpenBST, Linux kernel and the ExpressOS project.

6. Develop a preliminary annotation synthesizer based on natural proofs; the natural proof strategy is encoded into ghost annotations which tend to help semi-automatic verifiers (such as VCC) find a proof automatically. The reduced annotation burden can significantly benefit programmers without specialist proving knowledge.

The STRAND logic and its two decidable fragments were first defined in [50]. Furthermore, A dedicated, more efficient decision procedure was obtained in [51]. Our proof methodology of natural proofs was first proposed in [52], in the context of the logic $\text{DRYAD}_{\text{tree}}$ which is only for tree data-structures. In [64], aiming at providing natural proofs for general properties of structure, data, and separation, the $\text{DRYAD}_{\text{sep}}$ logic was proposed as a dialect of separation logic. In this paper, we developed natural proofs for $\text{DRYAD}_{\text{sep}}$ and reason with heaplet using classical logic over the theory of sets.

## 1.2 Organization

This rest of the dissertation is organized into chapters as follows:

**Chapter 2** presents a technical background for the rest of the dissertation.

**Chapter 3** defines the STRAND logic and shows how to derive verification conditions from STRAND-annotated programs.

**Chapter 4** gives decidability proofs for the two decidable fragments, and experimentally evaluates the effectiveness of the decision procedures.

**Chapter 5** illustrates the procedures for reasoning with heap-manipulating programs using natural proofs, particularly based on the $\text{DRYAD}_{\text{tree}}$ logic.

**Chapter 6** presents the DRYAD$_{\text{sep}}$ logic and shows its conversion to a classical logic using the theory of sets.

**Chapter 7** develops natural proofs for DRYAD$_{\text{sep}}$, and applies the strategy to the verification of various real world programs and libraries.

**Chapter 8** presents the conclusions and looks ahead to the future research directions.

# CHAPTER 2

# PRELIMINARIES

This chapter discusses technical preliminaries required for the rest of the dissertation.

## 2.1 Satisfiability Modulo Theories and Z3

A fundamental component of analysis techniques for complex programs is logical reasoning. The advent of efficient SMT solvers (satisfiability modulo theory solvers) have significantly advanced the techniques for the analysis of programs.

Though it is well known that the satisfiability of First-Order Logic (FOL) is undecidable, when the models are constrained by some background theories, the satisfiability could be decidable. In past decades, efficient decision procedures emerged for many logical theories and fragments (e.g. integers, arrays, theory of uninterpreted functions, etc.) that are typically useful in computer science [18]. These theories have been standard practice in program verification.

Moreover, by using techniques that *combine* theories, larger decidable theories can be obtained. The Nelson-Oppen framework [60] and the Shostak approach [70] allow generic combinations of *quantifier-free* theories, and has been used in efficient implementations of combinations of theories using a SAT solver that queries decision procedures of background theories.

SMT solvers advance several analysis techniques. They are useful in test-input generation, where the solver is asked whether there exists an input to a program that will drive it along a particular path; see for example [37]. SMT solvers are also useful in static-analysis based on abstract interpretation, where the solver is asked to compute precise abstract transitions, i.e. asked whether there is a concretization of an abstract state $a$ that transitions

to a concretization of another abstract state $a'$ (for example see SLAM [4] for predicate abstraction and TVLA [48, 80] for shape-analysis). Solvers are also useful in classical deductive verification, where Hoare-triples that state pre-conditions and post-conditions can be transformed into verification conditions whose validity is checked by the solver; for example BOOGIE [5] and ESC/Java [35] use SMT solvers to prove verification conditions.

Thank to the technical breakthrough and competition [61, 7] in recent years, many efficient, off-the-shelf SMT solvers [6, 25, 32, 28] are available, especially for verification. In this dissertation, we use Z3 [28] as the back-end theorem prover. Z3 is a high-performance, state-of-the-art SMT solver developed at Microsoft Research, and has been used in several program analysis, verification and test-case generation projects at Microsoft. It integrates an efficient SAT solving core with decision procedures for component theories such as bit-vectors, arithmetic, arrays, partial orders and tuples. Z3 also adopts several approaches to handle quantifiers, including model-based quantifier instantiation [36] and E-matching [29]. However, in general, the process of theorem proving is no longer a decision procedure for quantified formulas.

## 2.2 Monadic Second-Order Theory and MONA

Another set of well-known decidability results stems from Monadic Second-Order Logic (MSOL), and is related to regular languages and automata theory. Second-order logic extends first-order logic with quantifiers over relations, and MSOL is just a restriction of second-order logic where all the relational variables are unary. MSOL is a powerful logic and is capable of expressing complex properties such as ”the graph is 4-colorable” or ”the graph is connected”. In 1960s, Büchi and Elgot created the famous connection between MSOL and finite automata, which can be stated as:

**Theorem 2.2.1** ([20, 33])**.** *A language of finite words is recognizable by a finite automaton iff it is definable in MSO*[1]*.*

The intuition behind the translation from automata to MSO is that, the computations of the automaton can be captured by sets. Desired formula

[1]The signature consists of unary relations and the successor relation.

needs to say that "there is an encoding of a sequence of states that forms an accepting computation". Reversely, from a MSO formula, an automaton can be constructed by structural induction on the formula. The same idea can also be applied to finite tree automata:

**Theorem 2.2.2** ([74, 31])**.** *A set of finite trees is recognizable by a finite tree automaton iff it is MSO-definable.*

As a variation of MSO over finite words, WS1S (Weak monadic Second-order theory of 1 Successor) is the set of formulas true in the structure $(\mathbb{N}, S^2)$ under the restriction that all set quantifiers range over *finite* sets. Since a finite set of natural numbers can always be encoded as a finite string of $\{0, 1\}$, the decidability of WS1S immediately follows Theorem 2.2.1. WS2S (Weak monadic Second-order theory of 2 Successors) is a generalization of WS1S which is interpreted over infinite binary trees, and similar decidability can be obtained by Theorem 2.2.2.

**Corollary 2.2.3.** *The satisfiability of WS1S/WS2S is decidable.*

Though the complexity of deciding WS1S/WS2S is non-elementary [56], efficient implementations such as Mona are available. Mona encodes WS1S and WS2S formulas as minimum DFAs (Deterministic Finite Automata) and GTAs (Guided Tree Automata), which are represented by shared, multi-terminal BDDs (Binary Decision Diagrams). These techniques make Mona practically tractable and useful. Its most famous application has been the Pale program verifier [58].

## 2.3 Separation Logic

In recent years, *Separation Logic* (SL), especially in combination with recursive definitions, has emerged as a succinct and natural logic to express properties about structure and separation [68, 63].

The primary design principle behind separation logic is the decision to express *strict specifications*— logical formulas must naturally refer to *heaplets* (subparts of the heap), and, by default, the smallest heaplets over which the formula needs to refer to. This is in contrast to classical logics (such as

[2] $S(x, y)$ is true if $y = x + 1$.

First-Order Logic) which implicitly refer to the entire heap globally. Strict specifications permit elegant ways to capture how a particular sub-part of the heap changes due to a procedure, implicitly leaving the rest of the heap and its properties unchanged across a call to this procedure. Separation logic is a particular framework for strict specifications, where formulas are implicitly defined on strictly defined heaplets, and where heaplets can be combined using a novel *spatial conjunction operator* denoted by $*$.

Separation logic is interpreted over programs states consisting of a *store* and a *heaplet*. A store $s$ is a function mapping variables to values, which could be data values or memory addresses; a heaplet $h$ is a *partial* function mapping memory addresses to values. Assertions in SL can be constructed from the following constructs:

- `emp` asserts that the heap is empty, i.e., $(s, h) \models \mathtt{e}mp$ if $\mathrm{Dom}(h) = \emptyset$;
- $t \mapsto t'$ relates an address $t$ and a value $t'$, asserting that $t$ maps to $t'$, and the heaplet is defined only on the location of $t$; formally, $(s, h) \models t \mapsto t'$ if $h(t) = t'$ and $\mathrm{Dom}(h) = \{t\}$;
- $P * Q$ asserts that the heaplet can be split into two disjoint parts such that one satisfies $P$ and the other satisfies $Q$, i.e., $(s, h) \models P * Q$ if there exist $h_1, h_2$ such that $h_1 \perp h_2$ and $(s, h_1) \models P$ and $(s, h_2) \models Q$;
- $P - *Q$ asserts that extending the heaplet with a disjoint part that satisfies $P$ results in a heaplet that satisfies $Q$, i.e., $(s, h) \models P - *Q$ if for any $h'$ such that $h \perp h'$ and $(s, h') \models P$, $(s, h \cup h') \models Q$.

In addition, standard Boolean connectives are also allowed.

The Hoare-triples for SL also has the tight semantics. Given a program $C$ and two SL assertions $P$ and $Q$, the Hoare-triple $\{P\}\ C\ \{Q\}$ (as a partial specification) asserts that if the initial state satisfies $P$, and the program does not go wrong and terminates, then the final state will satisfy $Q$. Notice that $P$ and $Q$ describe only a portion of the heap, and the program can only access the locations that are asserted in the precondition and the locations that are newly allocated.

The *frame rule* in SL captures the main advantage of strict specifications:

$$\frac{\{P\}\ C\ \{Q\}}{\{P * R\}\ C\ \{Q * R\}} \quad \mathrm{Mod}(C) \cap \mathrm{fv}(R) = \emptyset$$

It says that if the Hoare-triple $\{P\}\ C\ \{Q\}$ holds for some program $C$, then $\{P{*}R\}\ C\ \{Q{*}R\}$ also holds (with side-conditions that the modified variables in $C$ are disjoint from the free variables in $R$). The frame rule elegantly enables the local reasoning, which allows one to derive a global fact of a program from a local fact of the program. Local reasoning is an important component of reasoning with heap-manipulating programs, but in classical logic, the frame rule is not straightforward, as it is difficult to ensure the conjoined property does not involve the portion of the program state that might be modified by the program.

Yang [76] showed that in general, SL is even not recursively enumerable. However, the quantifiers are the main source of the undecidability, i.e., SL becomes decidable when quantifiers are prohibited [21]. A small decidable fragment of SL is given in [9] for lists with both points-to and reachability relations. This fragment has been further extended in [17] to include a restricted form of arithmetic.

# CHAPTER 3

# STRAND LOGIC

Several approaches to program analysis, like deductive verification, generating tests using constraint solving, abstraction, etc. have greatly benefited from the engineering of efficient SMT solvers, which currently provide automated decision procedures for a variety of quantifier-free theories, including integers, bit-vectors, arrays, uninterpreted functions, as well as *combinations* of these theories using the Nelson-Oppen method [60]. One of the most important kinds of reasoning in program verification that has evaded tractable decidable fragments is reasoning with dynamic heaps and the data contained in them.

Reasoning with heaps and data seems to call for decidable combinations of logics on *graphs* that model the heap structure (with heap nodes modeled as vertices, and field pointers as edges) with a logic on the data contained in them (like the quantifier-free theory of integers already supported by current SMT solvers). The primary challenge in building such decidable combinations stems from the *unbounded* number of nodes in the heap structures. This mandates the need for universal quantification in any reasonable logic in order to be able to refer to all the elements of the heap (e.g. to say a list is sorted, we need some form of universal quantification). However, the presence of quantifiers immediately annuls the use of Nelson-Oppen combinations, and requires a new theory for combining unbounded graph theories with data.

There have been a few breakthroughs in combining heap structures and data recently. For instance, HAVOC [46] supports a logic that ensures decidability using a very highly restrictive syntax, and CSL [15] extends the HAVOC logic mechanism to handle constraints on sizes of structures. However, both these logics have very awkward syntaxes, that involve the domain being partially ordered with respect to *sorts*, and the logics are heavily curtailed so that the decision procedure can move down the sorted structures

hierarchically and hence terminate. Moreover, these logics cannot express even simple properties on trees of unbounded depth, like the property that a tree is a binary search tree. More importantly, the technique for deciding the logic is encoded in the *syntax*, which in turn narrowly aims for a fast reduction to the underlying data-logic, making it hard to extend or generalize.

## 3.1 Overview

In this chapter, we propose a new fundamental technique for deciding theories that combine heap structures and data, for fragments of a logic called STRAND. The technique is based on defining a notion of *satisfiability-preserving embeddings* between heap-structures, and extracting the minimal models with respect to these embeddings to synthesize a data-logic formula, which can then be decided by an SMT solver for the data-theory.

We define a new logic called STRAND (for "STRucture ANd Data"), that combines a powerful heap-logic with an arbitrary data-logic. STRAND formulas are interpreted over a class of *data-structures* $\mathcal{R}$, and are of the form $\exists\vec{x}\forall\vec{y}\varphi(\vec{x},\vec{y})$, where $\varphi$ is a formula that combines a complete monadic second-order logic over the heap-structure (and can have additional quantification), and a data-logic that can constrain the data-fields of the nodes referred to by $\vec{x}$ and $\vec{y}$.

The heap-logic in STRAND is derived from the rich logic tradition of designing decidable monadic second-order logics over graphs, and is extremely expressive in defining structural shapes and invariants. STRAND formulas are interpreted over a recursively defined class of data-structures $\mathcal{R}$, which is defined using a regular set of *skeleton* trees with MSO-defined edge-relations (pointer-relations) between them. This way of recursively defining data-structures is not new, and was pioneered by the PALE system [58], which reasons with purely structural properties of heaps defined in a similar manner. In fact, the notion of *graph types* [43] is a convenient and simple way to define data-structure types and invariants, and is easily expressible in our scheme. Data-structures defined over skeleton trees have enough expressive power to state most data-structure invariants of recursively defined data-structures, including nested lists, threaded trees, cyclic and doubly-linked lists, and separate or loosely connected combinations of these structures.

Moreover, they present a class of graphs that have a decidable MSO theory, as MSO on these graphs can be *interpreted* using MSO over trees, which is decidable. In fact, graphs defined this way are one of the largest classes of graphs that have a decidable MSO theory.

We show in this chapter, the STRAND logic is well-suited to reasoning with programs. In particular, assume we are given a (straight-line) program $P$, a pre-condition on the data-structure expressed as a set of recursive structures $\mathcal{R}$, and a pre-condition and a post-condition expressed in a sub-fragment of STRAND that allows Boolean combinations of the existential and universal fragments. We show that checking the invalidity of the associated Hoare-triple reduces to the satisfiability problem of STRAND over a new class of recursive structures $\mathcal{R}_P$. This facilitates using STRAND to express a variety of problems, including the applications of test-input generation, finding abstract transitions, and deductive verification mentioned above.

Note that despite its relative expressiveness in allowing quantification over nodes, STRAND formulas cannot express certain constraints such as those that constrain the *length* of a list of nodes (e.g., to express that the number of black nodes on all paths in a red-black tree are the same), nor express the *multi-set* of data-values stored in a data-structure (e.g., to express that one list's data contents are the same as that of another list). We hope that future work will extend the results in this paper to handle such constraints.

**Organization:** We first present the intuitions behind the logic design of STRAND in Section 3.2. We define recursively defined data-structures (RDDS) in Section 3.3, then introduce the STRAND logic with examples in Section 3.4. In Section 3.5, we present the VC-generation process with respect to STRAND.

## 3.2 Motivating Examples and Logic Design

The goal of this section is to present an overview of the issues involved in finding decidable logics that combine heap structure and data, which sets the stage for designing the the logic STRAND that is potentially decidable, and motivates the choices in our logic design using simple examples on lists.

Let us consider lists in this section, where each node $u$ has a data-field $d(u)$ that can hold a value (say an integer), and with two variables `head` and `tail` pointing to the first and last nodes of the list, respectively (see Figure 3.1). Consider first-order logic, where we are allowed to quantify over the nodes of the list, and further, for any node $x$, allowed to refer to the *data-field* of $x$ using the term $d(x)$. Let $x \rightarrow y$ denote that $y$ is the successor of $x$ in the list, and let $x \rightarrow^* y$ denote that $x$ is the same as $y$ or precedes $y$ in the list.

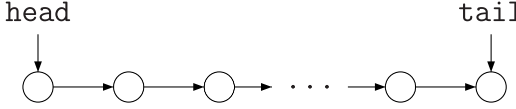



Figure 3.1: A list with head and tail

Consider a formula expressing the sortedness of lists:

**Example 3.2.1** (Sorted list).

$$\varphi_1: \quad d(\texttt{head}) = c_1 \quad \wedge \quad d(\texttt{tail}) = c_2 \quad \wedge \\ \forall y_1 \forall y_2 ((y_1 \rightarrow^* y_2) \Rightarrow d(y_1) \leq d(y_2))$$

The above says that the list must be *sorted* and that the head of the list must have value $c_1$ and the tail must have value $c_2$. Note that the formula is satisfiable iff $c_1 \leq c_2$, and in which case it is actually satisfied by a list containing *just two elements,* pointed to by `head` and `tail`, with values $c_1$ and $c_2$, respectively.

In fact, the property that the formula is satisfiable by a two-element list has *nothing* really to do with the data-constraints involved in the above formula. Assume that we have no idea as to what the data-constraints mean, and hence look upon the above formula by replacing all the data-constraints using uninterpreted predicates $p_1, p_2, \ldots$ to get the formula:

$$\widehat{\varphi}_1: \quad p_1(d(\texttt{head})) \quad \wedge p_2(d(\texttt{tail})) \quad \wedge \\ \forall y_1 \forall y_2 ((y_1 \rightarrow^* y_2) \Rightarrow p_3(d(y_1), d(y_2)))$$

Now, we do not know whether the formula is satisfiable (for example, $p_1$ may be unsatisfiable). But we still do know that two-element lists are *always*

*sufficient.* In other words, *if there is a list that satisfies the above formula, then there is a two-element list that satisfies it.* The argument is simple: take any list $l$ that satisfies the formula, and form a new list $l'$ that has only the head and tail of the list $l$, with an edge from head to tail, and with data values inherited from $l$ (see Figure 3.2). It is easy to see that $l'$ satisfies the formula as well, since whenever two nodes are related by $\rightarrow^*$ in the list $l'$, the corresponding elements in $l$ are similarly related: The constraints $p_1(head)$ and $p_2(\texttt{tail})$ are obviously satisfied in $l'$. Moreover, for any possible valuations of $y_1$ and $y_2$ in $l'$, where $y_1 \rightarrow^* y_2$ in $l'$ holds, $y_1 \rightarrow^* y_2$ also holds in $l$, and hence the constraint $p_3(y_1, y_2)$ is satisfied.

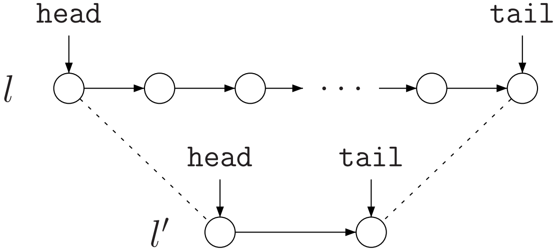



Figure 3.2: $l'$ inherits data values from $l$

In other words, given any arbitrarily large list satisfying the above formula, we can always find a two-element list satisfying the formula, independent of what the data-predicates may mean. This property, of course, does not hold on all formulas, as we see in the example below:

**Example 3.2.2** (A list counting from $c_1$ to $c_2$).

$$\varphi_2 : \quad d(\texttt{head}) = c_1 \quad \wedge \quad d(\texttt{tail}) = c_2 \quad \wedge$$
$$\forall y_1 \forall y_2 ((y_1 \rightarrow y_2) \Rightarrow d(y_2) = d(y_1) + 1)$$

The above says that the values in the list increase by one as we go one element down the list, and that the head and tail of the list have values $c_1$ and $c_2$, respectively. This formula is satisfiable iff $c_1 < c_2$. However, there is no bound on the size of the minimal model that is *independent* of the data-constraints. For example, if $c_1 = 1$ and $c_2 = 10^6$, then the smallest

list that satisfies the formula has a million nodes. In other words, the data-constraints place arbitrary lower bounds on the *size* of the minimal structure that satisfies the formula.

Intuitively, the formula $\varphi_2$ refers to *successive* elements in the list, and hence a large model that satisfies the formula is not necessarily contractible to a smaller model. The formula $\varphi_1$ in the sortedness example (Example 3.2.1) refers to pairs of elements that were reachable, leading to contraction of large models to small ones.

Above examples show the design principle of the decidable fragment of STRAND is to examine the structural constraints in a formula $\varphi$, and enumerate a *finite* set of structures such that the formula is satisfiable iff it one of these structures can be populated with values to satisfy the formula. This strategy necessarily fails for the above formula $\varphi_2$, as there is no class of finite structures that adequately captures all models of the formula, independent of the data-constraints. The *sortedness* formula $\varphi_1$ in the first example should be efficiently checked using the decision procedures for STRAND, while $\varphi_2$ should be not.

We further consider the relationship between quantifiers and decidability. Consider the following example:

**Example 3.2.3** (A counting but not sorted list)**.** *Consider the formula:*

$$\begin{aligned} \varphi_3: \quad & d(\texttt{head}) = c_1 \quad \wedge \quad d(\texttt{tail}) = c_2 \quad \wedge \\ & \forall y_1((y_1 \neq \texttt{tail}) \Rightarrow \exists y_2(d(y_2) = d(y_1) + 1)) \end{aligned}$$

This formula says that for any node $n$ except the tail, there is some node $n'$ that has the value $d(n)+1$. Notice that the formula is satisfiable if $c_1 < c_2$, but still there is no a priori bound on the minimal model that is independent of the data-constraints. In particular, if $c_1 = 0$ and $c_2 = 10^6$, then the smallest model is a list with $10^6$ nodes. Moreover, the reason why the bounded structure property fails is not because of the data-constraints referring to successive elements as in Example 2.2, but rather because the above formula has a $\forall\exists$ prefix quantification of data-variables. Formulas where an existential quantification follows a universal quantification in the prefix seldom have bounded models, and STRAND hence only allows formulas with $\exists^*\forall^*$ quantification prefixes. Note that quantification of *structure variables* (variables that quantify over nodes but whose data-field is not referenced in

the formula) can be arbitrary, and in fact we allow STRAND formulas to even have *set quantifications* over nodes.

## 3.3 Recursively Defined Data-Structures

In this section, we define recursively defined data-structures using a formalism that defines the nodes and edges using MSO formulas over a regular set of trees. Intuitively, a set of data-structures is defined by taking a regular class of trees that acts as a *skeleton* over which the data-structure will be defined. The precise set of nodes of the tree that corresponds to the nodes of the data-structure, and the edges between these nodes (which model pointer fields) will be captured using MSO formulas over these trees. We call such classes of data-structures *recursively defined data-structures* (RDDSs).

RDDS is very powerful mechanisms for defining invariants of data-structures. The notion of *graph types* [43] is a very similar notion, where again data-structure invariants are defined using a tree-backbone but where edges are defined using *regular path expressions*. Graph types can be modeled directly in our framework; in fact, our formalism is more powerful.

The framework of RDDS is also interesting because they define classes of graphs that have a *decidable MSO theory*. In other words, given a class $\mathcal{C}$ of recursively defined data-structures, the satisfiability problem for MSO formulas over $\mathcal{C}$ (i.e. the problem of checking, given $\varphi$, whether there is some structure $R \in \mathcal{C}$ that satisfies $\varphi$) is decidable. The decision procedure works by interpreting the MSO formula on the tree-backbone of the structures. In fact, our framework can capture all graphs definable using *edge-replacement grammars*, which are one of the most powerful classes of graphs known that have a decidable MSO theory [34].

**Remark:** We model heap structures as labeled directed graphs: the nodes of the graph correspond to heap locations, and an edge from $n$ to $n'$ labeled $f$ represents the fact that the field pointer $f$ of node $n$ points to $n'$. The nodes in addition have *labels* associated to them; labels are used to signify special nodes (like NIL nodes) as well as to denote the program's pointer variables that point to them.

### 3.3.1 Formal Definition

For any $k \in \mathbb{N}$, let $[k]$ denote the set $\{1, \ldots, k\}$. A $k$-ary tree is a set $V \subseteq [k]^*$, where $V$ is non-empty and prefix-closed. We call $u.i$ the $i$-th child of $u$, for every $u, u.i \in V$, where $u \in [k]^*$ and $i \in [k]$. Let us fix a countable set of first-order variables $FV$ (denoted by $s$, $t$, etc.), a countable set of set-variables $SV$ (denoted by $S$, $T$, etc.), and a countable set of Boolean-variables $BV$ (denoted by $p$, $q$, etc.). The syntax of the Monadic second-order (MSO) [75] formulas on $k$-ary trees is defined:

$$\delta ::= p \mid succ_i(s,t) \mid s = t \mid s \in S \mid \varphi \vee \varphi \mid \neg\varphi \mid \exists s.\varphi \mid \exists S.\varphi \mid \exists p.\varphi$$

where $i \in [k]$. The atomic formula $succ_i(s,t)$ holds iff $t$ is the $i$-th child of $s$. Other logical symbols are interpreted in the traditional way.

**Definition 3.3.1** (Recursively Defined Data-Structures). *A class of* recursively defined data-structures *(RDDS) over a graph signature* $\Sigma = (L_v, L_e)$ *(where* $L_v$ *and* $L_e$ *are finite sets of labels) is specified by a tuple* $\mathcal{R} = (\psi_{Tr}, \psi_U, \{\alpha_a\}_{a \in L_v}, \{\beta_b\}_{b \in L_e})$*, where* $\psi_{Tr}$ *is an MSO sentence,* $\psi_U$ *is a unary predicate defined in MSO, and each* $\alpha_a$ *(*$\beta_b$*) is a monadic (binary) predicate defined using MSO, respectively, where all MSO formulas are over k-ary trees, for some* $k \in \mathbb{N}$. □

Let $\mathcal{R} = (\psi_{Tr}, \psi_U, \{\alpha_a\}_{a \in L_v}, \{\beta_b\}_{b \in L_e})$ be an RDDS and $T$ be a $k$-ary $\Sigma$-labeled tree that satisfies $\psi_{Tr}$. Then $T = (V, \{E_i\}_{i \in [k]})$ defines a graph $Graph(T) = (N, E, \mu, \nu, L_v, L_e)$ as follows:

- $N = \{s \in V \mid \psi_U(s) \text{ holds in } T\}$
- $E = \{(s, s') \mid \psi_U(s) \text{ and } \psi_U(s') \text{ hold, and } \beta_b(s, s') \text{ holds in } T \text{ for some } b \in L_e\}$
- $\mu(s) = \{a \in L_v \mid \psi_U(s) \text{ holds and } \alpha_a(s) \text{ holds in } T\}$
- $\nu(s, s') = \{b \in L_e \mid \psi_U(s) \text{ and } \psi_U(s') \text{ hold and } \beta_b(s, s') \textit{ holds in } T\}$.

In the above, $N$ denotes the nodes of the graph, $E$ the set of edges, $\mu$ the labels on nodes, and $\nu$ the labels on edges. The class of graphs defined by $\mathcal{R}$ is the set $Graph(\mathcal{R}) = \{Graph(T) \mid T \models \psi_{Tr}\}$. These graphs are interpreted as heap structures.

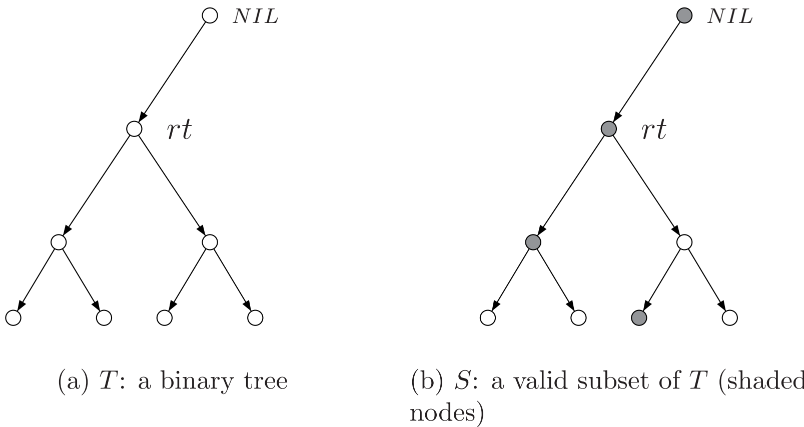
We give two examples of modeling heap structures as recursively defined data-structures below.

**Example 3.3.2** (Binary trees). *Binary trees are common data-structures, in which two field pointers, $l$ and $r$, point to the left and right children, respectively. If a node does not have a left (right) child, then the $l$ ($r$) field points to the unique NIL node in the heap. Moreover, there is a node $rt$ which is the root of the tree. Binary trees can be modeled as a recursively defined data-structure. For example, we can model the unique NIL node as the root of the tree, and model the actual nodes of the binary tree at the left subtree of the root (i.e. the tree under the left child of the root models $rt$). The right subtree of the root is empty. Binary trees can be modeled as $\mathcal{R}_{bt} = (\psi_{Tr}, \psi_U, \{\alpha_{rt}, \alpha_{nil}\}, \{\beta_l, \beta_r\})$ where*

$$\begin{aligned}
\psi_{Tr} &\equiv \exists y_1.\Big(root(y_1) \wedge \nexists y_2.\big(succ_r(y_1, y_2)\big)\Big) \\
\psi_U(x) &\equiv true \\
\alpha_{rt}(x) &\equiv \exists y.\big(root(y) \wedge succ_l(y, x)\big) \\
\alpha_{NIL}(x) &\equiv root(x) \\
\beta_l(x_1, x_2) &\equiv \exists y.\big(root(y) \wedge leftsubtree(y, x_1) \wedge succ_l(x_1, x_2)\big) \vee \\
&\quad \big(root(x_2) \wedge \nexists z.succ_l(x_1, z)\big) \\
\beta_r(x_1, x_2) &\equiv \exists y.\big(root(y) \wedge leftsubtree(y, x_1) \wedge succ_r(x_1, x_2)\big) \vee \\
&\quad \big(root(x_2) \wedge \nexists z.succ_r(x_1, z)\big)
\end{aligned}$$

*where the predicate $root(x)$ indicates whether $x$ is the root of the backbone tree, and the relation $leftsubtree(y, x)$ ($rightsubtree(y, x)$) indicates whether $x$ belongs to the subtree of the left (right) child of $y$. They can all be defined easily in MSO. As an example, Figure 3.3a shows a binary tree represented in $\mathcal{R}_{bt}$.*

**Example 3.3.3** (Two lists). *Consider modeling two disjoint lists, starting from $h_1$ and $h_2$, in the heap. They share the same field pointer $n$ and the same NIL node. This structure can be modeled as a recursively defined data-structure as follows. Intuitively, the two lists can be encoded as a binary tree. The tree simply consists of two lists. One list starts at the left child of the root and the second list at the right child of the root, and the $n$-pointer is modeled as the left-child relation. The root is labeled NIL. $\psi_{Tr}$ simply says there are no nodes other than these. Then the left child and the right child*




(a) $T$: a binary tree

(b) $S$: a valid subset of $T$ (shaded nodes)

Figure 3.3: A binary tree example represented in $\mathcal{R}_{bt}$

*of the root can be labeled $h_1$ and $h_2$, respectively. The above predicates and relations can all be described in MSOL:*

$$
\begin{aligned}
\psi_{Tr} &\equiv \exists y_1 y_2. \Big( root(y_1) \wedge succ_r(y_1, y_2) \wedge \\
&\qquad\qquad \forall y_3. \big( leftreach(y_1, y_3) \vee leftreach(y_2, y_3) \big) \Big) \\
\psi_U(x) &\equiv true \\
\alpha_{head1}(x) &\equiv root(x) \\
\alpha_{head2}(x) &\equiv \exists y. \big( root(y) \wedge succ_r(y, x) \big) \\
\alpha_{tail1}(x) &\equiv \exists y. \big( root(x) \wedge leftreach(y, x) \wedge \nexists z. succ_l(x, z) \big) \\
\alpha_{tail2}(x) &\equiv \exists y. \big( root(x) \wedge \neg leftreach(y, x) \wedge \nexists z. succ_l(x, z) \big) \\
\alpha_{NIL}(x) &\equiv root(x) \\
\beta_n(x_1, x_2) &\equiv \big( succ_l(x_1, x_2) \wedge \neg root(x_1) \big) \ \vee \big( root(x_2) \wedge \nexists z. succ_l(x_1, z) \big)
\end{aligned}
$$

*where the predicate $root(x)$ and the relation $leftreach(y, x)$ are defined in MSO: $root(x)$ is defined in the same way as in Example 3.3.2; $leftreach(y, x)$ indicates the reachability via the left-only path from $y$ to $x$.*

### 3.3.2 Submodels

We need to define the notion of *submodels* of a model. The definition of a submodel will depend on the particular RDDS we are working with, since we want to exploit the tree-representation of the models, which in turn will play a crucial role in deciding fragments of Strand, as it will allow us to check

satisfiability-preserving embeddings. In fact, we will define the submodel relation between trees that satisfy $\psi_{Tr}$.

We first define *valid subsets* of a tree, with respect to a recursive data-structure. this will then be used to define submodels.

**Definition 3.3.4** (Valid subsets). *Let $\mathcal{R} = (\psi_{Tr}, \psi_U, \{\alpha_a\}_{a \in L_v}, \{\beta_b\}_{b \in L_e})$ and $T = (V, \lambda)$ be a $\Sigma$-labeled tree that satisfies $\psi_{Tr}$, and let $S \subseteq V$. Then we say $S$ is a* valid subset *of $V$ if the following hold:*

- *$S$ is non-empty, and least-common-ancestor closed (i.e. for any $s, s' \in S$, the least common ancestor of $s$ and $s'$ wrt $T$ also belongs to $S$);*
- *The* subtree *defined by $S$, denoted by Subtree$(T, S)$, is the tree with nodes $S$, and where the $i$-th child of a node $u \in S$ is the (unique) node $u' \in S$ closest to $u$ that is in the subtree rooted at the $i$'th child of $u$. (This is uniquely defined since $S$ is least-common-ancestor (LCA) closed.) We require that Subtree$(T, S)$ also satisfy $\psi_{Tr}$;*
- *for every $s \in S$, if $\psi_U(s)$ holds in Subtree$(T, S)$, then $\psi_U(s)$ holds in $T$ as well;*
- *for every $s \in S$, for every $a \in L_v$, $\alpha_a(s)$ holds in Subtree$(T, S)$ iff $\alpha_a(s)$ holds in $T$.* □

Figure 3.3b shows a valid subset $S$ of the binary tree representation $T$ in Example 3.3.2. Now we can define a submodel based on a valid subset:

**Definition 3.3.5** (Submodels). *A tree $T' = (V', \lambda')$ is said to be a* submodel *of $T = (V, \lambda)$ if there is a valid subset $S$ of $V$ such that $T'$ is isomorphic to Subtree$(T, S)$.*

Note that while unary predicates ($\alpha_a$) are preserved in the submodel, the edge-relations ($\beta_b$) may be very different than its interpretation in the larger model. Intuitively, $T' = (V', \lambda')$ is a submodel of $T = (V, \lambda)$ if the vertices of $T'$ can be embedded in $T$, preserving the tree-structure. The nodes of the *Graph*$(T')$, are a subset of the nodes of *Graph*$(T)$ (because of the last condition in the definition of a submodel), and, given a valid subset $S$, there is in fact an injective mapping from the nodes of *Graph*$(T')$ to *Graph*$(T)$.

$$
\begin{array}{rcl}
tailor_X(succ_i(s,t)) & = & \exists s'.\Big(E_i\,(s,s') \ \wedge\ s' \leq t \ \wedge \\
& & \quad \forall t'.\big(\,(t' \in X \wedge s' \leq t') \Rightarrow t \leq t'\big)\Big) \\
& & \text{for every } i \in [k]. \\
tailor_X(s=t) & = & (s=t) \\
tailor_X(s \in W) & = & s \in W \\
tailor_X(\delta_1 \vee \delta_2) & = & tailor(\delta_1) \vee tailor_X(\delta_2) \\
tailor_X(\neg\delta) & = & \neg tailor_X(\delta) \\
tailor_X(\exists s.\delta) & = & \exists s.\big(s \in X \wedge tailor_X(\delta)\big) \\
tailor_X(\exists W.\delta) & = & \exists W.\big(W \subseteq X \wedge tailor_X(\delta)\big)
\end{array}
$$

Figure 3.4: Definition of the $tailor_X$ function

### 3.3.3 Interpreting Formulas on Submodels

We define a transformation $tailor_X$ from an MSO formula on trees to another MSO formula (with a free set variable $X$) on trees, such that for any MSO sentence $\delta$ on $k$-ary trees, for any tree $T=(V,\lambda)$ and any valid subset $X \subseteq V$, $Subtree(T,X)$ satisfies $\delta$ iff $T$ satisfies $tailor_X(\delta)$. The transformation is given in Figure 3.4, where we let $x \leq y$ mean that $y$ is a descendent of $x$ in the tree. The crucial transformations are the edge-formulas, which are interpreted as the edges of the subtree defined by $X$.

Now by the definition of valid subsets, we define a predicate $ValidSubset(X)$ using MSO, where $X$ is a free set variable, such that $ValidSubset(X)$ holds in a tree $T=(V,\lambda)$ iff $X$ is a valid subset of $V$ (below, *lca(x,y,z)* stands for a MSO formula says that $z$ is the LCA of $x$ and $y$ in the tree).

$$
\begin{array}{rl}
ValidSubset(X) \ \equiv & \forall s,t,u.\Big(\big(s \in X \wedge t \in X \wedge lca(s,t,u)\big) \Rightarrow u \in X\Big) \\
& \wedge\ tailor_X(\psi_{Tr}) \wedge \Bigg(\forall s.\Big(s \in X \wedge tailor_X\big(\psi_U(s)\big)\Big) \Rightarrow \psi_U(s)\Bigg) \\
& \wedge \bigwedge_{a \in L_v} \Bigg[\ \forall s.\Bigg(s \in X \Rightarrow \Big(tailor_X\big(\alpha_a(s)\big) \Leftrightarrow \alpha_a(s)\Big)\Bigg)\ \Bigg]
\end{array}
$$

### 3.3.4 Elasticity

Elastic relations are relations of the recursive data-structure that satisfy the property that a pair of nodes satisfy the relation in a tree iff they also satisfy the relation in any valid subtree. Formally,

**Definition 3.3.6** (Elastic relations)**.** *Let* $\mathcal{R} = (\psi_{Tr}, \psi_U, \{\alpha_a\}_{a \in L_v}, \{\beta_b\}_{b \in L_e})$*, and let* $b \in L_e$ *be an edge label. Then the relation* $E_b$ *(defined by* $\beta_b$*) is* elastic *if for every tree* $T = \{V, \lambda\}$ *satisfying* $\psi_{Tr}$*, for every valid subset* $S$ *of* $V$*, and for every pair of nodes* $u, v$ *in the model* $M' = \mathit{Graph}(\mathit{Subtree}(T, S))$*,* $E_b(u, v)$ *holds in* $M'$ *iff* $E_b(u, v)$ *holds in* $\mathit{Graph}(T)$*.* □

For example, let $\mathcal{R}$ be the class of binary trees, the *left-descendent* relation relating a node with any of the nodes in the tree subtended from the left child, is elastic, because for any binary tree $T$ and any valid subset of $S$ containing nodes $x$ and $y$, if $y$ is in the left branch of $x$ in $T$, $y$ is also in the left branch of $x$ in the subtree defined by $S$, and vice versa. However, the *left-child* relation is non-elastic. Consider a binary tree $T$ in which $y$ is in the left branch of $x$ but not the left child of $x$, then $S = \{x, y\}$ is a valid subset, and $y$ is the left child of $x$ in $\mathit{Subtree}(T, S)$.

It turns out that elasticity of $E_b$ can also be expressed by the following MSO formula

$$
\begin{aligned}
\psi_{Tr} \;\Rightarrow\; \forall X\ \forall u\ \forall v.\ \Big[ \Big( & \mathit{ValidSubset}(X) \wedge u{\in}X \wedge v{\in}X \wedge \\
& \mathit{tailor}_X\big(\psi_U(u)\big) \wedge \mathit{tailor}_X\big(\psi_U(v)\big)\Big) \\
\Rightarrow\ & \Big(\beta_b(u, v) \Leftrightarrow \mathit{tailor}_X\big(\beta_b(u, v)\big)\Big) \Big]
\end{aligned}
$$

$E_b$ is elastic iff the above formula is valid over all trees. Hence, we can *decide* whether a relation is elastic or not, by checking the validity of the above formula over $k$-ary $\Sigma$-labeled trees.

## 3.4 The Logic

We now introduce the Strand ("STRucture ANd Data") logic. Strand is a two-sorted logic parameterized by a first-order theory $\mathcal{D}$ of sort *Data*, and an RDDS $\mathcal{R} = (\psi_{Tr}, \psi_U, \{\alpha_a\}_{a \in L_v}, \{\beta_b\}_{b \in L_e})$ the syntax of Strand is presented in Figure 3.5. Strand is defined over the two-sorted signature $\Gamma(\mathcal{D}, \mathcal{R}) = \mathit{Sig}(\mathcal{D}) \cup \mathit{Sig}(\mathcal{R}) \cup \mathit{DF}$, where $\mathit{DF}$ is a set of functions of sort $\mathit{Loc} \rightarrow \mathit{Data}$. Strand formulas are of the form $\exists\vec{x}\forall\vec{y}.\varphi(\vec{x}, \vec{y})$, where $\vec{x}$ and $\vec{y}$ are $\exists$DVar and $\forall$DVar, respectively, of sort *Loc* (we also refer to both as DVar), $\varphi$ is a MSO formula with atomic formulas of the form either $Q_a(b)$,

$E_b(v, v')$ or $\gamma(e_1, \ldots, e_n)$, where $\gamma(e_1, \ldots, e_n)$ is an atomic $\mathcal{D}$-formula in which the data carried by *Loc*-variables can be referred as $df(x)$ or $df(y)$. Note that additional variables are allowed in $\varphi(\vec{x}, \vec{y})$, both first-order and second-order, but $\gamma(e_1, \ldots, e_n)$ is only allowed to refer to $\vec{x}$ and $\vec{y}$.

| | | | |
|---|---|---|---|
| Formula | $\psi$ | $::=$ | $\exists x.\psi \mid \omega$ |
| $\forall$Formula | $\omega$ | $::=$ | $\forall y.\omega \mid \varphi$ |
| QFFormula | $\varphi$ | $::=$ | $\gamma(e_1, \ldots, e_n) \mid Q_a(v) \mid E_b(v, v')$ |
| | | | $\mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2$ |
| | | | $\mid \exists z.\varphi \mid \forall z.\varphi \mid \exists S.\varphi \mid \forall S.\varphi$ |
| Expression | $e$ | $::=$ | $df(x) \mid df(y) \mid c \mid g(e_1, \ldots, e_n)$ |

| | | | |
|---|---|---|---|
| $\exists$DVar | $x$ | $\in$ | *Loc* |
| $\forall$DVar | $y$ | $\in$ | *Loc* |
| GVar | $z$ | $\in$ | *Loc* |
| Var | $v$ | $::=$ | $x \mid y \mid z$ |
| SVar | $S$ | $\in$ | $2^{Loc}$ |
| Constant | $c$ | $\in$ | $Sig(\mathcal{D})$ |
| Function | $g$ | $\in$ | $Sig(\mathcal{D})$ |
| $\mathcal{D}$-Relation | $\gamma$ | $\in$ | $Sig(\mathcal{D})$ |
| $\mathcal{L}$-Relation | $b$ | $\in$ | $Sig(\mathcal{L})$ |
| Data Field | $df$ | $\in$ | $DF$ |

Figure 3.5: Syntax of STRAND

The STRAND logic is interpreted on a structure $\mathcal{M} = (G_{\mathcal{R}}, M_{DF})$. $G_{\mathcal{R}}$ is a graph in *Graph*($\mathcal{R}$) with $N$ as the underlying set of nodes, and $M_{DF}$ is set of functions of the form $M_{DF} : N \rightarrow Data$. The semantics of STRAND formulas is the natural extension of $\mathcal{R}$ and $\mathcal{D}$.

We will refer to $G_{\mathcal{R}}$ as a *graph-model*. A *data-extension* of $G_{\mathcal{R}}$ is a STRAND model $(G_{\mathcal{R}}, M_{DF})$.

Then every STRAND model $\mathcal{M}$ can be canonically regarded as an $\mathcal{L}$-model. More precisely,

**Definition 3.4.1.** *Given a* STRAND *model* $\mathcal{M} = (G_{\mathcal{R}}, M_{DF})$, *the* structural-reduct *of* $\mathcal{M}$ *is* $G_{\mathcal{R}}$. *We also denote it as* $\mathcal{M}^{str}$.

For the reverse direction, an $\mathcal{L}$-model can be populated with data values to form a STRAND model:

**Definition 3.4.2.** *Given a* STRAND *model* $\mathcal{M}$ *and an* $\mathcal{L}$*-model* $\mathcal{N}$*,* $\mathcal{M}$ *is an* data-extension *of* $\mathcal{N}$ *if* $\mathcal{N}$ *is the structural-reduct of* $\mathcal{M}$*.*

By an abuse of language, in the rest of the dissertation, we sometimes call a STRAND model satisfies an MSO $\mathcal{L}$-formula if its structural-reduct does so, and call an $\mathcal{L}$-model satisfies an STRAND formula if there is a data-extension of it does so.

### 3.4.1 Examples

We now show various examples to illustrate the expressiveness of STRAND. In the following examples, we assume that $DF$ contains a single data field $d$.

**Example 3.4.3** (Sorted list)**.** *We first revisit the motivating Example 3.2.1 presented in Section 3.2. In the formula* $\varphi_1$*,* $y_1$ *and* $y_2$ *must be in* $DVar$ *since their data fields are referred. Thus* $\varphi_1$ *can be rewritten in* STRAND *as*

$$\psi_{sorted} \equiv \forall y_1 \forall y_2 .(\ d(\texttt{head}){=}c_1 \ \wedge\ d(\texttt{tail}) = c_2 \wedge$$

$$((y_1 \rightarrow^* y_2) \Rightarrow d(y_1) \leq d(y_2))\ )$$

**Example 3.4.4** (Min-heap)**.** *A* Min-heap *is a heap, which can be represented as a binary tree, that satisfies the min-heap property, which can be stated in* STRAND *as*

$$\psi_{minheap} \equiv \forall y_1 \forall y_2 .((y_1 \rightarrow^* y_2) \Rightarrow d(y_1) \leq d(y_2))$$

Both $y_1$ and $y_2$ are universally quantified data-variables in DVar.

**Example 3.4.5** (Binary search tree)**.** *In* STRAND*, a* binary search tree *(BST) is described as a binary tree data structure with an additional* key *field for each node. The keys in a BST are always stored in such a way as to satisfy the binary-search-tree property:*

- *The left subtree of a node contains only nodes with keys less than the node's key.*

- *The right subtree of a node contains only nodes with keys greater than the node's key.*

*Let the binary trees be defined as in Example 3.3.2 and let the key field be accessed by function d, this property can be expressed in* STRAND *as follows:*

$$\psi_{bst} \equiv \forall y_1, y_2.\Big(\big(leftsubtree(y_1,y_2) \Rightarrow d(y_2) < d(y_1)\big) \wedge \\ \big(rightsubtree(y_1,y_2) \Rightarrow d(y_1) \geq d(y_2)\big)\Big)$$

*where*

$$\begin{aligned} leftsubtree(y_1,y_2) &\equiv \exists z.(l(y_1,z) \wedge z \rightarrow^* y_2) \\ rightsubtree(y_1,y_2) &\equiv \exists z.(r(y_1,z) \wedge z \rightarrow^* y_2) \\ x \rightarrow^* y &\equiv \exists S.\big(x \in S \wedge y \in S \wedge \\ &\qquad \forall z.\big(z \in S \rightarrow (z = x \vee \exists u.(l(u,z) \vee r(u,z)))\ \big)\big) \end{aligned}$$

Note that $\psi_{bst}$ has an existentially quantified variable $z$ in GVar after the universal quantification of $y_1, y_2$. However, as $z$ is a structural quantification (whose data-field cannot be referred to), this formula is in STRAND.

**Remark:** The $\forall\exists$ alternation is unavoidable to express the BST property, but is excluded by many decidable logic fragments. However, STRAND is expressive enough to allow it, because STRAND allows arbitrary quantification combinations within the $\exists\forall$ quantifications of DVar's, as long as the data field of these quantified variables are not referred.

**Example 3.4.6** (Two disjoint lists)**.** *In* separation logic*[68], a novel binary operator* $*$*, or* separating conjunction*, is defined to assert that the heap can be split into two disjoint parts where its two arguments hold, respectively. Such an operator is useful in reasoning with* frame conditions *in program verification. Thanks to the powerful expressiveness of MSO logic, the separating conjunction is also expressible in* STRAND*. For example,* $(\texttt{head}_1 \rightarrow^* \texttt{tail}_1) * (\texttt{head}_2 \rightarrow^* \texttt{tail}_2)$ *states, in separation logic, that there are two disjoint lists such that one list is from* $\texttt{head}_1$ *to* $\texttt{tail}_1$*, and the other is from* $\texttt{head}_2$ *to* $\texttt{tail}_2$*. Let the RDDS for two lists be as defined in Exam-*

*ple 3.3.3, then this formula can be written in* STRAND *as:*

$$\begin{aligned}\psi_{2lists} \quad \equiv \quad & \exists S_1, S_2.(\mathit{disjoint}(S_1, S_2) \quad \wedge \mathtt{head}_1 \in S_1 \wedge \mathtt{tail}_1 \in S_1 \quad \wedge \\ & \mathtt{head}_2 \in S_\wedge \mathtt{tail}_2 \in S_2 \quad \wedge \mathtt{head}_1 \rightarrow^* \mathtt{tail}_1 \wedge \mathtt{head}_2 \rightarrow^* \mathtt{tail}_2) \quad \wedge \\ & (\forall z(\mathtt{head}_1 \rightarrow^* z \wedge z \rightarrow^* \mathtt{tail}_1) \Rightarrow z \in S_1) \quad \wedge \\ & (\forall z(\mathtt{head}_2 \rightarrow^* z \wedge z \rightarrow^* \mathtt{tail}_2) \Rightarrow z \in S_2)\end{aligned}$$

*where*

$$\mathit{disjoint}(S_1, S_2) \equiv \neg \exists z.(z \in S_1 \wedge z \in S_2)$$

Note that there are no data fields referred to in $\psi_{2lists}$, i.e., it is merely an MSO formula that describes the heap structure. This example shows that STRAND inherits the full power of MSO logic from the structure side.

**Remark:** In STRAND, each formula can be brought into its negation normal form by using De Morgan's Laws to push a negation inside, and eliminating double negations. The resulting formula remains in STRAND.

## 3.5 Program Verification Using STRAND

STRAND can be used to reason about the correctness of programs, in terms of verifying Hoare-triples where the pre- and post-conditions express both the structure of the heap as well as the data contained in them. The pre- and post-conditions that we allow are STRAND formulas that consist of Boolean combinations of the formulas with pure existential or pure universal quantification over the data-variables (i.e. Boolean combinations of formulas of the form $\exists \vec{x}.\varphi$ and $\forall \vec{y}.\varphi$); let us call this fragment $\text{STRAND}_{\exists,\forall}$.

Given a straight-line program $P$ that does destructive pointer-updates and data updates, we model a Hoare-triple as a tuple $(\mathcal{R}, \mathit{Pre}, P, \mathit{Post})$, where the pre-condition is given by the data-structure constraint $\mathcal{R}$ with the $\text{STRAND}_{\exists,\forall}$ formula $\mathit{Pre}$, and the post-condition is given by the $\text{STRAND}_{\exists,\forall}$ formula $\mathit{Post}$ (note that structural constraints on the data-structure for the post-condition are also expressed in $\mathit{Post}$, using MSO logic).

In this section, we show that given such a Hoare-triple, we can reduce checking whether the Hoare-triple is *not* valid can be reduced to a satisfiability problem of a STRAND formula over a class of recursively defined data-structures $\mathcal{R}_P$. This then allows us to use STRAND$_{\exists,\forall}$ to verify programs (where, of course, loop-invariants are given by the programmer, which breaks down verification of a program to verification of straight-line code). Intuitively, this reduction *augments* the structures in $\mathcal{R}$ with extra nodes that could be created during the execution of $P$, and models the *trail* the program takes by logically defining the configuration of the program at each time instant. Over this trail, we then express that the pre-condition holds and the post-condition fails to hold. We also construct formulas that check if there is any memory access violation during the run of $P$ (e.g. free-ing locations twice, dereferencing a null pointer, etc.).

### 3.5.1 Syntax of Programs

Let us define the syntax of a basic programming language manipulating heaps and data; more complex constructs can be defined by combining these statements appropriately. Let *Var* be a countable set of *pointer variables*, $F$ be a countable set of *structural pointer fields*, and `data` be a *data field*. A *condition* is defined as follows: (for technical reasons, negations are pushed all the way in):

$$\begin{aligned}\psi \in \texttt{Cond} ::= &\ \gamma(q^1.\texttt{data}, \ldots, q^k.\texttt{data}) \mid \neg\gamma(q^1.\texttt{data}, \ldots, q^k.\texttt{data}) \\ &\mid p == q \mid p \neq q \mid p == nil \mid p \neq nil \mid \psi_1 \wedge \psi_2 \mid \psi_1 \vee \psi_2\end{aligned}$$

where $p, q, q^1, \ldots, q^k \in Var$, and $\gamma$ is a predicate over data values. The set of statements `Stmt` defined over *Var*, $F$, and `data` is defined as follows:

$$\begin{aligned}s \in \texttt{Stmt} ::= &\ p := new \mid free(p) \mid \texttt{assume}(\psi) \mid p := nil \\ &\mid p := q \mid p.f := q \mid p := q.f \mid p.\texttt{data} := h(q^1.\texttt{data}, \ldots, q^k.\texttt{data})\end{aligned}$$

where $p, q, q^1, \ldots, q^k \in Var$, $f \in F$, $h$ is a function over data, and $\psi$ is a condition. A *program* $P$ over *Var*, $F$, and `data` is a non empty finite sequence of statements $s_1; s_2; \ldots; s_m$, with $s_i \in \texttt{Stmt}$.

Here forward, whenever we refer to statements, programs, conditions, graphs, recursively data-structures, and STRAND formulas, we assume that are defined over the same sets *Var*, $F$, and `data`.

### 3.5.2 Semantics of Programs

The semantics of the statements in `Stmt`, and hence the semantics of programs is as follows. A statement of a program $P$ operates on graphs $G$ which have the following template restrictions: $G = (V, E, \mu, \nu, L_v, L_e)$, where $L_v = \mathit{Var} \cup \{\mathit{xnil}\}$, and $L_e = F$, with the constraint that:

- Every pointer variable $p \in \mathit{Var}$ labels exactly one node;
- Every edge is labeled by a unique symbol of $F$;
- For each node $v \in V$, $v$ has exactly one outgoing edge labeled by $f$, for every $f \in F$;
- $G$ has a special node, called *nil*, which is labeled by *xnil*.

Node *nil* models an undefined area of memory. In the following whenever a node $v \in V$ is labeled by $p$ (that is, $p \in \mu(v)$), we say that $p$ points to $v$.

Let $\mathcal{R}$ be a recursive data-structure, *PreandPost* be two STRAND $_{\exists,\forall}$ formulas, and $P ::= s_1; s_2; \ldots; s_m$ be a program. The configuration of the program at any point is given by a heap modeled as a graph, where nodes of the graph are assigned data values. For a program with $m$ statements, let us fix the configurations to be $G_0, \ldots, G_m$.

The execution of a statement $s \in \mathit{Stmt}$ on a graph $G = (V, E, \mu, \nu, L_v, L_e)$ has the effect of transforming $G$ into another graph $G' = (V', E', \mu', \nu', L_v, L_e)$. Given a statement $s \in$ `Stmt` and a graph $G$, we define a relation $G \hookrightarrow_s G'$ that capture the graph transformation according to the semantics of $s$, as following:

[$\boldsymbol{p}$ := `new` ]: $G'$ is obtained by $G$ by adding a new node which is now pointed by the variable $p$. All the edges outgoing from the new node will point to *nil*.

[$\texttt{free}(\boldsymbol{p})$ ]: If $p$ points to the node $v$ of $G$, which is not *nil*, then $G'$ is obtained by $G$ by removing $v$, where now all the edges and pointers that were pointing to $v$ now will point to *nil*. Conversely, if $p$ points to *nil*, then the program terminates with an error, and $G \not\rightarrow_s G'$, for any graph $G'$.

[$\boldsymbol{p} := \texttt{nil}$ ]: $G'$ is the same as $G$ except that $p$ now points to *nil*. This statement will never lead to an error, for any graph $G$.

[$\boldsymbol{p} := \boldsymbol{q}$ ]: $G'$ is the same as $G$ with the exception that $p$ points to the same node as $q$. The statement $p := q$ never goes wrong.

[$\boldsymbol{p.f} := \boldsymbol{q}$ ]: If $p$ does not point to *nil*, $G'$ is obtained by $G$ by modifying only the $f$-edge outgoing from the node pointed by $p$ that now points to the same node as $q$ (which can also be the *nil* node). Otherwise, $p$ points to *nil* and the execution ends with an error ($G \not\rightarrow_s G'$, for any $G'$).

[$\boldsymbol{p} := \boldsymbol{q.f}$ ]: If $q$ does not point to *xnil*, $G'$ is structurally the same as $G$ except that the variable $p$ now points to the same node as that pointed by the $f$-labeled edge outgoing from the node pointed by $q$. Instead, if $q$ points to *nil* the execution of the statement ends with an error ($G \not\rightarrow_s G'$, for any $G'$).

[$\texttt{assume}(\psi)$ ]:

If all pointer variables in the condition $\psi$ point to a node which is not *nil*, then (1) $G'$ is the same as $G$ provided that $\psi$ holds on $G$, (2) the program gets blocked in case $\psi$ does not hold on $G$. Otherwise, there is a pointer variable of $\psi$ that points to *nil*, and the execution of the statement terminates with an error.

[$\boldsymbol{p}.\texttt{data} := \boldsymbol{h}(\boldsymbol{q}^1.\texttt{data}, \ldots, \boldsymbol{q}^k.\texttt{data})$ ]: If $p$ and $q^1, \ldots, q^k$ are all not pointing to *nil* in $G$, then $G'$ is the same as $G$ except that the data-value associated to the node pointed by $p$ is now updated with the new value $h(q^1.\texttt{data}, \ldots, q^k.\texttt{data})$, where $q^j.\texttt{data}$ is the data value associated to the node pointed by $q^j$, for every $j \in [k]$.

Otherwise, if $p$ or one among $q^j$ pointers points to *nil*, the execution of the statement terminates with an error and $G \not\rightarrow_s G'$, for any $G'$.

Let $P = s_1; s_2; \ldots; s_m$, be a program and $G_0$ be a graph. Furthermore, let $P = s_1; s_2; \ldots; s_i$ be the program obtained by $P$ taking only the first $i$ statements of $P$.

We define the relation $G_0 \hookrightarrow_{P_i} G_i$ if there exists a sequence of graphs $G_1, G_2, \ldots, G_{m-1}$ such that $G_{j-1} \hookrightarrow_{s_j} G_i$, for every $j \in [m]$. In other words, the relation $\hookrightarrow_{P_i}$ captures the effect/result ($G_i$) that the execution of $P_i$ has on the initial graph $G_0$. Here forward, for every graph $G$, we denote with $G_i^P$ the graph such that $G \hookrightarrow_{P_i} G_i^P$, and with $G^P$ to be the graph such that $G \hookrightarrow_{P_m} G^P$ which corresponds to the graph obtained at the end of the execution of $P$ starting with the graph $G$.

### 3.5.3 The Problem

Let $\mathcal{R}$ be a recursively data-structure, $Pre, Post$ be two STRAND formulas, and $P ::= s_1; s_2; \ldots; s_m$ be a program. The problem we want to address is that of checking whether there exists a graph $G \in Graph(\mathcal{R})$ on which $Pre$ holds, and either (1) $G^P$ exists and $Post$ does not hold, or (2) there exists an index $i$ such that $G_i^P$ exists and the execution of $s_{i+1}$ gives an error.

We show that such a problem is reducible to the satisfiability problem of a STRAND formula over a newly recursively data-structure $\mathcal{R}_P$ that is defined by $\mathcal{R}$ and $P$ called the *trail*.

### 3.5.4 The Trail

The idea is to capture the entire computation starting from a particular data-structure using a single data-structure. The main intuition is that if we run $P$ over a graph $G_0 \in Graph(\mathcal{R})$ then a new class of recursive data-structures $\mathcal{R}_P$ will define a graph $G_{trail}$ which encodes in it $G_0$, as well as all the graphs $G_i$, for every $i \in [m]$. $G_{trail}$ has the nodes of $G_0$ plus $m$ other fresh nodes (these nodes will be used to model newly created nodes $P$ creates as well as to hold new data-values of variables that are assigned to in $P$). Each of these new nodes are pointed by a distinguished pointer variable $new_i$. Initially, these additional nodes are all inactive in $G_0$. We build an MSO-defined unary predicate $active_i$ that captures at each step $i$ the precise set of active nodes in the heap. To capture the pointer variables at each step

of the execution, we define a new unary predicate $p_i$, for each $p \in Var$ and $i \in [0, m]$. Similarly, we create MSO-defined binary predicates $f_i$ for each $f \in F$ and $i \in [0, m]$, to capture structural pointer fields at step $i$. The heap $G_i$ at step $i$ is hence the graph consisting of all the nodes $x$ of $G_{trail}$ such that $active_i(x)$ holds true, and the pointers and edges of $G_i$ are defined by $p_i$ and $f_i$ predicates, respectively.

Formally, fix a recursively defined data-structure $\mathcal{R} = (\psi_{Tr}, \psi_U, \{\alpha_p\}_{p \in Var}, \{\beta_f\}_{f \in F})$, with a monadic predicate $\alpha_{xnil}$, which evaluates to a unique NIL node in the data-structure. Then its trail with respect to the program $P$ is defined as $\mathcal{R}_P = (\psi'_{Tr}, \psi'_U, \{\alpha'_p\}_{p \in Var'}, \{\beta'_f\}_{f \in F'})$ where:

- $\psi'_{Tr}$ is designed to hold on all trees in which the first subtree of the root satisfies $\psi_{Tr}$ and the second child of the root has a chain of $m-1$ nodes where each of them is the second child of the parent.

- $\psi'_U$ holds true on the root, on all the second child descendent of the root, and on all first child descendent on which $\psi_U$ holds true.

- $Var' = \{new_i \mid i \in [m]\} \cup \{p_i \mid p \in Var, i \in [0, m]\}$, and
  -(1) $\alpha'_{new_1}$ holds only on the root, and $\alpha'_{new_i}$ holds true only on the $i+1$'th descendent of the second child of the root, for every $i \in [m-1]$.
  -(2) for every $p \in Var$ and $i \in [m]$, $\alpha'_{p_0} = \alpha_p$ and $\alpha'_{p_i}$ is defined as in Figure 3.6 and Figure 3.7.

- $F' = \{f_i \mid f \in F, i \in [0, m]\}$, and for every $f \in F$ and $i \in [m]$, $\beta'_{f_0} = \beta_f$ and $\beta'_{f_i}$ is defined as in Figure 3.6 and 3.7.

In Figure 3.6, the MSO formulas $\alpha'_{p_i}$ and $\beta'_{f_i}$ are derived from the semantics of the non-updating statements. In Figure 3.7, similar formulas are derived for the updating statements. All the formulas are derived in the natural way except $p.\texttt{data} := h(q^1.\texttt{data}, \ldots, q^k.\texttt{data})$. Although the semantics for this statement does not involve any structural modification of the graph (it changes only the data value associated $p$), we represent this operation by making a *new version* of the node pointed by $p$ in order to represent explicitly the change for the data value corresponding to that node. We deactivate the node pointed by $p_{i-1}$ and activate the dormant node pointed by $new_i$. All the edges in the graph and the pointers are rearranged to reflect this exchange of nodes.

$[\boldsymbol{p} := \texttt{nil}\ ]$:

$$\alpha'_{p_i(x)} = \alpha'_{xnil_{i-1}}(x),\ \ \alpha'_{z_i}(x) = \alpha'_{z_{i-1}}(x), \qquad \forall z \in Var \setminus \{p\}$$
$$\beta'_{f_i}(x, y) = \beta'_{f_{i-1}}(x, y), \qquad \forall f \in F$$
$$active_i(x) = active_{i-1}(x),\ \ error_i = \texttt{false}$$

---

$[\boldsymbol{p} := \boldsymbol{q}\ ]$:

$$\alpha'_{p_i}(x) = \alpha'_{q_{i-1}}(x),\ \ \alpha'_{z_i}(x) = \alpha'_{z_{i-1}}(x), \qquad \forall z \in Var \setminus \{p\}$$
$$\beta'_{f_i}(x, y) = \beta'_{f_{i-1}}(x, y), \qquad \forall f \in F$$
$$active_i(x) = active_{i-1}(x),\ \ error_i = \texttt{false}$$

---

$[\boldsymbol{p} := \boldsymbol{q}.\boldsymbol{f}\ ]$:

$$\begin{aligned}
\alpha'_{p_i(x)} &= \exists ex.\ (\alpha'_{q_{i-1}}(ex) \wedge \beta'_{f_{i-1}}(ex, x)) \\
\alpha'_{q_i}(x) &= \alpha'_{q_{i-1}}(x), \qquad \forall q \in (Var \setminus \{p\}) \\
\beta'_{f_i}(x, y) &= \beta'_{f_{i-1}}(x, y) \\
\beta'_{g_i}(x, y) &= \beta'_{g_{i-1}}(x, y), \qquad \forall g \in (F \setminus \{f\}) \\
active_i(x) &= active_{i-1}(x),\ \ error_i = \exists x.\ (\alpha'_{q_{i-1}}(x) \wedge \alpha'_{xnil_{i-1}}(x))
\end{aligned}$$

---

$[\texttt{assume}(\psi)\ ]$:

$$\begin{aligned}
\alpha'_{q_i}(x) &= \alpha'_{q_{i-1}}(x), \qquad \forall q \in Var \\
\beta'_{l_i}(x, y) &= \beta'_{l_{i-1}}(x, y), \qquad \forall l \in F \\
active_i(x) &= active_{i-1}(x), \\
error_i &= \exists x. \bigvee_{p \in Var(\psi)} (\alpha'_{p_{i-1}}(x) \wedge \alpha'_{xnil_{i-1}}(x))
\end{aligned}$$

where $Var^{\psi}$ is the set of all variables occurring in $\psi$.

Figure 3.6: Predicates defined for non-updating statements.

In Figure 3.6 and Figure 3.7, we also define two more MSO formulas, $active_i$ and $error_i$, which are not part of the trail, where the first models the active nodes at step $i$, and the second expresses when an error occurs due to the dereferencing of a variable pointing to *xnil*, respectively.

$[\boldsymbol{p} := \texttt{new}\ ]$:

$$\alpha'_{p_i}(x) = \alpha'_{new_i}(x), \quad \alpha'_{q_i}(x) = \alpha'_{q_{i-1}}(x), \quad \forall q \in Var \setminus \{p\},$$
$$\beta'_{f_i}(x,y) = \beta'_{f_{i-1}}(x,y), \quad \forall f \in F$$
$$active_i(x) = active_{i-1}(x) \vee \alpha'_{new_i}(x), \quad error_i = \texttt{false}$$

---

$[\texttt{free}(\boldsymbol{p})\ ]$:

$$\begin{aligned}
\alpha'_{z_i}(x) &= \big(\alpha'_{z_{i-1}}(x) \wedge (\alpha'_{xnil_{i-1}}(x) \vee \neg\alpha'_{p_{i-1}}(x))\big) \ \vee \\
&\quad \big(\alpha'_{xnil_{i-1}}(x) \wedge \neg\alpha'_{z_{i-1}}(x)\big) \\
\beta'_{f_i}(x,y) &= \big(\beta'_{f_{i-1}}(x,y) \wedge \neg\alpha'_{p_{i-1}}(x)\big) \ \vee \\
&\quad \big(\alpha'_{xnil_{i-1}}(y) \wedge \exists ex.\ (\beta'_{f_{i-1}}(x,ex) \wedge \alpha'_{p_{i-1}}(ex))\big) \\
active_i(x) &= active_{i-1}(x) \wedge \neg\alpha'_{p_{i-1}}(x) \\
error_i &= \exists x.(\alpha'_{p_{i-1}}(x) \wedge \alpha'_{xnil_{i-1}}(x))
\end{aligned}$$

---

$[\boldsymbol{p}.\boldsymbol{f} := \boldsymbol{q}\ ]$:

$$\begin{aligned}
\alpha'_{z_i}(x) &= \alpha'_{z_{i-1}}(x), \quad \forall z \in Var \\
\beta'_{f_i}(x,y) &= (\neg\alpha'_{p_{i-1}}(x) \wedge \beta'_{f_{i-1}}(x,y)) \vee (\alpha'_{p_i-1}(x) \wedge \alpha'_{q_{i-1}}(y)) \\
\beta'_{g_i}(x,y) &= \beta'_{g_{i-1}}(x,y), \quad \forall g \in (F \setminus \{f\}) \\
active_i(x) &= active_{i-1}(x), \\
error_i &= \exists x.(\alpha'_{p_{i-1}}(x) \wedge \alpha'_{xnil_{i-1}}(x))
\end{aligned}$$

---

$[\boldsymbol{p}.\texttt{data} := \boldsymbol{h}(\boldsymbol{q}^1.\texttt{data}, \ldots, \boldsymbol{q}^k.\texttt{data})\ ]$:

$$\begin{aligned}
\alpha'_{p_i}(x) &= \alpha'_{new_i}(x), \quad \alpha'_{q_i}(x) = \alpha'_{q_{i-1}}(x), \ \forall q \in Var \setminus \{p\} \\
\beta'_{f_i}(x,y) &= \big(\beta'_{f_{i-1}}(x,y) \wedge \neg\alpha'_{p_{i-1}}(x)\big) \ \vee \\
&\quad \big(\alpha'_{new_i}(y) \wedge \exists ex.\ (\beta'_{f_{i-1}}(x,ex) \wedge \alpha'_{p_{i-1}}(ex))\big) \\
active_i(x) &= (active_{i-1}(x) \wedge \neg p_{i-1}(x)) \vee \alpha'_{new_i}(x) \\
error_i &= \exists x.\Big( \bigvee_{z\ \in \{p, q^1, \ldots, q^k\}} (\alpha'_{z_{i-1}}(x) \wedge \alpha'_{xnil_{i-1}}(x)) \ \Big)
\end{aligned}$$

Figure 3.7: Predicates defined for updating statements.

### 3.5.5 Handling Data Constraints

The trail $\mathcal{R}_P$ captures all the structural modifications made to the graph during the execution $P$. However, data constrains entailed by `assume` statements and data-assignments cannot be expressed in the trail as they are not expressible in MSO. We impose them in the STRAND formula. We define a formula $\varphi_i$ for each statement index $i \in [m]$, where if $s_i$ is not an assume or a data-assignment statement, then $\varphi_i = \texttt{true}$. Otherwise, there are two cases:

**Handling assume-statements.** If $s_i$ is the statement $\texttt{assume}(\psi)$, then $\varphi_i$ is the STRAND formula obtained by *adapting* the constraint $\varphi$ to the $i$'th stage of the trail. We use the recursive translation $\psi' = adapt_i^{\psi}$, defined in Figure 3.8, to adapt the condition $\psi$ to refer to the trail at step $i-1$. Formula $\psi'$ is not yet a STRAND formula since there may be existential quantifiers inside the formula that may involve data variables. However, all the existential quantifiers are not in the scope of any Boolean negation and hence all of them can be moved at the beginning of $\psi'$ (after renaming variables, if necessary). The resulting formula $\varphi_i$ is the STRAND formula associated to $s_i$.

$$
\begin{array}{rcl}
adapt_i^{p}(x) & := & \alpha'_{p_{i-1}}(x) \\
adapt_i^{\neg p}(x) & := & \neg\alpha'_{p_{i-1}}(x) \\
adapt_i^{p.f}(x) & := & \exists ex.(\beta'_{f_{i-1}}(ex,x) \wedge adapt_{i-1}^{p}(ex)) \\
adapt_i^{\neg p.f}(x) & := & \exists ex.\ (\beta'_{f_{i-1}}(ex,x) \wedge adapt_{i-1}^{\neg p}(ex)) \\
adapt_i^{p==q} & := & \exists ex.\ (adapt_{i-1}^{p}(ex) \wedge adapt_{i-1}^{q}(ex)) \\
adapt_i^{p\neq q} & := & \exists ex.\ (adapt_{i-1}^{p}(ex) \wedge adapt_{i-1}^{\neg q}(ex)) \\
adapt_i^{p==nil} & := & \exists ex.\ (adapt_{i-1}^{p}(ex) \wedge xnil_{i-1}(x)) \\
adapt_i^{p\neq nil} & := & \exists ex.\ (adapt_{i-1}^{p}(ex) \wedge \neg xnil_{i-1}(ex)) \\
adapt_i^{\psi_1\wedge\psi_2} & := & adapt_{i-1}^{\psi_1} \wedge adapt_{i-1}^{\psi_2} \\
adapt_i^{\psi_1\vee\psi_2} & := & adapt_{i-1}^{\psi_1} \vee adapt_{i-1}^{\psi_2} \\
adapt_i^{r(q^1.\texttt{data},\ldots,q^k.\texttt{data})} & := & \exists ex_1,\ldots,ex_k.\big(\ \bigwedge_{i\in[k]} adapt_{i-1}^{q^j}(ex_j)\big)\ \wedge \\
 & & \qquad r(\texttt{data}(ex_1),\ldots,\texttt{data}(ex_k)\ \big) \\
adapt_i^{\neg r(q^1.\texttt{data},\ldots,q^k.\texttt{data})} & := & \exists ex_1,\ldots,ex_k.\big(\ \bigwedge_{i\in[k]} adapt_{i-1}^{q^j}(ex_j)\big)\ \wedge \\
 & & \qquad \neg r(\texttt{data}(ex_1),\ldots,\texttt{data}(ex_k)\ \big)
\end{array}
$$

Figure 3.8: A syntactic transformation for conditions.

**Handling data-assignments.** The STRAND formula $\varphi_i$ for a data-assignment statement $p.\texttt{data} := h(q^1.\texttt{data}, \ldots, q^k.\texttt{data})$ is:

$$\begin{aligned}\varphi_i \;:=\; & \exists ex, ex_1, \ldots, ex_k.\; p_i(ex) \;\wedge \\ & (\bigwedge_{i \in [k]} q^j_{i-1}(ex_j)) \wedge \texttt{data}(ex) = h(\texttt{data}(ex_1), \ldots, \texttt{data}(ex_k))\end{aligned}$$

which translates $s_i$ into STRAND making sure that it refers to the heap at step $i-1$.

### 3.5.6 Adapting Pre- and Post-Conditions to the Trail

The last ingredient that we need is to express the STRAND $_{\exists,\forall}$ formulas *Pre* and the negation of the *Post* on the trail $\mathcal{R}_P$. More specifically, we need to adapt *Pre* to the trail for index 0, which corresponds to the original graph, i.e. the predicates $p$ are replaced with $p_0$, for every $p \in \mathit{Var}$, and the edge predicates $f$ with $f_0$, for every $f \in F$. Moreover, whenever we refer to a node in the graph we need to be sure that node is active which can be done by using the predicate $\mathit{active}_0(x)$ which holds true if $x$ is in the first subtree of the root and $\psi'_U(x)$ holds. A similar transformation is done for the formula $\neg \mathit{Post}$, where now we consider pointers, edge labels, and active nodes at the last step $m$. Let $\mathit{Pre}_{\mathcal{R}_P}$ (resp., $\mathit{Post}_{\mathcal{R}_P}$) be the STRAND formula corresponding to the adaptation of *Pre* (resp., *Post*).

### 3.5.7 Reduction to Satisfiability Problem on the Trail

It is easy to see that an error occurs during the execution of $P$ on a graph defined through $\mathcal{R}$ that satisfies *Pre* if the following STRAND formula is satisfiable on the trail $\mathcal{R}_P$:

$$\mathit{Error} = \bigvee_{i \in [m]} (\mathit{Pre}_{\mathcal{R}_P} \wedge \bigwedge_{j \in [i-1]} \varphi_j \quad \wedge \quad \mathit{error}_i)$$

Similarly, the Hoare-triple is not valid iff the following STRAND formula is satisfiable on the trail:

$$Violate_{Post} = Pre_{\mathcal{R}_P} \wedge (\bigwedge_{i\in[m]} \varphi_i) \wedge \neg Post_{\mathcal{R}_P}$$

Therefore, the main result of the section which claims that the verification of programs defined with recursively data-structures can be reduced to the satisfiability of a STRAND formula is formally stated as follows.

**Theorem 3.5.1.** *Let $P$ be a program, $\mathcal{R}$ be an RDDS, and $Pre, Post$ be two* STRAND$_{\exists,\forall}$ *formulas over Var, $F$, and* `data`*. Then, there is a graph $G \in Graph(\mathcal{R})$ that satisfies Pre and where either $P$ terminates with an error or the obtained graph $G'$ does not satisfy Post iff the* STRAND *formula $Error \vee Violate_{Post}$ is satisfiable on the trail $\mathcal{R}_P$.*

*Proof.* Let us fix $P$ as a basic block consisting of $m$ statements: $s_0, \ldots, s_{m-1}$. Then the soundness of the VC-generation can be proved by showing:

1. Each memory-error free run of $P$ can be represented by a model of $\mathcal{R}_P$ satisfying $\bigwedge_{i\in[m]} \varphi_i$;
2. If a run encounters memory error at the $j$-th statement, it can be represented by a model of $\mathcal{R}_P$ satisfying $Pre_{\mathcal{R}_P} \wedge \bigwedge_{j\in[i-1]} \varphi_j \quad \wedge \quad error_i$;
3. The pre- and post-conditions of a successful run can be captured by *Pre* and *Post* over $\mathcal{R}_P$, respectively.

Now we prove the three claims as follows.

**Successful runs.** For the $i$-th statement, the structural modification is captured by the constraints on $\alpha'$ and $\beta'$ in Figure 3.6 and 3.7. For example, if the $i$-th statement is $p.f := q$, the variable store is not modified at all, so $\bigwedge_{z\in Var} \left(\alpha'_{z_i}(x) = \alpha'_{z_{i-1}}(x)\right)$. For the heap, the fields other than $f$ are also unchanged, so $\bigwedge_{g\in(F\setminus\{f\})} \left(\beta'_{g_i}(x, y) = \beta'_{g_{i-1}}(x, y)\right)$. For the $f$ field, $\beta'_{f_i}(x, y)$ holds in two cases: either $x$ is not pointed by $p$ at the $(i-1)$-th configuration and $\beta'_{f_{i-1}}(x, y)$ holds, or $x$ is pointed by $p$ at the $(i-1)$-th configuration, and $y$ is pointed by $q$ at the $(i-1)$-th configuration. These modifications are formally captured by the predicate definitions in Figure 3.7, and we leave the other cases to the reader to verify.

The extra structural and data modifications for the $i$-th statement is captured by the formula $\varphi_i$. There are only two non-trivial cases: the assume-statements and the data-assignments. The assume-statement $\texttt{assume}(\psi)$ guarantees that the $\psi$ is satisfied at the $(i-1)$-th configuration. In this case, $\varphi_i$ is just the formula $adapt_i^{\psi}$ defined in Figure 3.8, which inductively interprets $\psi$ in the signature of $\mathcal{R}_P$. In particular, for a constraint $r(q^1.\texttt{data}, \ldots, q^k.\texttt{data})$, $\varphi_i$ guesses the nodes $ex_1, \ldots, ex_k$ that are pointed by $q^1, \ldots q^k$, respectively, and asserts the relation $r$ holds over the data fields of these nodes. As to the data-assignment $p.\texttt{data} := h(q^1.\texttt{data}, \ldots, q^k.\texttt{data})$, where $h$ is an expression in the data-logic, the formula $\varphi_i$ introduces $ex_1, \ldots, ex_k$ similarly, and the node $ex$ for $p$. $ex_1, \ldots, ex_k$ represent where $q^1, \ldots q^k$ point to at the $(i-1)$-th configuration, and $ex$ represents where $p$ points to at the $i$-th configuration. Moreover, the data value of $ex$ is equal to $h(\texttt{data}(ex_1), \ldots, \texttt{data}(ex_k))$.

**Memory error at the $j$-th statement.** If an memory error occurs in the $j$-th statement, we assume that the previous $i-1$ statements are executed successfully. Then the memory error can be captured by the corresponding formula $error_i$ defined in Figure 3.6 and 3.7. Intuitively, whenever a variable $p$ is dereferenced or mutated, $error_j$ asserts that $\exists x.(\alpha'_{p_{j-1}}(x) \wedge \alpha'_{xnil_{j-1}}(x))$. In particular, for the data-assignment of the form $p.\texttt{data} := h(q^1.\texttt{data}, \ldots, q^k.\texttt{data})$, in which $k$ variables are dereferenced and one variable is modified, $error_j$ consists of $k+1$ conjuncts, each for one variable.

**Pre- and post-conditions.** Note that each model of the RDDS $\mathcal{R}_P$ encodes an execution of the basic block $P$, every STRAND formula for every intermediate $i$-th configuration can be easily interpreted on $\mathcal{R}_P$: simply replace each $\alpha_p$ with $\alpha'_{p_i}$ and replace each $\beta_f$ with $\beta'_{f_i}$. In particular, when interpreting $Pre$ on the first configuration of $\mathcal{R}_P$ and interpreting $Post$ on the last configuration of $\mathcal{R}_P$, the obtained formulas are just $Pre_{\mathcal{R}_P}$ and $Post_{\mathcal{R}_P}$, respectively. □

# CHAPTER 4

# DECISION PROCEDURES

Now that we have introduced recursively defined data-structures and the STRAND logic, we turn our attention to checking satisfiability of STRAND.

## 4.1 Overview

In general, the satisfiability of STRAND is undecidable, even if both its underlying data-theories $\mathcal{D}$ is decidable.

**Theorem 4.1.1.** *The satisfiability of* STRAND *is undecidable.*

*Proof.* We show the undecidability via a reduction from the halting problem of 2-counter machines, which is a well-known undecidable problem [57].

Let $\mathcal{D}$ be linear integer arithmetic and $\mathcal{R}$ be the class of lists of even length. It is easy to model an execution of a 2-counter machine using a list with integers. Each configuration is represented by two adjacent nodes, which are labeled by the current instruction. The data fields of the two nodes hold the value of the two registers, respectively. The first two nodes are the initial configuration, the last two nodes are a halting configuration, and for any two consecutive configurations, the inbetween instruction is executed correctly. Then a halting computation can be expressed by a STRAND formula:

$$\begin{array}{l}
\exists x_1, x_2.(\texttt{head}(x_1) \wedge \mathit{next}(x_1, x_2) \wedge \mathit{init_conf}(x_1, x_2)) \;\; \wedge \\
\forall z_1, z_2, z_3, z_4.\big(\; (\mathit{odd}(z_1) \wedge \mathit{next}(z_1, z_2) \wedge \mathit{next}(z_2, z_3) \wedge \mathit{next}(z_3, z_4)) \;\; \Rightarrow \\
\qquad\qquad\qquad \mathit{one_step_exec}(z_1, z_2, z_3, z_4) \;\big) \;\; \wedge \\
\exists y_1, y_2.(\texttt{tail}(y_2) \wedge \mathit{next}(y_1, y_2) \wedge \mathit{terminating}(x_1, x_2))
\end{array}$$

Hence the halting problem 2-counter machines reduces to the satisfiability of STRAND. Notice that the satisfiability of the STRAND logic is undecidable, even though the underlying logics $\mathcal{L}$ and $\mathcal{D}$ are decidable. □

### 4.1.1 Two Decidable Fragments

The primary contribution of this chapter is in identifying two decidable fragments of Strand: (1) a semantically defined fragment $\textsc{Strand}_{dec}^{sem}$, and (2) a syntactically defined fragment $\textsc{Strand}_{dec}^{syn}$. Both fragments admit automata-theoretic decision procedures.

The decision procedures for $\textsc{Strand}_{dec}^{sem}$ are based on

a) abstracting the data-predicates in the formula with Boolean variables to obtain a formula purely on graphs;

b) extracting the set of *minimal graphs* according to a *Satisfiability-Preserving Embedding* (SPE) relation that was completely agnostic to the data-logic, and is guaranteed to be minimal for the two fragments; and

c) checking whether any of the minimal models admits a data-extension that satisfies the formula, using a data-logic solver.

As a syntactically defined fragment that is subsumed under $\textsc{Strand}_{dec}^{sem}$, $\textsc{Strand}_{dec}^{syn}$ also admits the above decision procedures. However, we develop a new method to solve satisfiability for $\textsc{Strand}_{dec}^{syn}$ using a notion called *small models*, which are not the precise set of minimal models but a slightly larger class of models.

We also showed that the decidable fragments can be used in the verification of pointer-manipulating programs. We implemented the decision procedures for both $\textsc{Strand}_{dec}^{sem}$ and $\textsc{Strand}_{dec}^{syn}$ using Mona on tree-like data-structures for the graph logic and Z3 for quantifier-free arithmetic, and reported experimental results in Hoare-style deductive verification of certain programs manipulating data-structures.

### 4.1.2 Some Intuitions

We here give some intuition behind the decision procedures presented in this chapter.

**Satisfiability-preserver embeddings:** The crucial notion behind both the two decidable fragments is called *Satisfiability-Preserving Embeddings*

(SPEs). Intuitively, for two heap structures (without data) $S$ and $S'$, $S$ *satisfiability-preservingly embeds* in $S'$ with respect to a STRAND formula $\psi$ if there is an embedding of the nodes of $S$ in $S'$ such that *no matter how the data-logic constraints are interpreted*, if $S'$ satisfies $\psi$, then so will the submodel $S$ satisfy $\psi$, by inheriting the data-values. We define the notion of satisfiability-preserving embeddings so that it is entirely structural in nature, and is definable using MSO on an underlying graph that simultaneously represents $S$, $S'$, and the embedding of $S$ in $S'$.

If $S$ satisfiability-preservingly embeds in $S'$, then clearly, when checking for satisfiability, we can ignore $S'$ if we check satisfiability for $S$. More generally, the satisfiability check can be done only for the *minimal* structures with respect to the partial-order (and well-order) defined by SPEs.

The semantic decidable fragment $\text{STRAND}^{syn}_{dec}$ is defined to be the class of all formulas for which the set of minimal structures with respect to satisfiability-preserving embeddings is *finite*, and where the quantifier-free theory of the underlying data-logic is decidable. Though this fragment of STRAND is semantically defined, we show that it is syntactically checkable. Given a STRAND formula $\psi$, we show that we can build a regular finite representation of all the minimal models with respect to satisfiability-preserving embeddings, even if it is an infinite set, using automata-theory. Then, checking whether the number of minimal models is finite is decidable. If the set of minimal models is finite, we show how to enumerate the models, and reduce the problem of checking whether they admit a data-extension that satisfies $\psi$ to a formula in the *quantifier-free* fragment of the underlying data-logic, which can then be decided.

**The Bernays-Schönfinkel-Ramsey class:** Having motivated formulas with the $\exists^*\forall^*$ quantification in Section 3.2, it is worthwhile to examine this fragment in classical first-order logic (over arbitrary infinite universes), which is known as the Bernays-Schönfinkel-Ramsey class, and is a classical decidable fragment of first-order logic [13].

Consider first a purely relational vocabulary (assume there are no functions and even no constants). Then, given a formula $\exists\vec{x}\forall\vec{y}.\varphi(\vec{x},\vec{y})$, let $M$ be a model that satisfies this formula. Let $v$ be an interpretation for $\vec{x}$ such that $M$ under $v$ satisfies $\forall\vec{y}.\varphi(\vec{x},\vec{y})$. Then it is not hard to argue that the submodel obtained by picking only the elements used in the interpretation of $\vec{x}$ (i.e.

$v(\vec{x})$), and projecting each relation to this smaller set, satisfies the formula $\exists\vec{x}\forall\vec{y}.\varphi(\vec{x},\vec{y})$ as well [13]. Hence a model of size at most $k$ always exists that satisfies $\varphi$, if the formula is satisfiable, where $k$ is the size of the vector of existentially quantified variables $\vec{x}$. This *bounded model property* extends to when constants are present as well (the submodel should include all the constants) but fails when more than two functions are present. Satisfiability hence reduces to propositional satisfiability, and this class is also called the *effectively propositional* class, and SMT solving for this class exists.

When the signature contains a single function symbol, the class is still decidable [13]; since arbitrary projection to a smaller universe does not give rise to a model (as functions have to be defined on every element), the argument is slightly more complex, and creates finite *psuedo-models* of bounded size. When two or more functions are present, satisfiability of the fragment becomes undecidable [13].

The decidable fragment of STRAND is fashioned after a similar but more complex argument. Given a subset of nodes of a model, the subset itself may not form a valid graph/data-structure. We define a notion of *submodels* that allows us to extract proper subgraphs that contain certain nodes of the model. However, the relations (edges) in the submodel will *not* be the projection of edges in the larger model. Consequently, the submodel may not satisfy a formula, even though the larger model does.

We define a notion called *satisfiability-preserving embeddings* that allows us to identify when a submodel $S$ of $T$ is such that, whenever $T$ satisfies $\psi$ under some interpretation of the data-logic, $S$ can *inherit* values from $T$ to satisfy $\psi$ as well. This is considerably more complex and is the main technical contribution of the paper. We then build decision procedures to check the minimal models according to this embedding relation.

**Organization:** We first formally define the SPEs in Section 4.2, then present the two decidable fragments and their decision procedures in Section 4.3 and Section 4.4, respectively. We also report and evaluate an implementation of the above decision procedures with experimental results in Section 4.5. We conclude this chapter with related work in Section 4.6.

## 4.2 Satisfiability-Preserving Embeddings

Given a Strand formula $\exists\vec{x}\forall\vec{y}.\varphi(\vec{x},\vec{y})$ over an RDDS $\mathcal{R} = (\psi_{Tr}, \psi_U, \{\alpha_a\}_{a\in L_v}, \{\beta_b\}_{b\in L_e})$, we can transform this to an *equisatisfiable* formula $\forall\vec{x}\forall\vec{y}.\varphi'(\vec{x},\vec{y})$ over a different RDDS $\mathcal{R}'$, where data-structures in $\mathcal{R}'$ are data-structures in $\mathcal{R}$ with new unary predicates that give a valuation for the variables in $\vec{x}$. $\mathcal{R}'$ can be formally defined as $= (\psi'_{Tr}, \psi'_U, \{\alpha'_a\}_{a\in L'_v}, \{\beta'_b\}_{b\in L'_e})$, where

- $L'_v \;=\; L_v \cup \{a_i \mid x_i \in \vec{x}\}$
- $L'_e \;=\; L_e$
- $\psi'_{Tr} \;\equiv\; \psi_{Tr} \wedge \bigwedge_{x_i\in\vec{x}} \exists z.\big(\psi_U(z) \wedge Q_{a_i}(z) \wedge \forall z'.(Q_{a_i}(z') \Rightarrow z = z')\big)$
- $\psi'_U \;\equiv\; \psi_U$
- $\alpha'_a \;\equiv\; \alpha_a$ for every $a \in L_v$
- $\alpha'_{a_i} \;\equiv\; \texttt{true}$ for every $x_i \in \vec{x}$
- $\beta'_b \;\equiv\; \beta_b$ for every $b \in L_e$

Intuitively, we modify $\psi_{Tr}$ to accept trees with extra labelings $a_i$ that give (an arbitrary) singleton valuation of each $x_i \in \vec{x}$ that satisfies $\psi_U$, and introduce new unary predicates $Val_i(x) = Q_{a_i}(x)$. Then we can define $\varphi'(\vec{x},\vec{y})$ to be

$$\big(\bigwedge_i Val_i(x_i)\big) \Rightarrow \varphi(\vec{x},\vec{y})$$

It is easy to see there is a graph in $Graph(\mathcal{R})$ that satisfies $\exists\vec{x}\forall\vec{y}.\varphi(\vec{x},\vec{y})$ iff there is a graph in $Graph(\mathcal{R}')$ that satisfies $\forall\vec{x}\forall\vec{y}.\varphi'(\vec{x},\vec{y})$. The latter is a Strand formula with no existential quantification of variables whose data is referred to by the formula. Let us refer to these formulas with no leading existential quantification on data-variables as *universal* Strand *formulas*; we will now outline techniques to solve the satisfiability problem of a certain class of universal Strand formulas.

Let $\psi = \forall\vec{y}.\ \varphi(\vec{y})$ be a universal Strand formula.

### 4.2.1 Structural Abstraction

Before defining SPE formally, we first define a notion calls *structural abstraction* of $\psi$. Let $\gamma_1, \gamma_2, \ldots, \gamma_r$ be the *atomic* relational formulas of the

data-logic in $\varphi$. Note that each of these relational formulas will be over the data fields of variables in $\vec{y}$ only (since the data-logic is restricted to working over the terms $data(y)$, where $y \in \vec{y}$).

Consider evaluating $\psi$ over a particular model. After fixing a particular valuation of $\vec{y}$, notice that the data-relations $\gamma_i$ get all fixed, and evaluate to true or false. Moreover, once the values of $\gamma_i$ are fixed, the rest of the formula is purely *structural* in nature. Now, if $\psi$ is to hold in the model, then no matter how we choose to evaluate $\vec{y}$ over the nodes of the model, the $\gamma_i$ relations must evaluate to true or false in such a way that $\varphi$ holds.

Since we want, in the first phase, to *ignore* the data-constraints entirely, we will abstract $\psi$ using a purely structural formula by using Boolean variables $b_1, \ldots b_r$ instead of the data-relations $\gamma_1, \gamma_2, \ldots, \gamma_r$. However, since these Boolean variables get determined only *after* the valuation of $\vec{y}$ gets determined, and since we are solving for satisfiability, we existentially quantify over these Boolean variables and quantify them *after* the quantification of $\vec{y}$. Formally,

**Definition 4.2.1.** *Let $\psi = \forall \vec{y}.\ \varphi(\vec{y})$ be a universal* STRAND *formula, and let the atomic relational formulas of the data-logic that occur in $\varphi$ be $\gamma_1, \gamma_2, \ldots, \gamma_r$. Then its structural abstraction $\widehat{\psi}$ is defined as the pure MSO formula on graphs:*

$$\forall \vec{y} \ \ \exists b_1 \ldots b_r. \ \ \varphi'(\vec{y}, \vec{b})$$

*where $\varphi'$ is $\varphi$ with every occurrence of $\gamma_i$ replaced with $b_i$.*

**Remark:** The definition of structural abstractions can be strengthened in two ways. First, if $\gamma_i$ and $\gamma_j$ are of the same arity and over $\vec{z}$ and $\vec{z'}$, respectively, and further uniformly replacing $z_i$ with $z_i'$ in $\gamma_i$ yields $\gamma'$, then we can express the constraint $((\vec{z_i}{=}\vec{z_i}') \Rightarrow (b_i \Leftrightarrow b_j))$, in the inner formula $\varphi'$. Second, if a constraint $\gamma_i$ involves only existentially quantified variables in $\vec{x}$, then we can move the quantification of $b_i$ outside the universal quantification. Doing these steps gives a more accurate structural abstraction, and in practice, restricts the number of models created. We use these more precise abstractions in the experiments, but use the less precise abstractions in the theoretical narrative. The proofs in this section, however, smoothly extend to the more precise abstractions.

**Example 4.2.2.** *Consider the sortedness formula $\psi_{sorted}$ from Example 3.4.3. Then its structural abstraction can be derived as*

$$\widehat{\psi}_{sorted} \quad \equiv \quad \forall y_1, y_2\ \exists b_1, b_2, b_3.\ \Big(b_1 \wedge b_2 \wedge \big((y_1 \rightarrow^* y_2) \Rightarrow b_3\big)\Big)$$

*Note that each Boolean variable $b_i$ replaces an atomic relational formula $\gamma_i$, which is some data-constraint on the data-fields of some of the quantified variables.*

The structural abstraction has not only lost the constraint, but has even lost the precise variables whose data-fields the constraint was over. Nevertheless, the abstraction is enough to define the notion of satisfiability-preserving embeddings below.

The following proposition is obvious; it says that if a universal STRAND formula $\psi$ is satisfiable, then so is its structural abstraction $\widehat{\psi}$. The proposition is true because the values for the Boolean variables can be set in the structural abstraction precisely according to how the relational formulas $\gamma_i$ evaluate in $\psi$:

**Proposition 4.2.3.** *Let $\psi = \forall \vec{y}.\varphi(\vec{y})$ be a universal* STRAND *formula, and $\widehat{\psi}$ be its structural abstraction. If $\psi$ is satisfiable over an RDDS $\mathcal{R}$, then the MSO formula on graphs (with no constraints on data) $\widehat{\psi}$ is also satisfiable over $\mathcal{R}$.*

*Proof.* Let $\mathcal{R}$ be an RDDS satisfying $\psi = \forall \vec{y}.\varphi(\vec{y})$, then its formula abstraction $\widehat{\psi} = \forall \vec{y} \quad \exists b_1 \ldots b_r.\quad \varphi'(\vec{y}, \vec{b})$ is satisfied by $\mathit{Graph}(\mathcal{R})$. Let the atomic data-logic subformulas appearing in $\psi$ be $\gamma_1, \ldots, \gamma_r$, then for each assignments of $\vec{y}$, simply assign each $b_i$ the same value as $\gamma_i$. Then the abstracted $\varphi'(\vec{y}, \vec{b})$ is obviously satisfied. ∎

### 4.2.2 Formal Definition

We are now ready to define satisfiability-preserving embeddings using structural abstractions. Given a model defined by a tree $T = (V, \lambda)$ satisfying $\psi_{Tr}$, and a valid subset $S \subseteq V$, and a universal STRAND formula $\psi$, we would like to define the notion of when the submodel defined by $S$ satisfiability-preservingly embeds in the model. The most crucial requirement for the

definition is that if $S$ satisfiability-preservingly embeds in $T$, then we require that if there is a data-extension of $\mathit{Graph}(T)$ that satisfies $\psi$, then the nodes of the submodel defined by $S$, $\mathit{Graph}(\mathit{Subtree}(T, S))$, can inherit the data-values and also satisfy $\psi$. The notion of structural abstractions defined above allows us to define such a notion.

Intuitively, if a model satisfies $\psi$, then it would satisfy $\widehat{\psi}$ too, as for every valuation of $\vec{y}$, there is some way it would satisfy the atomic data-relations, and using this we can pull out a valuation for the Boolean variables to satisfy $\widehat{\psi}$ (as in the proof of Proposition 4.2.3 above). Now, since the data-values in the submodel are *inherited* from the larger model, the atomic data-relations would hold *in the same way* as they do in the larger model. However, the submodel may not satisfy $\psi$ if the conditions on the truth- and false-hood of these atomic relations demanded by $\psi$ are not the same.

For instance, consider a list and a sublist of it. Consider a formula that demands that for any two successor elements $y_1, y_2$ in the list, the data-value of $y_2$ is the data-value of $y_1$ incremented by 1 (as in Example 3.2.2):

$$\psi \equiv \forall y_1 \forall y_2. \big( (y_1 \rightarrow y_2) \Rightarrow (d(y_2) = d(y_1) + 1) \big)$$

Now consider two nodes $y_1$ and $y_2$ that are successors in the sublist but not successors in the list. The list hence could satisfy the formula by setting the data-relation $\gamma : d(y_2) = d(y_1) + 1$ to false. Since the sublist inherits the data values, $\gamma$ would be false in the sublist as well, but the sublist will *not* satisfy the formula $\psi$. We hence want to ensure that *no matter how the larger model satisfies the formula* using some valuation of the atomic data-relations, the submodel will be able to satisfy the formula using the *same valuation of the atomic data-relations*. This leads us to the following definition:

**Definition 4.2.4.** *Let $\psi = \forall \vec{y}.\ \varphi(\vec{y})$ be a universal* STRAND *formula, and let its structural abstraction be $\widehat{\psi} = \forall \vec{y}\ \exists \vec{b}.\ \varphi'(\vec{y}, \vec{b})$. Let $T = (V, \lambda)$ be a tree that satisfies $\psi_{Tr}$, and let a submodel be defined by $S \subseteq V$. Then $S$ is said to* satisfiability-preservingly embed *into $T$ wrt $\psi$ if for every possible valuation of $\vec{y}$ over the elements of $S$, and for every possible Boolean valuation of $\vec{b}$, if $\varphi'(\vec{y}, \vec{b})$ holds in the graph defined by $T$ under this valuation, then the submodel defined by $S$, $\mathit{Graph}(\mathit{Subtree}(T, S))$, also satisfies $\varphi'(\vec{y}, \vec{b})$ under the same valuation. Moreover, $S$* strictly satisfiability-preservingly embeds *into $T$ if $S$ satisfiability-preservingly embeds into $T$ and $S \neq T$.* □

The SPE relation can be seen as a *partial order* over trees (a tree $T'$ satisfiability-preservingly embeds into $T$ if there is a subset $S$ of $T$ such that $S$ satisfiability-preservingly embeds into $T$ and $\mathit{Subtree}(T,S)$ is isomorphic to $T'$); it is easy to see that this relation is reflexive, anti-symmetric and transitive.

It is now not hard to see that if $S$ satisfiability-preservingly embeds into $T$ wrt $\psi$, and $\mathit{Graph}(T)$ satisfies $\psi$, then $\mathit{Graph}(\mathit{Subtree}(T,S))$ also necessarily satisfies $\psi$, which is the main theorem we seek.

**Theorem 4.2.5.** *Let $\psi = \forall \vec{y}.\varphi(\vec{y})$ be universal* STRAND *formula. Let $T = (V,\lambda)$ be a tree that satisfies $\psi_{Tr}$, and $S$ be a valid subset of $T$ that satisfiability-preservingly embeds into $T$ wrt $\psi$. Then, if there is a data-extension of $\mathit{Graph}(T)$ that satisfies $\psi$, then there exists a data-extension of*

$$\mathit{Graph}(\mathit{Subtree}(T,S))$$

*that satisfies $\psi$.*

*Proof.* The gist of the proof of the above theorem goes similar to the arguments given above. Consider a data-extension of $\mathit{Graph}(T)$ that satisfies $\psi$. Each node $u$ of $\mathit{Graph}(\mathit{Subtree}(T,S))$ corresponds to a unique node $u'$ in $\mathit{Graph}(T)$ (see above). Define the data-extension of $\mathit{Graph}(\mathit{Subtree}(T,S))$ by assigning the data-value of each node $u$ to the data-value of the corresponding node $u'$ in $\mathit{Graph}(T)$. For any valuation of $\vec{y}$ over $\mathit{Graph}(\mathit{Subtree}(T,S))$, consider the corresponding valuation over $T$. Since the data-extension of $\mathit{Graph}(T)$ satisfies $\varphi$, by Proposition 4.2.3, $\mathit{Graph}(T)$ must satisfy the formula $\varphi$ with its atomic data-predicates replaced by Boolean variables. Since $S$ satisfiability-preservingly embeds in $T$, the same valuation of the Boolean variables also satisfies $\varphi$ with its atomic data-predicates replaced by Boolean variables. Since the data-extension of $\mathit{Graph}(\mathit{Subtree}(T,S))$ is derived by inheriting the data-values from the data-extension of $\mathit{Graph}(T)$, it follows that the data-values of $\vec{y}$ satisfy the same predicates $\gamma_i$ as they did in the data-extension of $\mathit{Graph}(T)$. Hence it follows that $\varphi(\vec{y})$ holds in the extension of $\mathit{Graph}(\mathit{Subtree}(T,S))$ as well. Since this is true for any valuation of $\vec{y}$ over $S$, it follows that the data-extension of $\mathit{Graph}(\mathit{Subtree}(T,S))$ satisfies $\psi$. $\square$

Notice that the above theorem crucially depends on the formula being universal over data-variables. For example, if the formula was of the form

$\forall y_1 \exists y_2 . \gamma(y_1, y_2)$, then we would have no way of knowing *which* nodes are used for $y_2$ in the data-extension of *Graph*($T$) to satisfy the formula. Without knowing the precise meaning of the data-predicates, we would not be able to declare that whenever a data-extension of *Graph*($T$) is satisfiable, a data-extension of a strict submodel $S$ is satisfiable (even over lists).

The above notion of SPEs is the property that will be used to decide if a formula falls into our decidable fragments.

## 4.3 A Semantically Defined Fragment

We are now ready to define $\text{Strand}_{dec}^{sem}$. This fragment is semantically defined (but syntactically checkable, as we show below), and intuitively contains all Strand formulas that have a *finite number of minimal models* with respect to the partial-order defined by satisfiability-preserving embeddings.

Formally, let $\psi = \forall \vec{y}. \varphi(\vec{y})$ be a universal Strand formula, and let $T = (V, \lambda)$ be a tree that satisfies $\psi_{Tr}$. Then we say that $T$ is *a minimal model* with respect to $\psi$ if there is no strict valid subset $S$ of $V$ that satisfiability-preservingly embeds in $T$.

**Definition 4.3.1.** *Let $\mathcal{R}$ be an RDDS.*

*A universal formula $\psi = \forall \vec{y}.\ \varphi(\vec{y})$ is in* $\text{Strand}_{dec}^{sem}$ *iff the number of minimal models with respect to $\mathcal{R}$ and $\psi$ is finite.*

*A* Strand *formula of the form $\psi = \exists \vec{x}\ \forall \vec{y}\ \varphi(\vec{x}, \vec{y})$ is in* $\text{Strand}_{dec}^{syn}$ *iff the corresponding equisatisfiable universal formula $\psi'$ over set of data-structure $\mathcal{R}'$ (as defined in Section 5.1) is in* $\text{Strand}_{dec}^{sem}$. □

### 4.3.1 Checking Membership in $\text{Strand}_{dec}^{syn}$

We now show that we can *effectively check* if a Strand formula belongs to the decidable fragment $\text{Strand}_{dec}^{sem}$. The idea, intuitively, is to express that a model is a minimal model with respect to satisfiability-preserving embeddings, and then check, using automata theory, that the number of minimal models is finite.

Let $\psi = \forall \vec{y}. \varphi(\vec{y})$ be universal Strand formula, and let its structural abstraction be $\widehat{\psi} = \forall \vec{y}\ \exists \vec{b}.\ \varphi'(\vec{y}, \vec{b})$.

$$
\begin{array}{rcl}
interpret(p) & = & p \\
interpret(Q_a(s)) & = & \psi_U(s) \wedge \alpha_a(s), \quad \text{for every } a \in L_v \\
interpret(E_b(s,t)) & = & \psi_U(s) \wedge \psi_U(t) \wedge \beta_b(s,t), \quad \text{for every } b \in L_e \\
interpret(s = t) & = & (\psi_U(s) \wedge \psi_U(t) \wedge s = t) \\
interpret(s \in W) & = & \psi_U(s) \wedge s \in W \\
interpret(\varphi_1 \vee \varphi_2) & = & interpret(\varphi_1) \vee interpret(\varphi_2) \\
interpret(\neg\varphi) & = & \neg(interpret(\varphi)) \\
interpret(\exists s.\varphi) & = & \exists s.(\psi_U(s) \wedge interpret(\varphi)) \\
interpret(\exists W.\varphi) & = & \exists W.((\forall s.(s \in W \Rightarrow \psi_U(s))) \wedge interpret(\varphi))
\end{array}
$$

Figure 4.1: Definition of the *interpret* function

We now show that we can define an MSO formula $MinModel_\psi$, such that it holds on a tree $T = (V, \lambda)$ iff $T$ defines a minimal model with respect to satisfiability-preserving embeddings.

Before we do that, we need some technical results and notation. Let $\mathcal{R} = (\psi_{Tr}, \psi_U, \{\alpha_a\}_{a \in L_v}, \{\beta_b\}_{b \in L_e})$.

We first show that any (pure) MSO formula $\delta$ on $(L_v, L_e)$-labeled graphs can be *interpreted* on trees. Formally, we show that any (pure) MSO formula $\delta$ on $(L_v, L_e)$-labeled graphs can be transformed syntactically to a (pure) MSO formula $interpret(\delta)$ on trees such that for any tree $T = (V, \lambda)$, $Graph(T))$ satisfies $\delta$ iff $T$ satisfies $interpret(\delta)$.

This is not hard to do, since the graph is defined using MSO formulas on the trees, and we can adapt these definitions to work over the tree instead. In fact, this is the reason why MSO on recursive data-structures is decidable: we can translate the formula to trees, and check satisfiability of the transformed formula over trees that satisfy $\psi_{Tr}$. The transformation is given by the following function *interpret*; the predicates for edges, and the predicates that check vertex labels and edges labels are transformed according to their definition, and all quantified variables are restricted to quantify over nodes that satisfy $\psi_U$. The *interpret* function is presented in Figure 4.1.

**Proposition 4.3.2.** *For any RDDS $\mathcal{R}$ and for any formula $\delta$ w.r.t. $\mathcal{R}$, A $(L_v, L_e)$-labeled graphs $Graph(T)$ satisfies $\delta$ iff the $(L_v, L_e)$-labeled $T$ satisfies $interpret(\delta)$.*

*Proof.* Let us fix the $(L_v, L_e)$-labeled tree $T$, and prove the proposition by induction on the structure of $\delta$:

[$\delta \equiv p$ ] $p$ and *interpret*$(p)$ (which is just $p$) are obviously equisatisfiable.

[$\delta \equiv Q_a(s)$ ] By definition, a node $s$ is labeled $Q_a$ in *Graph*$(T)$ if and only if $s$ satisfies both $\psi_U$ and $\alpha_a$ in $T$.

[$\delta \equiv E_b(s,t)$ ] Similar to the above case.

[$\delta \equiv s = t$ ] $s$ and $t$ represent the same node in *Graph*$(T)$ is and only if the node satisfies $\psi_U$ in T, i.e., $\psi_U(s) \wedge \psi_U(t) \wedge s = t$.

[$\delta \equiv s \in W$ ] Similar to the above case.

[$\delta \equiv \varphi_1 \vee \varphi_2$ ] As an inductive hypothesis, assume the proposition is true for both $\varphi_1$ and $\varphi_2$, then *Graph*$(T)$ satisfies $\delta$ iff *Graph*$(T)$ satisfies $\varphi_1$ or $\varphi_2$, by induction, iff $T$ satisfies *interpret*$(\varphi_1)$ or *interpret*$(\varphi_2)$, i.e., $T$ satisfies *interpret*$(\varphi_1) \vee$ *interpret*$(\varphi_2)$.

[$\delta \equiv \neg\varphi$ ] Similar to the above case.

[$\delta \equiv \exists s.\varphi$ ] As an inductive hypothesis, assume the proposition is true on $\varphi$. Now *Graph*$(T)$ satisfies $\delta$ iff there exists a node $s$ in *Graph*$(T)$ s.t. $\varphi(s)$ is true, in other word, by induction, there exists a node $s$ in $T$ satisfying $\psi_U$ s.t. *interpret*$(\varphi(s))$ is true. This is exactly the case that $T$ satisfies *interpret*$(\exists s.\varphi)$.

[$\delta \equiv \exists W.\varphi$ ] Similar to the above case.

□

Now, we give another transformation, that transforms an MSO formula $\delta$ on trees to a formula $tailor_X(\delta)$ on trees, over a free set-variable $X$, such that for any tree $T = (V, \lambda)$ and any valid subset $S \subseteq V$, *Subtree*$(T, S)$ satisfies $\delta$ iff $T$ satisfies $tailor_X(\delta)$ when $X$ is interpreted to be $S$. In other words, we can transform a formula that expresses a property of a subtree to a formula that expresses the same property on the subtree defined by the free variable $X$. The transformation is given by the function *tailor* presented in Figure 4.2; the crucial transformation are the edge-formulas, which has to be interpreted as the edges of the subtree defined by $X$.

The above *tailor* transformation leads to the following proposition:

$$
\begin{array}{rcl}
tailor_X(p) & = & p \\
tailor_X(succ_i(s,t)) & = & s \in X \wedge t \in X \wedge \\
& & \exists s' \quad [succ_i\,(s,s') \wedge \; s' \leq t \;\; \wedge \\
& & \;\; \forall t'.\,((t' \in X \wedge s' \leq t') \Rightarrow t \leq t')]\,, \text{ for every } i \in [k]. \\
tailor_X(s=t) & = & (s \in X \wedge t \in X \wedge s = t) \\
tailor_X(s \in W) & = & s \in W \wedge W \subseteq X \\
tailor_X(\varphi_1 \vee \varphi_2) & = & tailor(\varphi_1) \vee tailor(\varphi_2) \\
tailor_X(\neg\varphi) & = & \neg(tailor(\varphi)) \\
tailor_X(\exists s.\varphi) & = & \exists s.(s \in X \wedge tailor(\varphi)) \\
tailor_X(\exists W.\varphi) & = & \exists W.(W \subseteq X \wedge tailor(\varphi))
\end{array}
$$

Figure 4.2: Definition of the *tailor* function

**Proposition 4.3.3.** *For any MSO sentence $\delta$ on $k$-ary trees, for any tree $T = (V, \lambda)$ and for any valid subset $S \subseteq V$, Subtree$(T, S)$ satisfies $\delta$ iff $T$ satisfies $tailor_X(\delta)$ when $X$ is interpreted to be $S$.*

*Proof.* Let us fix the tree $T$, and the valid subset $S$, and prove the proposition by induction on the structure of $\delta$:

[$\delta \equiv p$ ] Obvious as $p$ and *tailor*$(p)$ are identical.

[$\delta \equiv \boldsymbol{succ}_i(s,t)$ ] If *Subtree*$(T, S)$ satisfies $succ_i(s,t)$, then both $s$ and $t$ are in $S$, moreover, $t$ is the $i$-th successor of $s$ in the subtree, then by definition, $t$ is the closest descendant of $s'$, the $i$-th successor of $s$ in $T$. All these properties are captured by the formula $tailor_X(succ_i(s,t))$ in $T$. For the reverse direction, when $T$ satisfies $tailor_X(succ_i(s,t))$, both $s$ and $t$ are in $S$ and $t$ is the $i$-th successor of $s$ in *Subtree*$(T, S)$.

[$\delta \equiv s = t$ ] When both $s$ and $t$ are in $X$, $s = t$ and *tailor*$(s = t)$ are equi-satisfiable.

[$\delta \equiv s \in W$ ] Similar to the above case.

[$\delta \equiv \varphi_1 \vee \varphi_2$ ] As an inductive hypothesis, assume the proposition is true for both $\varphi_1$ and $\varphi_2$, then *Subtree*$(T, S)$ satisfies $\delta$ iff *Subtree*$(T, S)$ satisfies $\varphi_1$ or $\varphi_2$, by induction, iff $T$ satisfies $tailor_X(\varphi_1)$ or $tailor_X(\varphi_2)$, i.e., $T$ satisfies $tailor_X(\varphi_1) \vee tailor_X(\varphi_2)$.

[$\delta \equiv \neg\varphi$ ] Similar to the above case.

[$\delta \equiv \exists s.\varphi$ ] Similar to the above case.

[$\delta \equiv \exists W.\varphi$ ] Similar to the above case.

$\square$

Note that the above transformations can be combined. For any MSO formula $\delta$ on $(L_v, L_e)$ labeled graphs, consider the formula $\mathit{tailor}_X(\mathit{interpret}(\delta))$. For any tree $T = (V, \lambda)$ and for any valid subset $S \subseteq V$, $\mathit{Graph}(\mathit{Subtree}(T, S))$ satisfies $\delta$ iff $T$ satisfies $\mathit{tailor}_X(\mathit{interpret}(\delta))$, where $X$ is interpreted as $S$.

### 4.3.2 Expressing Minimal Models in MSO

Remember that we can express with an MSO formula $\mathit{ValidSubModel}(X)$, with a free set variable $X$, that holds in a tree $T = (V, \lambda)$ iff $X$ is interpreted as a valid submodel of $T$ This is easy; we express the properties of $X$ being LCA-closed, and also check that the subtree defined by $X$ satisfies $\psi_{Tr}$:

$$\begin{aligned}
&\mathit{ValidSubModel}(X) \equiv \\
&\forall s, t, u\,((s \in X \wedge t \in X \wedge \mathit{lca}(s,t,u)) \Rightarrow u \in X) \wedge \mathit{tailor}_X(\psi_{Tr}) \\
&\qquad\qquad \wedge (\forall s (s \in X \wedge \mathit{tailor}_X(\psi_U(s))) \quad \Rightarrow \quad \psi_U(s))
\end{aligned}$$

where $\mathit{lca}(s, t, u)$ is an MSO formula that checks whether $u$ is the LCA of $s$ and $t$ in the tree; this expresses the requirements in Definition 3.3.4.

Now we can also define the MSO formula on $k$-ary trees $\mathit{MinModel}_\psi$ that captures minimal models. Let $\widehat{\psi} = \forall \vec{y}\, \exists \vec{b}\, \varphi'(\vec{y}, \vec{b})$ be the structural abstraction of $\psi$, then

$$\begin{aligned}
\mathit{MinModel}_\psi \equiv \neg\exists X.\Big[\ & \mathit{ValidSubset}(X) \ \wedge\ \exists s.(s \in X) \wedge \exists s.(s \notin X)\ \wedge \\
& \forall \vec{y}\ \forall \vec{p}.\ \Big(\Big(\wedge_{y \in \vec{y}} \big(y \in X \wedge \psi_U(y)\big) \wedge \mathit{interpret}(\widehat{\varphi}(\vec{y}, \vec{p}))\Big) \\
& \qquad \Rightarrow \mathit{tailor}_X\big(\mathit{interpret}(\widehat{\varphi}(\vec{y}, \vec{p}))\big)\Big)\ \Big]
\end{aligned}$$

The above formula when interpreted on a tree $T$ says that there does not exists a set $X$ that defines a non-empty valid strict subset of the nodes of $T$,

which defines a model $\mathit{Graph}(\mathit{Subtree}(T,X))$ that further satisfies the following: for every valuation of $\vec{y}$ over the nodes of $\mathit{Graph}(\mathit{Subtree}(T,S))$ and for every valuation of the Boolean variables $\vec{b}$ such that the structural abstraction of $\varphi$ holds in $\mathit{Graph}(T)$, the same valuation also makes the structural abstraction of $\varphi$ hold in $\mathit{Graph}(\mathit{Subtree}(T,S))$.

**Theorem 4.3.4.** *Given a sentence $\exists\vec{x}\forall\vec{y}.\ \varphi(\vec{x},\vec{y})$, the membership problem of the fragment* $\textsc{Strand}^{sem}_{dec}$ *is decidable.*

*Proof.* Note that $\mathit{MinModel}_\psi$ is a *pure* MSO formula on trees, and encodes the properties required of a minimal model with respect to satisfiability-preserving embeddings. Using the classical logic-automaton connection [13], we can transform the MSO formula $\mathit{MinModel}_\psi \wedge \psi_{Tr} \wedge \widehat{\psi}$ to a tree automaton that accepts precisely those trees that define data-structures that satisfy the structural abstraction and are minimal models. Since the finiteness of the language accepted by a tree automaton is decidable, we can check whether there are only a finite number of minimal models w.r.t. SPEs, and hence decide membership in the decidable fragment $\textsc{Strand}^{sem}_{dec}$. $\square$

### 4.3.3 Deciding formulas in $\textsc{Strand}^{sem}_{dec}$

We now give the decision procedure for satisfiability of $\textsc{Strand}^{sem}_{dec}$ formulas over an RDDS. First, we transform the satisfiability problem to that of satisfiability of universal formulas of the form $\psi = \forall\vec{y}.\ \varphi(\vec{y})$. Then, using the formula $\mathit{MinModel}_\psi$ described above, and by transforming it to tree automata, we extract the set of *all* trees accepted by the tree-automaton in order to get the tree-representation of all the minimal models. Note that this set of minimal models is finite, and the sentence is satisfiable iff it is satisfiable in some data-extension of one of these models.

We can now write a quantifier-free formula over the data-logic that asserts that one of the minimal models has a data-extension that satisfies $\psi$. This formula will be a disjunction of $m$ sub-formulas $\eta_1,\ldots,\eta_m$, where $m$ is the number of minimal models. Each formula $\eta_i$ will express that there is a data-extension of the $i$'th minimal model that satisfies $\psi$. First, since a minimal model has only a finite number of nodes, we create one data-variable for each of these nodes, and associate them with the nodes of the model. It is now not hard to transform the formula $\psi$ to this model using no quantification.

The universal quantification over $\vec{y}$ translates to a conjunction of formulas over all possible valuations of $\vec{y}$ over the nodes of the fixed model. Existential (universal) quantified variables are then "expanded" using disjunction (conjunction, respectively) of formulas for all possible valuations over the fixed model. The edge-relations between nodes in the model are interpreted on the tree using MSO formulas in $\mathcal{R}$, which are then expanded to conditions over the fixed set of nodes in the model. Finally, the data-constraints in the STRAND formula are directly written as constraints in the data-logic.

The resulting formula is a pure data-logic formula without quantification that is satisfiable if and only if $\psi$ is satisfiable over $\mathcal{R}$. This is then decided using the decision procedure for the data-logic.

**Theorem 4.3.5.** *Given a sentence $\exists\vec{x}\forall\vec{y}.\ \varphi(\vec{x},\vec{y})$ over $\mathcal{R}$ in* STRAND$_{dec}^{sem}$*, the problem of checking whether $\psi$ is satisfiable reduces to the satisfiability of a quantifier-free formula in the data-logic. Since the quantifier-free data-logic is decidable, the satisfiability of* STRAND$_{dec}^{sem}$ *formulas is decidable.*

*Proof.* The decidability has been shown by giving the decision procedure described above. Every step in the procedure can be finished algorithmically. □

## 4.4 A Syntactically Defined Fragment

The bottleneck in the decision procedure for STRAND$_{dec}^{sem}$ is the phase that computes the set of all minimal models. This is done using monadic second-order logic (MSO) on trees, and is achieved by a *complex* MSO formula that has quantifier alternations and also includes adaptations of the STRAND formula *twice* within it. In experiments, this phase is clearly the bottleneck (for example, a verification condition for binary search trees takes about 30s while the time spent by Z3 on the minimal models is less than 0.5s).

we define another decidable fragment STRAND$_{dec}^{syn}$, which is also a fragment of STRAND$_{dec}^{sem}$, using the notion of elastic relations. Hence $Strand_{dec}^{syn}$ admits the same decision procedure for STRAND$_{dec}^{sem}$. Given an RDDS $\mathcal{R}$ and a first-order theory $\mathcal{D}$, the syntax of STRAND$_{dec}^{syn}$ is presented in Figure 4.3. We let $L_e^E$ denote those edge labels $b$ such that $E_b$ is elastic, and $L_e^{NE}$ denote other non-elastic labels. Intuitively, STRAND$_{dec}^{syn}$ formulas are of the kind

$\exists\vec{x}\forall\vec{y}.\varphi(\vec{x},\vec{y})$, where $\varphi$ is *quantifier-free* and combines both data-constraints and structural constraints, with the important restriction that *the atomic relations involving universally quantified variables be only elastic relations.*

$$
\begin{array}{llcl}
\text{Formula} & \psi & ::= & \exists x.\psi \mid \omega \\
\forall\text{Formula} & \omega & ::= & \forall y.\omega \mid \varphi \\
\text{QFFormula} & \varphi & ::= & \gamma(e_1,\ldots,e_n) \mid Q_a(v) \mid E_b(v,v') \mid E_{b'}(x,x') \\
 & & & \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \\
\text{Expression} & e & ::= & \mathit{df}(x) \mid \mathit{df}(y) \mid c \mid g(e_1,\ldots,e_n)
\end{array}
$$

$$
\begin{array}{llcl}
\exists\text{DVar} & x & \in & \mathit{Loc} \\
\forall\text{DVar} & y & \in & \mathit{Loc} \\
\text{Var} & v & ::= & x \mid y \\
\text{Constant} & c & \in & \mathit{Sig}(\mathcal{D}) \\
\text{Function} & g & \in & \mathit{Sig}(\mathcal{D}) \\
\mathcal{D}\text{-Relation} & \gamma & \in & \mathit{Sig}(\mathcal{D}) \\
\text{E-Relation} & b & \in & L_e^E \\
\text{NE-Relation} & b' & \in & L_e^{NE} \\
\text{Predicate} & a & \in & L_v \\
\text{DataField} & \mathit{df} & \in & \mathit{DF}
\end{array}
$$

Figure 4.3: Syntax of $\textsc{Strand}_{dec}^{syn}$

More importantly, we also present a new method to solve satisfiability for $\textsc{Strand}_{dec}^{syn}$ using a notion called *small models*, which are not the precise set of minimal models but a slightly larger class of models. We show that the set of small models is always bounded, and also equisatisfiable (i.e. if there is any model that satisfies the $\textsc{Strand}_{dec}^{syn}$ formula, then there is a data-extension of a small model that satisfies it). The salient feature of small models is that it can be expressed by a much simpler MSO formula that is *completely independent of the* $\textsc{Strand}_{dec}^{syn}$ *formula*! The definition of small models depends only on the *signature* of the formula (in particular, the set of variables existentially quantified). Consequently, it does not mention any structural abstractions of the formula, and is much simpler to solve. This

formulation of decidability is also a theoretical contribution, as it gives a much simpler alternative proof that the logic $\text{STRAND}_{dec}^{syn}$ is decidable.

### 4.4.1 Proof of Decidability

We here give another proof of decidability for $\text{STRAND}_{dec}^{syn}$, which leads to a more efficient decision procedure. Recall that decision procedure for $\text{STRAND}_{dec}^{syn}$ presented in Section 4.3 works as follows. Given a $\text{STRAND}_{dec}^{syn}$ formula $\psi$ over a class of recursively defined data-structures $\mathcal{R}$, we first construct a pure MSO formula on $k$-ary trees $\mathit{MinModel}_\psi$ that captures the subset of trees that are minimal with respect to an equi-satisfiability preserving embedding relation. This assures that if the formula $\psi$ is satisfiable, then it is satisfiable by a data-extension of a minimal model (a minimal model is a model satisfying $\mathit{MinModel}_\psi$). Furthermore, this set of minimal models was guaranteed to be finite. The decision procedure is then to do a simple analysis on the tree automaton accepting all minimal models, to determine the maximum height $h$ of all minimal trees, and then query the data-solver as to whether any tree of height bounded by $h$ satisfies the $\text{STRAND}_{dec}^{syn}$ formula.

In the new decision procedure, we replace the notion of minimal models with a new notion called *small models*. Given a $\text{STRAND}_{dec}^{syn}$ formula $\psi = \exists\vec{x}\forall\vec{y}.\varphi(\vec{x},\vec{y})$ over a class of recursively defined data-structures $\mathcal{R} = (\psi_{Tr}, \psi_U, \{\alpha_a\}_{a\in L_v}, \{\beta_b\}_{b\in L_e})$, the MSO formula $\mathit{SmallModel}(\vec{x})$ is defined on $k$-ary trees, with free variables $\vec{x}$. Intuitively, $\mathit{SmallModel}(\vec{x})$ says that there does not exist a nontrivial valid subset $X$ such that $X$ contains $\vec{x}$, and further satisfies the following: for every non-elastic relation possibly appearing in $\varphi(\vec{x},\vec{y})$, it holds in $\mathit{Graph}(T)$ iff it holds in $\mathit{Graph}(\mathit{Subtree}(T,X))$. Since the evaluations of other atomic formulas, including elastic relations and data-logic relations, are all preserved, we can prove that $\mathit{SmallModel}(\vec{x})$ is equi-satisfiable to the structural constraints in $\psi$, but has only a finite number of

models. The formula $\mathit{SmallModel}(\vec{x})$ is defined as follows:

$$
\begin{aligned}
\mathit{SmallModel}(\vec{x}) \quad \equiv \quad & \psi_{Tr} \ \wedge \ \bigwedge_{x \in \vec{x}} \psi_U(x) \\
\wedge \quad & \neg \exists X. \Big( \ \mathit{ValidSubset}(X) \ \wedge \ \bigwedge_{x \in \vec{x}} (x \in X) \ \wedge \\
& \qquad \exists s.(s \in X) \ \wedge \ \exists s.(s \notin X) \ \wedge \\
& \qquad \bigwedge_{b \in L_e^{NE}, x, x' \in \vec{x}} \Big( \beta_b(x, x') \Leftrightarrow \mathit{tailor}_X\big(\beta_b(x, x')\big) \Big) \Big)
\end{aligned}
$$

Note that the above formula *does not depend* on the STRAND formula $\psi$ at all, except for the set of existentially quantified variables $\vec{x}$.

Our proof strategy is now as follows. We show two technical results:

**(a)** For any $\vec{x}$, $\mathit{SmallModel}(\vec{x})$ has only finitely many models (Theorem 4.4.1). This result is independent of the fact that we are dealing with STRAND formulas.

**(b)** A STRAND formula $\psi$ with existentially quantified variables $\vec{x}$ has a model iff there is some data-extension of a model satisfying $\mathit{SmallModel}(\vec{x})$ that satisfies $\psi$ (Theorem 4.4.2).

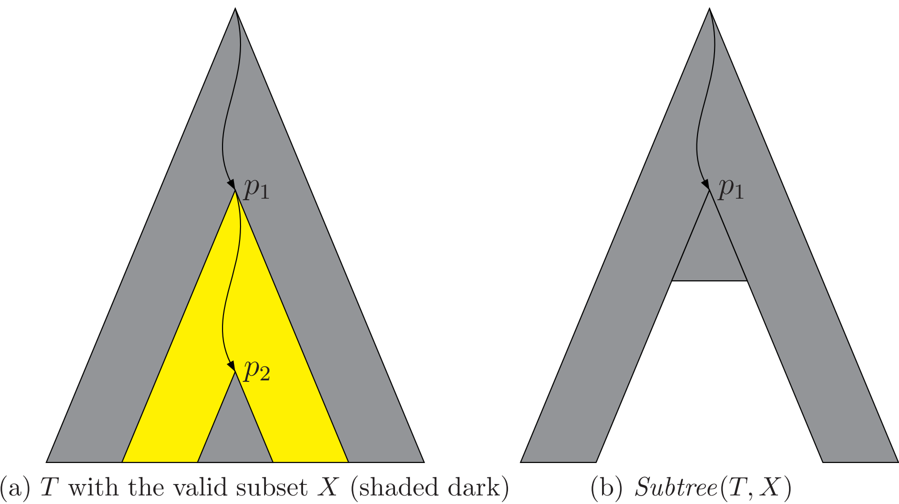
The above two establish the correctness of the decision procedure. Given a $\text{STRAND}_{dec}^{syn}$ formula $\psi$, with existential quantification over $\vec{x}$, we can compute a tree-automaton accepting the set of all small models (i.e. the models of $\mathit{SmallModel}(\vec{x})$), compute the maximum height $h$ of the small models, and then query the data-solver as to whether there is a model of height at most $h$ *with data* that satisfies $\psi$.

We prove the two technical results as below.

**Theorem 4.4.1.** *For any recursively defined data-structure $\mathcal{R}$ and any finite set of variables $\vec{x}$, the number of models of $\mathit{SmallModel}(\vec{x})$ is finite.*

*Proof.* Fix a recursively defined data-structure $\mathcal{R}$ and a finite set of variables $\vec{x}$. It is sufficient to show that for any model $T$ of $\mathit{SmallModel}(\vec{x})$, the size of $T$ is bounded.

We first split $\mathit{SmallModel}(\vec{x})$ into two parts: let $\zeta$ be the first two conjuncts, i.e., $\psi_{Tr} \ \wedge \ \bigwedge_{x \in \vec{x}} \psi_U(x)$, and $\eta$ be the last conjunct.



(a) $T$ with the valid subset $X$ (shaded dark) (b) $Subtree(T, X)$

Figure 4.4: A valid subset $X$ that falsifies $\beta$

Recall the classic logic-automata connection: for any MSO formula $\theta(\vec{y}, \vec{Y})$ with free first-order variables $\vec{y}$ and free set-variables $\vec{Y}$, we can construct a tree-automaton that precisely accepts those trees with encodings of the valuation of $\vec{y}$ and $\vec{Y}$ as extra labels that satisfy the formula $\theta$ [75].

Construct a deterministic (bottom-up) tree automaton $A_\zeta$ that accepts precisely the models satisfying $\zeta(\vec{x})$, using this classic logic-automata connection [75]. Also, for each *non-elastic* edge label $b \in L_r^{NE}$, and each pair of variables $x, x' \in \vec{x}$, let $A_{b,x,x'}$ be a deterministic (bottom-up) tree automaton that accepts the models of the formula $\beta_b(x, x')$.

It is clear that $T$ is accepted by $A_\zeta$, while $A_{b,x,x'}$, for each $b, x, x'$, either accepts or rejects $T$. Construct the *product* of the automaton $A_\zeta$ and all automata $A_{b,x,x'}$, for each $b, x, x'$, with the acceptance condition derived solely from $A_\zeta$; call this automaton $B$; then $B$ accepts $T$.

If the accepted run of $B$ on $T$ is $r$, then we claim that $r$ is *loop-free* (a run of a tree automaton is loop-free if for any path of the tree, there are no two nodes in the path labeled by the same state). Assume not. Then there must be two different nodes $p_1, p_2$ such that $p_2$ is in the subtree of $p_1$, and both $p_1$ and $p_2$ are labeled by the same state $q$ in $r$. Then we can *pump down* $T$ by merging $p_1$ and $p_2$. The resulting tree is accepted by $A_T$ as well. Consider the subset $X$ of $T$ that consists of those remaining nodes, as shown in Figure 4.4. It is not hard to see $X$ is a nontrivial valid subset of $T$. Also for each $b \in L_r^{NE}$ and each $x, x' \in \vec{x}$, since the run of $A_{b,x,x'}$ ends up in the

same state on reading the subtree corresponding to $X$, $\beta_b(x, x')$ holds in $T$ iff $\beta_b(x, x')$ holds in $\mathit{Subtree}(T, X)$. Thus $X$ is a valid subset of $T$ that acts to falsify $\eta$, which contradicts our assumption that $T$ satisfies $\zeta \wedge \eta$.

Since $r$ is loop-free, the height of $T$ is bounded by the number of states in $B$. ☐

We now show that the small models define an adequate set of models to check for satisfiability.

**Theorem 4.4.2.** *Let $\mathcal{R}$ be a recursively defined data-structure and let $\psi = \exists\vec{x}\forall\vec{y}.\varphi(\vec{x}, \vec{y})$ be a* STRAND$^{syn}_{dec}$ *formula. If $\psi$ is satisfiable, then there is a model $\mathcal{M}$ of $\psi$ and a model $T$ of $\mathit{SmallModel}(\vec{x})$ such that $\mathit{Graph}(T)$ is isomorphic to the graph structure of $\mathcal{M}$.*

*Proof.* Let $\psi$ be satisfiable and let $\mathcal{M}$ satisfy $\psi$. Then there is an assignment $I$ of $\vec{x}$ over the nodes of $\mathcal{M}$ under which $\forall\vec{y}\varphi(\vec{x}, \vec{y})$ is satisfied.

Let $T$ be the backbone tree of the graph model of $\mathcal{M}$, and further let us add an extra label over $T$ to denote the assignment $I$ to $\vec{x}$.

Let us, without loss of generality, assume that $T$ is a *minimal* tree; i.e. $T$ has the least number of nodes among all models satisfying $\psi$.

We claim that $T$ satisfies $\mathit{SmallModel}(\vec{x})$ under the interpretation $I$.

Assume not, i.e., $T$ does not satisfy $\mathit{SmallModel}(\vec{x})$ under $I$. Since $T$ under $I$ satisfies $\zeta$, it must not satisfy $\eta$. Hence there exists a strict valid subset of nodes, $X$, such that every non-elastic relation holds over every pair of variables in $\vec{x}$ the same way in $T$ as it does on the subtree defined by $X$.

Let $\mathcal{M}'$ be the model obtained by taking the graph of the subtree defined by $X$ as the underlying graph, with data at each obtained node inherited from the corresponding node in $\mathcal{M}$. We claim that $\mathcal{M}'$ satisfies $\psi$ as well, and since the tree corresponding to $\mathcal{M}'$ is a strict subtree of $T$, this contradicts our assumption on $T$.

We now show that the graph of the subtree defined by $X$ has a data-extension that satisfies $\psi$.

In order to satisfy $\psi$, we take the interpretation of each $x \in \vec{x}$ to be the node in $\mathcal{M}'$ corresponding to $I(x)$. Now consider any valuation of $\vec{y}$. We will show that every *atomic* relation in $\varphi$ holds in $\mathcal{M}$ in precisely *the same way* as it does on $\mathcal{M}'$; this will show that $\varphi$ holds in $\mathcal{M}$ iff $\varphi$ holds in $\mathcal{M}'$, and hence that $\varphi$ holds in $\mathcal{M}'$.

By definition, an atomic relation $\tau$ could be a unary predicate, an elastic binary relation, a non-elastic binary relation, or an atomic data-relation. If $\tau$ is a unary predicate $Q_a(v)$ (where $v \in \vec{x} \cup \vec{y}$), then by definition of submodels (and valid subsets), $\tau$ holds in $\mathcal{M}'$ iff $\tau$ holds in $\mathcal{M}$. If $\tau$ is an elastic relation $E_b(v_1, v_2)$, by definition of elasticity, $\tau$ holds in $\mathcal{M}$ iff $\beta_b(v_1, v_2)$ holds in $T$ iff $\beta_b(v_1, v_2)$ holds in $\mathit{Subtree}(T, X)$ iff $\tau(v_1, v_2)$ holds in $\mathcal{M}'$. If $\tau$ is a non-elastic relation, it must be of form $E_b(x, x')$ where $x, x' \in \vec{x}$. By the properties of $X$ established above, it follows that $E_b(x, x')$ holds in $\mathcal{M}'$ iff $E_b(x, x')$ holds in $\mathcal{M}$. Finally, if $\tau$ is an atomic data-relation, since the data-extension of $\mathcal{M}'$ is inherited from $\mathcal{M}$, the data-relation holds in $\mathcal{M}'$ iff it holds in $\mathcal{M}$.

The contradiction shows that $\mathcal{M}$ is a small model. $\square$

By Theorem 4.4.1, there must be a bound tree $T_B$ such that each models of $\mathit{SmallModel}(\vec{x})$ is a prefix of $T_B$. Such a $T_B$ is not hard to obtain by forgetting the labels of the automaton $A$ in the proof of Theorem 4.4.1. Then we can transform $\psi$ to a quantifier-free data-logic formula $\psi_{T_B}$ by unrolling each universal (existential) quantification over the nodes of $T_b$. By Theorem 4.4.2, $\psi$ and $\psi_{T_B}$ are equisatisfiable.

### 4.4.2 Comparison with $\textsc{Strand}_{dec}^{sem}$

We now compare the new decision procedure, technically, with the previous decision procedure for $\textsc{Strand}_{dec}^{syn}$, which was also the decision procedure for the semantic decidable fragment $\textsc{Strand}_{dec}^{sem}$.

Recall the decision procedure for $\textsc{Strand}_{dec}^{sem}$. Given a $\textsc{Strand}$ formula, we first eliminate the leading existential quantification, by absorbing it into the signature, using new constants. Then, for the formula $\forall \vec{y}.\ \varphi$, we define a *structural abstraction* of $\varphi$, named $\widehat{\varphi}$, where all data-predicates are replaced uniformly by a set of Boolean variables $\vec{p}$. A model that satisfies $\widehat{\varphi}$, for every valuation of $\vec{y}$, using some valuation of $\vec{p}$ is said to be a *minimal model* if it has no proper submodel that satisfies $\widehat{\varphi}$ under the *same* valuation $\vec{p}$, for every valuation of $\vec{y}$. Intuitively, this ensures that if the model can be populated with data in some way so that $\forall \vec{y} \varphi$ is satisfied, then the submodel satisfy the formula $\forall \vec{y} \varphi$ as well, by inheriting the same data-values from the model.

It turns out that for any $\textsc{Strand}_{dec}^{sem}$ formula, the number of minimal models (with respect to the submodel relation) is *finite*. Moreover, we can capture

the class of all minimal models using an MSO formula of the following form:

$$\begin{aligned} \mathit{MinModel} = \neg\exists X. \Big[\ & \mathit{ValidSubset}(X)\ \wedge\ \exists s.(s \in X) \wedge \exists s.(s \notin X) \wedge \\ & \forall \vec{y}\ \forall \vec{p}\ \Big(\Big(\ \wedge_{y \in \vec{y}} \big(y \in X \wedge \psi_U(y)\big) \wedge \mathit{interpret}(\widehat{\varphi}(\vec{y}, \vec{p}))\Big) \\ & \Rightarrow \mathit{tailor}_X\big(\mathit{interpret}(\widehat{\varphi}(\vec{y}, \vec{p}))\big)\Big)\ \Big] \end{aligned}$$

The above formula intuitively says that a model is minimal if there is no valid non-trivial submodel such that for all possible valuations of $\vec{y}$ in the submodel, and all possible valuations of $\vec{p}$, the model satisfies the structural abstraction $\widehat{\varphi}(\vec{y}, \vec{p})$, then so does the submodel.

We can enumerate all the minimal models (all models satisfying the above formula), and using a data-constraint solver, ask whether any of them can be extended with data to satisfy the Strand formula. Most importantly, if none of them can, we know that the formula is unsatisfiable (for if there was a model that satisfies the formula, then one of the minimal models will be a submodel that can inherit the data values and satisfy the formula).

Comparing the formula *MinModel* to the formula *SmallModel*, notice that the latter is incredibly simpler as it does not refer to the Strand formula (i.e. $\varphi$) at all! The *SmallModel* formula just depends on the *set of existentially quantified variables* and the non-elastic relations in the signature. In contrast, the formula above for *MinModel* uses adaptations of the Strand formula $\varphi$ *twice* within it. In practice, this results in a very complex formula, as the verification conditions get long and involved, and this results in poor performance by the MSO solver. In contrast, as we show in the next section, the new procedure results in simpler formulas that get evaluated significantly faster in practice.

The formula is hence much easier to evaluate, using a tool like Mona.

### 4.4.3 Computing the bound more efficiently

The above comparison shows that the formula $\mathit{SmallModel}(\vec{x})$ only relies on the RDDS and the number of existential variables in the $\text{Strand}^{syn}_{dec}$ formula to solve, hence is easier to solve. To make the structure-solving phase even more efficient, when an RDDS $\mathcal{R}$ is given, we can even *pre-compute* the

distance bounds for $\mathcal{R}$ and for each non-elastic $b$, then compute the combined bound on sizes of the structural models *analytically* when the number of existentially quantified variables is given. We first define the distance bound:

**Definition 4.4.3** (Distance bound)**.** *Let* $\mathcal{R} = (\psi_{Tr}, \psi_U, \{\alpha_a\}_{a\in L_v}, \{\beta_b\}_{b\in L_e})$ *be an RDDS. Then a distance bound of* $\mathcal{R}$ *is an integer* $k$ *such that: for every tree* $T$ *labeled by* $x_1$ *and* $x_2$*, if* $x_2$ *is a descendent of* $x_1$ *of distance* $d$ *and* $d > k$*, then there exist two nodes* $y_1, y_2$ *on the path from* $x_1$ *to* $x_2$ *(excluding* $x_1$*), such that replacing the subtree of* $y_1$ *with the subtree of* $y_2$ *forms a tree* $T'$ *that preserves the evaluation of* $\psi_{Tr}$*, and every predicate* $\alpha_a(x_1)$*,* $\alpha_a(x_2)$*, and every relation* $\beta_b(x_1, x_2)$*.*

**Remark:** The length bound of $\mathcal{R}$ only depends on its predicate and non-elastic relations. Consider adding an elastic relation $b$ into $\mathcal{R}$, it is not hard to see that its distance bound is not affected, since for whichever $x_1, x_2$-labeled $T$ and whichever contracted $T'$, $T'$ is a valid subset of $T$ and preserves the evaluation of $\beta_b(x_1, x_2)$ (by the elasticity).

When a distance bound is known, we can analytically compute the size bound of the structural model with respect to a given number of existentially quantified variables.

**Proposition 4.4.4.** *Given an RDDS* $\mathcal{R}$ *with a distance bound* $B$ *and a* $\text{STRAND}^{syn}_{dec}$ *formula* $\psi = \exists\vec{x}\forall\vec{y}.\varphi(\vec{x}, \vec{y})$*, then* $\psi$ *is satisfiable if and only if it is satisfied by a model with the structural size not larger than* $B \cdot (|\vec{x}| + 1)$*.*

*Proof.* We only need to prove the direction from left to right. Consider a satisfying model with a large enough underlying tree. We label the tree with the satisfying assignment of $\vec{x}$ for $\psi$, and also label extra nodes to make the set LCA-closed. It is not hard to prove that at most $2 \cdot |\vec{x}| - 1$ nodes are labeled, and every path from the root to a leaf contains at most $|\vec{x}|$ labeled nodes. Now for each path from the root to a leaf, for each segment between two labeled nodes, if the distance is greater than $B$, one can always contract the tree and obtain a smaller model. Repeating this procedure results in a tree model such that every two labeled nodes are of distance at most $B$, hence the height of the tree is at most $B \cdot (|\vec{x}| + 1)$. Then a model satisfying $\psi$ can be obtained by populating the same data values to the small tree model. □

The distance bound of $b$ can always be over-approximated by the tree automaton that characterizes each predicate $a$ and each relation $b$.

**Proposition 4.4.5.** *Let $\mathcal{R} = (\psi_{Tr}, \psi_U, \{\alpha_a\}_{a \in L_v}, \{\beta_b\}_{b \in L_e})$ be an RDDS. Let $q_t$ be the number of states for the tree automaton characterizing $\psi_{Tr}$, and similarly, let $q_a$ and $q_b$ be the number of states for the tree automaton characterizing each predicate $a$ and each relation $b$, respectively. Then there is a distance bound of $\mathcal{R}$:*

$$B_{\mathcal{R}} \;=\; q_t \cdot \Big(\prod_{a \in L_v} q_a\Big) \cdot \Big(\prod_{b \in L_e^{NE}} q_b\Big)$$

*Proof.* Consider a tree $T$ labeled by $x_1$ and $x_2$, if $x_2$ is a descendent of $x_1$ of distance $d$ such that $d$ is greater than $B_{\mathcal{R}}$. Then consider the run of the product of the tree automata characterizing $\psi_{Tr}$, each $\alpha_a$ and each non-elastic $\beta_b$. Since $d$ is greater than the number of states of the product automaton, there must be two nodes $y_1$ and $y_2$ on the path from $x_1$ to $x_2$ (excluding $x_1$) that are labeled the same state. Then replacing the subtree of $y_1$ with the subtree of $y_2$ results another tree $T'$ that satisfies every formula if and only if the original $T$ does so. As explained above, elastic relations are always preserved in $T'$. Hence $B_{\mathcal{R}}$ is a distance bound by the definition. □

**Example 4.4.6.** *Consider the simple RDDS $\mathcal{R}_{bt}$ for binary trees: both $\psi_{Tr}$ and $\psi_U$ are true; there are one unary predicate (root), two elastic relations (left-desc and right-desc) and two non-elastic relations (left-child and right-child), all defined in MSO. It is not hard to convert the MSO definitions for root to a tree automaton comprises of up to* 3 *states, and left-child and right-child to tree automata comprises up to* 6 *states. Then by Proposition 4.4.5, we can analytically compute a distance bound $B_{bt} = 3 \cdot 6 \cdot 6 = 108$. Now for arbitrary* $\textsc{Strand}_{dec}^{syn}$ *formula $\psi$ defined over $\mathcal{R}_{bt}$ with $k$ existential variables, by Proposition 4.4.4, the bound of the tree height of a satisfying model is* $108 \cdot (k-1)$.

The above example shows that when an RDDS is fixed, the bound of the sizes of the structural models can be computed analytically, and *completely* avoid the structural phase altogether, without the use of a solver. While the bound is much larger than necessary, We can even establish smaller bounds by analyzing the distance bound manually, for some data-structures.

For instance, consider Example 4.4.6 again, we claim that the minimum distance bound is just 2. For every tree $T$, consider two labelings $x_1$ and $x_2$, where $x_2$ is a descendant of $x_1$. When their distance is greater than 2, one can simply replace the subtree of $x_2$'s parent with the subtree of $x_2$. The resulting tree $T'$ preserves the evaluation of every predicate or non-elastic relation: the *root* property is obviously not affected; if $x_2$ is in the right (left) subtree of $x_1$, the *left-child* (*right-child*) relation is not affected anyway, and the *right-child* (*left-child*) relation is preserved as well, because the distance between $x_1$ and $x_2$ is still at least 2. Therefore, by Proposition 4.4.4 again, a $\textsc{Strand}^{syn}_{dec}$ formula is satisfiable iff it is satisfiable by a tree of size at most $2 \cdot (k+1)$, where $k$ is the number of existentially quantified variables in the formula.

## 4.5 Experimental Evaluation

We demonstrate the effectiveness and practicality of the decision procedures for Strand by checking verification conditions generated in proving properties of several heap-manipulating programs. As shown in Section 3.5, given pre-conditions, post-conditions and loop-invariants, each linear block of statements of a program yields a Hoare-triple, which is manually translated into a Strand formula $\psi$ over trees and integer arithmetic, as a verification condition. These examples include list-manipulating and tree-manipulating programs, including searching a sorted list, inserting into a sorted list, in-place reversal of a sorted list, the bubble-sort algorithm, searching for a key in a binary search tree, inserting into a binary search tree, and doing a left- or right-rotate on a binary search tree.

The programs `sorted-list-search` and `sorted-list- insert` search and insert a node in a sorted singly-linked list, respectively, while `sorted-list-insert-error` is the insertion program with an intended error. The program `sorted-list -reverse` is a routine for in-place reversal of a sorted singly-linked list, which results in a reverse-sorted list, and `bubblesort` is the code for Bubble-sort of a list. The routines `bst-search` and `bst-insert` search and insert a node in a binary search tree, respectively, while the programs `left-rotate` and `right-rotate` perform rotations (for balancing) in a binary search tree.

For all these examples, a set of partial correctness properties including both structural and data requirements is checked. For example, assuming a node with value $k$ exists, we check if both `sorted-list-search` and `bst-search` return a node with value $k$. For `sorted-list-insert`, we assume that the inserted value does not exist, and check if the resulting list contains the inserted node, and the sortedness property continues to hold. In the program `bst-insert`, assuming the tree does not contain the inserted node in the beginning, we check whether the final tree contains the inserted node, and the binary-search-tree property continues to hold. In `sorted-list-reverse`, we check if the output list is a valid list that is reverse-sorted. The code for `bubblesort` is checked to see if it results in a sorted list. And the `left-rotate` and `right-rotate` codes are checked to see whether they maintain the binary search-tree property.

Note that each program requires checking several verification conditions (typically for the linear block from the beginning of the program to a loop, for the loop invariant linear block, and for the block from the loop invariant to the end of the program).

### 4.5.1 Implementing $\textsc{Strand}_{dec}^{sem}$

We first implement the decision procedure for the semantically defined fragment $\textsc{Strand}_{dec}^{sem}$. Given a STRAND formula, our procedure will first determine if it is in the semantic decidable fragment, and if not, will halt and report that satisfiability of the formula is not checkable. When given a formula in the syntactic fragment $\textsc{Strand}_{dec}^{syn}$, this procedure will always succeed, and the decision procedure will determine satisfiability of the formula.

The decision procedure consists of a structural phase, where we determine whether the number of minimal models is finite, and if so, determine a bound on the size of the minimal models. This phase is effected by using MONA [42], a WS1S/WS2S solver over (strings and) trees. In the second data-constraint solving phase, the finite set of minimal models, if any, are examined by the data-solver Z3 [28] to check if they can be extended with data-values to satisfy the formula.

Instead of building an automaton representing the minimal models and then checking it for finiteness, we check the finiteness formula $MinModel_\psi$

using WS2S, which is supported by MONA, WS2S is interpreted over *infinite* trees with set-quantification restricted to finite sets. By quantifying over a finite universe $U$, and transforming all quantifications to be interpreted over $U$, we can interpret $MinModel_\psi$ over all finite trees. Let us denote this emulation as $MinModel'_{U,\psi}$. The finiteness condition can now be checked by asking if *there exists a finite set* $B$ such that any minimal model for $\psi$ is contained within the nodes of $B$:

$$\exists\ Bound\ \forall U\ \forall Q_{a(a\in\Sigma)}.\ \big(MinModel'_{U,\psi} \Rightarrow (U \subseteq Bound)\big)$$

This formula has no free-variables, and hence either holds on the infinite tree or not, and can be checked by MONA. This formula evaluates to `true` iff the formula is in $\text{STRAND}^{sem}_{dec}$.

We also follow a slightly different procedure to synthesize the data-logic formula. Instead of extracting each minimal model, and checking if there is a data-extension for it, we obtain a *bound* on the size of minimal models, and ask the data-solver to check for any model within that bound. This is often a much simpler formula to feed to the data-solver.

In our current implementation, the MONA constraints are encoded manually, and once the bound is obtained, we write a program that outputs the Z3 constraints for the verification condition and the bound. The translation from STRAND to MONA formulas and the translation from STRAND formulas to Z3 formulas for any bound can be automated, and is a plan for the future.

If the finiteness condition above evaluates to `true`, then we first check the satisfiability of the structural abstraction $\widehat{\psi}$ of $\psi$. If it is unsatisfiable, then by Proposition 4.2.3, $\psi$ is also unsatisfiable. Otherwise, we ask MONA for the minimal *Bound* of minimal models, which will be a prefix-closed set of nodes of the tree. Given *Bound*, $\psi$ is manually translated into a quantifier free formula over integer arithmetic, which is fed to Z3.

### 4.5.2 Implementing $\text{STRAND}^{syn}_{dec}$

Though the decision procedure for $\text{STRAND}^{sem}_{dec}$ in Section 4.3 is effective in verifying several heap-manipulating programs, the structural solving phase turns out to be the bottleneck for scaling up [50], because building an automaton representing the *minimal* models w.r.t. $\psi$ (those models satisfying

*MinModel*$_\psi$) is inefficient and depends on the size of $\psi$. Our new decision procedure for STRAND$_{dec}^{syn}$ also consists of a structural solving phase and a data solving phase, where the structure solving is done using *SmallModel*($\vec{x}$), which is independent to $\psi$, using MONA. Then we obtain a bound for small models by computing the longest acyclic path in the state diagram of the automaton generated by MONA. By Theorem 4.4.1, such a bound always exists. However, sinice the formula *SmallModel*($\vec{x}$) is easier to satisfy, we could obtain potentially larger bound from it than that from *MinModel*$_\psi$. The data solving phase, which is the same as the old decision procedure, checks for any model of $\psi$ within the bound.

In the structural solving phase using MONA, when a STRAND$_{dec}^{syn}$ formula $\psi$ is given, we further optimize the formula *SmallModel*($\vec{x}$) with respect to $\psi$ for better performance, as follows. First, a sub-formula $\beta_b(x, x') \Leftrightarrow$ $tailor_X(\beta_b(x, x'))$ appears in the formula only if the atomic formula $E_b(x, x')$ appears in $\psi$. Moreover, if $E_b(x, x')$ only appears positively, we use $\beta_b(x, x') \Rightarrow$ $tailor_X(\beta_b(x, x'))$ instead; similarly if $E_b(x, x')$ occurs only negatively, then we use $\beta_b(x, x') \Leftarrow tailor_X(\beta_b(x, x'))$ instead. This is clearly sound.

### 4.5.3 Experiments

For the heap-manipulating programs mentioned at the beginning of this section, all the generated verification conditions can be found at `http://web.engr.illinois.edu/~qiu2/strand` . It turns out that all of our verification conditions can be written entirely in the *syntactic* decidable fragment STRAND$_{dec}^{syn}$. Hence they definitely also fall in STRAND$_{dec}^{sem}$, which is strictly larger than STRAND$_{dec}^{syn}$.

In Table 4.1, we present the evaluation of our decision procedures for both STRAND$_{dec}^{sem}$ and STRAND$_{dec}^{syn}$ on checking these verification conditions. The experiments were conducted on a 2.2GHz, 4GB machine running Windows 7. For the structural-constraint solving phase, we report the maximum BDD sizes to represent automata, and the time taken by MONA to compute the minimal models or small models. For the data-constraint solving phase, we report the time spent by Z3 to check whether any structural model can be populated with data values to satisfy the verification condition. Note that if the formula checked by MONA is already unsatisfiable and there are no

| Program | Verification condition | Minimal Model computation (Alg. in Sec. 4.3) | | Small Model computation (Alg. in Sec. 4.4) | | Data constraint solving (Z3, QF-LIA) Sec. 4.3/4.4 Time (s) |
|---|---|---|---|---|---|---|
| | | Max. BDD size | Time (s) | Max. BDD size | Time (s) | |
| sorted-list-search | before-loop | 10009 | 0.34 | **540** | **0.01** | - |
| | in-loop | 17803 | 0.59 | **12291** | **0.14** | - |
| | after-loop | 3787 | 0.18 | **540** | **0.01** | - |
| sorted-list-insert | before-head | 59020 | 1.66 | **242** | **0.01** | 0.02/0.02 |
| | before-loop | 15286 | 0.38 | **595** | **0.01** | - |
| | in-loop | 135904 | 4.46 | **3003** | **0.03** | - |
| | after-loop | 475972 | 13.93 | **1250** | **0.01** | 0.02/0.03 |
| sorted-list-insert-error | before-loop | 14464 | 0.34 | **595** | **0.01** | 0.02/0.02 |
| sorted-list-reverse | before-loop | 2717 | 0.24 | **1155** | **0.01** | - |
| | in-loop | 89342 | 2.79 | **12291** | **0.14** | - |
| | after-loop | 3135 | 0.35 | **1155** | **0.01** | - |
| bubblesort | loop-if-if | 179488 | 7.70 | **73771** | **1.31** | - |
| | loop-if-else | 155480 | 6.83 | **34317** | **0.48** | - |
| | loop-else | 95181 | 2.73 | **7017** | **0.07** | 0.02/0.04 |
| bst-search | before-loop | 9023 | 5.03 | **1262** | **0.31** | - |
| | in-loop | 26163 | 32.80 | **3594** | **2.43** | 0.02/0.11 |
| | after-loop | 6066 | 3.27 | **1262** | **0.34** | - |
| bst-insert | before-loop | 3485 | 1.34 | **1262** | **0.34** | - |
| | in-loop | 17234 | 9.84 | **1908** | **1.38** | - |
| | after-loop | 2336 | 1.76 | **1807** | **0.46** | - |
| left-/right-rotate | bst-preserving | **1086** | 1.59 | 1510 | **0.48** | 0.05/0.14 |
| Total | | | 98.15 | | **7.99** | 0.15/0.36 |

Table 4.1: Results of program verification using STRAND
more details at `http://web.engr.illinois.edu/~qiu2/strand` .

models, the data-constraint solving phase is skipped (these are denoted by "-" annotations in the table for Z3).

All the verification conditions were successfully proved unsatisfiable, except those for the `sorted-list-insert-error` program, in which we introduced an error by removing the initial code that checks whether the inserted value is greater than the value at the head of the list and inserts the element before the head. The Z3 phase failed and gave as a counter-example a two-element list, with value at head equal to 0 and the value of the inserted value as $-6$. The `bst-insert` routine verifies without going to the Z3 phase, which means that the correctness is *independent* of the data-solver entirely (not even the transitivity of $\leq$ on integers is needed). The `bst-search` routine however does require help from the arithmetic solver.

The experimental results show that natural verification conditions tend to be expressible in the syntactic decidable fragment $\text{STRAND}_{dec}^{syn}$. Moreover, the expressiveness of our logic allows us to write complex conditions involving structure and data, and yet are handled well by MONA and Z3. We believe that a full-fledged engineering of an SMT solver for $\text{STRAND}_{dec}^{syn}$ that answers queries involving heap structures and data is a promising future direction. Towards this end, an efficient *non-automata theoretic* decision procedure (unlike MONA) that uses search techniques (like SAT) instead of representing the class of all models (like BDDs and automata) may yield more efficient decision procedures.

The experiments show that the decision procedure for $\text{STRAND}_{dec}^{syn}$ is considerably more efficient than the one for $\text{STRAND}_{dec}^{sem}$ on this set of examples.

## 4.6 Related Work

The work closest to ours is PALE [58], which is a logic on heaps structures but not data, and uses MSO and MONA [42] to decide properties of heaps. TASC [38] is similar but generalizes to reason balancedness in the cases of AVL and red-black trees. First-order logics with axiomatizations of the reachability relation (which cannot be expressed in FOL) have been proposed: axioms capturing *local* properties [55], a logic on regular patterns that is decidable [79], among others.

The logic in HAVOC, called LISBQ [46], offers reasoning with generic heaps combined with an arbitrary data-logic. The logic has restricted reachability predicates and universal quantification, but is syntactically severely curtailed to obtain decidability. Though the logic is not very expressive, it is extremely efficient, as it uses no structural solver, but translates the structure-solving also to the (Boolean aspect) of the SMT solver. The logic CSL [15] is defined in a similar vein as HAVOC, with similar sort-restrictions on the syntax, but generalizes to handle doubly-linked lists, and allows size constraints on structures. As far as we know, neither HAVOC nor CSL can express the verification conditions of searching a binary search tree. The work reported in [12] gives a logic that extends an LTL-like syntax to define certain decidable logic fragments on heaps.

The inference rule system proposed in [65] for reasoning with restricted reachability does not support universal quantification and cannot express disjointness constraints, but has an SMT solver based implementation [66]. Restricted forms of reachability were first axiomatized in early work by Nelson [59]. Several mechanisms without quantification exist, including the work reported in [67, 3]. Kuncak's thesis describes automatic decision procedures that approximate higher-order logic using first-order logic, through approximate logics over sets and their cardinalities [44].

Finally, separation logic [68] is a convenient logic to express heap properties of programs, and a decidable fragment (without data) on lists is known [9]. However, not many extensions of separation logics support data constraints (see [53] for one that does).

# CHAPTER 5

# NATURAL PROOFS

The Strand verification framework shown in Chapter 3 and Chapter 4 is a representative example of the *automated deductive verification paradigm* for software verification that combines user written modular contracts and loop invariants with automated theorem proving of the resulting verification conditions. The latter process is often executed by automated logical decision procedures supported by SMT solvers, which have emerged as robust and powerful engines to automatically find proofs. Besides Strand, several techniques and tools have been developed [26, 5, 41] and there have been several success stories of large software verification projects using this approach (the Verve OS project [78], the Microsoft hypervisor verification project using VCC [26], and a recent verified-for-security OS+browser for mobile applications [54], to name a few).

Verification conditions do not, however, always fall into decidable theories. In particular, the verification of properties of the *dynamically modified heap*, the focus of this dissertation, is a big challenge for logical methods. The dynamically manipulated heap poses several challenges, as typical correctness properties of heaps require complex combinations of structure (e.g., $p$ points to a tree structure, or to a doubly-linked list, or to an almost balanced tree, with respect to certain pointer-fields), data (the integers stored in data-fields of the tree respect the binary search tree property, or the data stored in a tree is a max-heap), and separation (the procedure modifies one list and not the other and leaves the two lists disjoint at exit, etc.).

The fact that the dynamic heap contains an unbounded number of locations means that expressing the above properties requires *quantification* in some form, which immediately precludes the use of most SMT decidable theories (there are only a few of them known that can handle quantification; e.g., the array property fragment [19] and the Strand logic presented in Chapter 3 and 4). Consequently, *expressing* such properties naturally and

succinctly in a logical formalism has been challenging, and reasoning with them automatically even more so.

For instance, in the Boogie line of tools (including VCC) of writing specifications using FOL and employing SMT solvers to validate verification conditions, the specification of invariants of even simple methods like *singly-linked-list insert* is tedious. In such code[1], second-order properties (reachability, acyclicity, separation, etc.) are smuggled in using carefully chosen *ghost variables*; for example, acyclicity of a list is encoded by assigning a ghost number (idx) to each node in the list, with the property that the numbers associated with adjacent nodes strictly increase going down the list. These ghost variables require careful manipulation when the structures are updated; for example, inserting a node may require updating the ghost numbers for other nodes in the list, in order to maintain the acyclicity property. Once such a ghost-encoding of the specification is formulated, the validation of verification conditions, which typically have quantifiers, are dealt with using sound heuristics (a wide variety of them including e-matching, model-based quantifier instantiation, etc. are available), but are still often not enough and have to be augmented by *instantiation triggers* from the verification engineer to help the proof go through.

Due to the inherent complexity and difficulty of dynamically allocated heap, most research on program logics for functional verification of heap-manipulating programs can be roughly divided into two classes:[2]

- **Logics for manual/semi-automatic reasoning:** The most popular of these are the class of *separation logics*[63, 68], but several others exist (see *matching logic* [69], for example). Complex structural properties of heaps are expressed using *inductive algebraic definitions*, the logic combines several other theories like arithmetic, etc., and uses a special separation operator ($*$) to compositionally reason with a footprint and the frame. The analysis is either manual or semi-automatic, the latter being usually sound, incomplete, and non-terminating, and proceeds by heuristically searching for proofs using a proof system, unrolling

[1] `http://vcc.codeplex.com/SourceControl/changeset/view/dcaa4d0ee8c2#vcc/Docs/Tutorial/c/7.2.list.c`

[2] We do not discuss abstraction-based approaches such as shape analysis here as such approaches are geared towards less complex specifications, and often are completely automatic, not even requiring proof annotations such as loop invariants; see Section 5.6.

recursive definitions arbitrarily. Typically, such tools can find simple proofs if they exist, but are unpredictable, and cannot robustly produce counter-examples.

- **Logics for completely automated reasoning:** These logics stem from the SMT (Satisfiability Modulo Theories) and automata theory literature, where the goal is to develop fast, terminating, sound and complete decision procedures, but where the logics are often constrained heavily on expressivity in order to reach these goals. Examples include several logics that extend first-order logic with reachability, the logics Lisbq [46] and CSL [15], and the logic $\text{Strand}_{dec}^{syn}$ (see Chapter 4) that combines tree theories with integer theories. The problem with these logics, in general, is that they are often not sufficiently expressive to state complex properties of the heap (e.g. the balancedness of an AVL tree, or that the set of keys stored in a heap do not change across a program).

We prefer an approach that combines the two methodologies above. In this chapter, we propose a novel proof strategy that calls *natural proofs*. Natural proofs are a subclass of proofs that are amenable to completely automated reasoning, that provide sound but incomplete procedures, and that capture common reasoning tactics in program verification. Intuitively, the strategy exploits a *fixed* set of proof tactics, keeping the expressiveness of powerful logics, retaining the automated nature of proving validity, but giving up on completeness (i.e., giving up decidability, retaining soundness). The idea of natural proofs is to identify a subclass of proofs $\mathcal{N}$ such that (a) a large class of valid verification conditions of real-world programs have a proof in $\mathcal{N}$, and (b) searching for a proof in $\mathcal{N}$ is *decidable*. In fact, we would even like the search for a proof in $\mathcal{N}$ to be efficiently decidable, possibly utilizing the automatic logic solvers (SMT solvers) that exist today. Natural proofs are hence a fixed set of proof tactics whose application is itself expressible in a decidable logic.

In particular, we investigate two commonly exploited proof tactics:

a) identifies a class of *simple and natural* proofs for proving verification conditions for heap-based programs, founded on how people prove these conditions manually, and

b) builds terminating procedures that efficiently and thoroughly search this class of proofs.

This results in a sound, incomplete, but terminating procedure that finds natural proofs automatically and efficiently. Many correct programs have simple proofs of correctness, and a terminating procedure that searches for these simple proofs efficiently can be a very useful tool in program verification. Incompleteness is, of course, a necessary trade-off to keep the logics expressive while having a terminating procedure, and a terminating automatic procedure is useful as it does not need manual help. Furthermore, as we shall see in this proposal, such decision algorithms are particularly desirable when they can be made to work very efficiently, especially using the fast-growing class of efficient SMT solvers for quantifier-free theories.

**Remark:** The idea of searching for only simple and natural proofs is not new; after all, type systems that prove properties of programs are essentially simple (and often scalable) proof mechanisms. The class of simple and natural proofs that we identify in this chapter is, however, quite different from those found by type systems.

When manually verifying a Hoare-triple that consists of code that manipulates heaps and where properties of heaps are expressed using inductive algebraic definitions, a very common tactic is to *unfold* the recursive definitions across the footprint, then *abstract* the recursively terms as uninterpreted terms (which we call *formula abstraction*) and using unification, prove the verification condition valid.

In order to illustrate the procedures for reasoning with heap-manipulating programs using natural proofs, we consider programs manipulating trees and develop a new recursive extension of first-order logic, called $\textsc{Dryad}_{\text{tree}}$ that allows stating complex properties of heaps without recourse to explicit quantification. $\textsc{Dryad}_{\text{tree}}$ combines quantifier-free first-order logic with recursive definitions, and these recursive definitions, themselves expressed in $\textsc{Dryad}_{\text{tree}}$, can capture several interesting properties of trees, including their height, the multiset of keys stored in them, whether they correspond to a binary search tree (or an AVL tree), etc.

**Organization:** We present the syntax and semantics of $\text{DRYAD}_{\text{tree}}$ in Section 5.1 with formal definition and examples. Section 5.2 shows the main technical contribution of this chapter: a precise VC-generation process for tree-manipulating programs annotated with the $\text{DRYAD}_{\text{tree}}$ logic. To reason with the resulting VC, we identify a decidable fragment of $\text{DRYAD}_{\text{tree}}$ in Section 5.3. Moreover, in Section 5.4, we apply the second proof tactic, the formula abstraction technique, to the $\text{DRYAD}_{\text{tree}}$ logic, and result in a sound but incomplete procedure. In Section 5.5, we experimentally evaluate the practicality of the natural proof strategy by verifying a set of tree-manipulating program. Section 5.6 compares our approach with the rich literature on heap verification, in both manual/automatic fashion.

## 5.1 The $\text{DRYAD}_{\text{tree}}$ Logic

The recursive logic over trees, $\text{DRYAD}_{\text{tree}}$, is essentially a *quantifier-free* first-order logic over heaps augmented with recursive definitions of various types (e.g., integers, sets/multisets of integers, etc.) defined for locations that have a tree under them. While FOL gives the necessary power to talk precisely about locations that are near neighbors, the recursive definitions allow expressing properties that require quantifiers, including reachability, collecting the set/multiset of keys in a tree, and defining natural metrics, like the height of a tree, that are typically useful in defining properties of trees.

### 5.1.1 Syntax

Given a finite set of *directions* $Dir$, let us define $PF$-trees as finite trees where every location has either $|PF|$ children, or is the `nil` location, which has no children (we assume there is a single `nil` location). Binary trees have two directions: $Dir = \{l, r\}$.

The logic $\text{DRYAD}_{\text{tree}}$ is parameterized by a finite set of *directions* $Dir$ and also by a finite set of *data-fields* $DF$. Let us fix these sets.

Let $Bool = \{\texttt{true}, \texttt{false}\}$ stand for the set of Boolean values, $Int$ stand for the set of integers and $Loc$ stand for the universe of locations. For any set $A$, let $\mathcal{S}(A)$ denote the set of subsets of $A$, and let $\mathcal{MS}(A)$ denote the set of all multisets with elements in $A$.

**Remark:** In this chapter, to simplify the presentation, we assume that the data fields are of type *Int*, and only addition and subtraction are allowed so that the underlying data theory (Presburger arithmetic) is decidable. In general, the data fields could be of arbitrary type, as long as a decidable data theory, e.g., real arithmetic, exists.

The DRYAD$_{\text{tree}}$ logic allows four kinds of recursively defined notions for a location that is the root of a *PF*-tree:

- recursively defined integer functions ($Loc \rightarrow Int$)
- recursively defined set-of-integers functions ($Loc \rightarrow \mathcal{S}(Int)$)
- recursively defined multiset-of-integers functions ($Loc \rightarrow \mathcal{MS}(Int)$)
- recursively defined Boolean predicates ($Loc \rightarrow Bool$)

Let us fix disjoint sets of countable names for such functions. We will refer to these recursive functions as recursively defined integers, recursively defined sets/multisets of integers, and recursively defined predicates, respectively. Typical examples of these include the height of a tree or the height of black-nodes in the tree rooted at a node (recursively defined integers), the set/multiset of keys stored at a particular data-field under nodes (recursively defined set/multiset of integers), and the property that the tree rooted at a node is a binary search tree or a balanced tree (recursively defined predicates).

A DRYAD$_{\text{tree}}$ formula consists of a pair $(Def, \varphi)$, where *Def* is a set of recursive definitions and $\varphi$ is a formula. The syntax of DRYAD$_{\text{tree}}$ logic is given in Figure 6.1, where the syntax of the formulas is followed by the syntax for recursive definitions. We require that every recursive function/predicate used in the formula $\varphi$ has a unique definition in *Def*. The figure does not define the syntax of the base and inductive formulas in recursive definitions (e.g. $i_{base}$, $i_{ind}$, etc.); we give that in the text below.

Location terms are formed using pointer fields from location variables, and include a special location called `nil`. Integer terms are obtained from integer constants, data-fields of locations, and from recursively defined integers, and combined using basic arithmetic operations of addition and subtraction and conditionals (ITE stands for if-then-else terms that evaluate to the second

$$
\begin{array}{lll}
dir \in Dir & i^* : Loc \rightarrow Int & x \in Loc \text{ Vars} \\
f \in DF & si^* : Loc \rightarrow \mathcal{S}(Int) & j \in Int \text{ Vars} \\
c : Int \text{ Constant} & msi^* : Loc \rightarrow \mathcal{MS}(Int) & MS \in \mathcal{MS}(Int) \text{ Vars} \\
q \in Boolean \text{ Vars} & p^* : Loc \rightarrow \{\texttt{true}, \texttt{false}\} & S \in \mathcal{S}(Int) \text{ Vars}
\end{array}
$$

$$
\begin{array}{rcl}
Loc \text{ Term: } lt, lt_1, lt_2 \ldots & ::= & x \mid \texttt{nil} \mid lt.dir \\
Int \text{ Term: } it, it_1, it_2, \ldots & ::= & c \mid j \mid lt.f \mid i^*(lt) \mid it_1 + it_2 \mid it_1 - it_2 \mid \\
 & & \text{ITE}(\varphi, it_1, it_2) \\
\mathcal{S}(Int) \text{ Term: } sit, sit_1, sit_2, \ldots & ::= & \emptyset \mid S \mid \{it\} \mid si^*(lt) \mid \\
 & & sit_1 \cup sit_2 \mid sit_1 \cap sit_2 \mid \\
 & & sit_1 \setminus sit_2 \mid \text{ITE}(\varphi, sit_1, sit_2) \\
\mathcal{MS}(Int) \text{ Term: } msit, msit_1, \ldots & ::= & \emptyset_m \mid MS \mid \{it\}_m \mid msi^*(lt) \mid \\
 & & msit_1 \cap msit_2 \mid msit_1 \cup msit_2 \mid \\
 & & msit_1 \setminus msit_2 \mid \text{ITE}(\varphi, msit_1, msit_2) \\
 & & \\
\text{Formula: } \varphi, \varphi_1, \varphi_2, \ldots & ::= & \texttt{true} \mid q \mid p^*(lt) \mid lt_1 = lt_2 \mid it_1 \leq it_2 \mid \\
 & & sit_1 \subseteq sit_2 \mid msit_1 \subseteq msit_2 \mid \\
 & & sit_1 \leq sit_2 \mid msit_1 \leq msit_2 \mid \\
 & & it \in sit \mid it \in msit \mid \neg\varphi \mid \varphi_1 \vee \varphi_2
\end{array}
$$

$$
\begin{array}{rcl}
\text{Recursively-defined integer :} & & \\
i^*(x) & \stackrel{def}{=} & \text{ITE}(x = \texttt{nil},\ i_{base},\ i_{ind}) \\
\text{Recursively-defined set-of-integers :} & & \\
si^*(x) & \stackrel{def}{=} & \text{ITE}(x = \texttt{nil},\ si_{base},\ si_{ind}) \\
\text{Recursively-defined multiset-of-integers :} & & \\
msi^*(x) & \stackrel{def}{=} & \text{ITE}(x = \texttt{nil},\ msi_{base},\ msi_{ind}) \\
\text{Recursively-defined predicate :} & & \\
p^*(x) & \stackrel{def}{=} & \text{ITE}(x = \texttt{nil},\ p_{base},\ p_{ind})
\end{array}
$$

Figure 5.1: Syntax of DRYAD$_{\text{tree}}$

argument if the first argument evaluates to `true` and evaluate to the third argument otherwise).

Terms that evaluate to a set/multiset of integers are obtained from recursively defined sets/multisets of integers corresponding to a location term, and are combined using set/multiset operations as well as conditional choices. Formulas are obtained by Boolean combinations of Boolean variables, recursively defined predicates on a location term, and using various relations

between set and multiset terms. The relations on sets and multisets include the subset relation as well as the relation $\leq$ which is interpreted as follows: for two sets (or multisets) of integers $S_1$ and $S_2$, $S_1 \leq S_2$ holds whenever for every $i \in S_1, j \in S_2$, $i \leq j$.

The recursively defined functions (or predicates) are defined using the syntax: $f^*(x) = \text{ITE}(x = \texttt{nil},\ f_{base},\ f_{ind})$, where $f_{base}$ and $f_{ind}$ are themselves terms (or formulas) that stand for what $f$ evaluates to when $x = \texttt{nil}$ (the base-case) and when $x \neq \texttt{nil}$ (the inductive step), respectively. There are two restrictions on these terms/formulas:

- $f_{base}$ has no free variables and hence evaluates to a fixed value (for integers, it is a fixed integer; for sets/multisets of integers, it is a fixed set; for Boolean predicates, it evaluates to `true` or `false`).

- $f_{ind}$ only has $x$ as a free variable. Furthermore, the location terms in it can only be $x$ and $x.pf$ (further dereferences are disallowed). Moreover, integer terms $x.pf.f$ are disallowed.

Intuitively, the above conditions demand that when $x$ is `nil`, the function evaluates to a constant of the appropriate type, and when $x \neq \texttt{nil}$, it evaluates to a function that is defined recursively using properties of the location $x$, which may include properties of the children of $x$, and these properties may in turn involve other recursively defined functions.

We assume that the inductive definitions are not *circular*. Formally, let *Def* be a set of definitions and consider a recursive definition of a function $f^*$ in *Def*. Define the sequence $\psi_0, \psi_1, \ldots$ as follows. Set $\psi_0 = f^*(x)$. Obtain $\psi_{i+1}$ by replacing every occurrence of $g^*(x)$ in $\psi_i$ by $g_{ind}(x)$, where $g$ is any recursively defined function in *Def*. We require that this sequence eventually stabilizes (i.e. there is a $k$ such that $\psi_k = \psi_{k+1}$). Intuitively, we require that the definition of $f^*(x)$ be rewritable into a formula that does not refer to a recursive definition of $x$ (by getting rewritten to properties of its descendants). We require that every definition in *Def* have the above property.

### 5.1.2 Examples

We illustrate the syntax of $\text{DRYAD}_{\text{tree}}$ with two examples as below. The BST data-structure can be handled in our $\text{STRAND}^{syn}_{dec}$ logic (see Example 3.4.5),

but in $\textsc{Dryad}_{\text{tree}}$, it is expressed in a recursive way, which is more intuitive. The RBT example is more challenging and beyond the expressiveness of STRAND.

**Example 5.1.1** (Binary search tree). *Binary search trees are defined with two directions $Dir = \{l, r\}$, with one data-field key, and with the following two simple recursive definitions: one recursively definition of sets of integers defines the set of keys of in the subtree below a node, and one recursive Boolean predicate that identifies nodes that have binary search trees under them.*

$$
\begin{aligned}
keys^*(u) &\stackrel{def}{=} \texttt{ite}(\ u = \texttt{nil}, \emptyset, \{u.key\} \cup keys^*(u.l) \cup keys^*(u.r)\ ) \\
bst^*(u) &\stackrel{def}{=} \texttt{ite}(\ u = \texttt{nil}, \texttt{true}, bst^*(u.l) \wedge bst^*(u.r) \wedge \\
&\qquad\qquad keys^*(u.l) \leq \{u.key\} \wedge \{u.key\} \leq keys^*(u.r)\ )
\end{aligned}
$$

*Now, $bst^*(x)$ says that $x$ has a binary search tree under it.*

*Note that though a binary search tree expressed in classical logic would require quantification over nodes, the above notation avoids this by using recursively-defined sets of keys to gather the data under a node, and using the $S_1 \leq S_2$ operation that implicitly has quantification, comparing all elements of $S_1$ with all elements of $S_2$.*

*We can also express, in our logic, various properties of binary search trees, like*

$$(bst^*(x) \wedge x.l \neq nil \wedge x.key = 20) \Rightarrow x.l.key \leq 20$$

*and using the natural proofs outlined in this chapter, check the validity of the above statement.*

**Example 5.1.2** (Red black tree). *Red black trees are semi-balanced binary search trees with nodes colored red and black, with all the leaves colored black, satisfying the condition that the left and right children of a red node are black, and the condition that the number of black nodes on paths from the root to any leaf is the same. This ensures that the longest path from root to a leaf is at most twice the shortest path from root to a leaf, making the tree roughly balanced.*

*We have two directions $PF = \{l,\ r\}$, and two data fields, key, and color. We model the color of nodes using an integer data-field color, which can be* 0

$$\begin{aligned}
black^*(x) &\stackrel{def}{=} \text{ITE}(\ x = \texttt{nil},\ \texttt{true},\ x.color = 0\ ) \\
bh^*(x) &\stackrel{def}{=} \text{ITE}(\ x = \texttt{nil},\ 1,\ \texttt{ite}(\ x.color = 0,\ 1,\ 0\ ) + \\
&\qquad \text{ITE}(bh^*(x.l) \geq bh^*(x.r), bh^*(x.l), bh^*(x.r)\ ) \\
keys^*(x) &\stackrel{def}{=} \text{ITE}(\ x = \texttt{nil}, \emptyset, \{x.key\} \cup keys^*(x.l) \cup keys^*(x.r)\ ) \\
rbt^*(x) &\stackrel{def}{=} \text{ITE}\,(\ x = \texttt{nil},\quad \texttt{true}, \\
&\qquad rbt^*(x.l)\ \wedge\ rbt^*(x.r) \wedge \\
&\qquad keys^*(x.l) \leq \{x.key\}\ \wedge\ \{x.key\} \leq keys^*(x.r) \wedge \\
&\qquad (x.color = 1 \rightarrow (black^*(x.l) \wedge black^*(x.r))\ ) \wedge \\
&\qquad bh^*(x.l) = bh^*(x.r)\ )
\end{aligned}$$

Figure 5.2: Recursive definitions for red black trees

*(black) or* $1$ *(red). We define four recursive functions/predicates: a predicate* $black^*(x)$ *that checks whether the root of the tree under* $x$ *is colored black (this is defined as a recursive predicate for technical reasons), the black height of a tree,* $bh^*(x)$*, the multiset of keys stored in a tree,* $keys^*(x)$*, and a recursive predicate that identifies red-black trees,* $rbt^*(x)$*. Their formal definitions in* DRYAD$_{tree}$ *are presented in Figure 5.2.*

*The* $black^*(xx)$ *recursive predicate asserts that the color of a* `nil` *node is* `black`*, and the color of a non-*`nil` *node is stored in the field color. The* $bh^*(t)$ *function definition says that the black height of a tree is* $1$ *for a* `nil` *node (*`nil` *nodes are assumed to be* `black`*), and, otherwise, the maximum of the black heights of the left and right subtree if the node* $x$ *is red, and the maximum of the black heights of the left and right subtree plus one, if* $x$ *is* `black`*. The* $keys^*(t)$ *function says that the multiset of keys stored in a tree is* $\emptyset$ *for a* `nil`*-node, and the union of the key stored in the node, and the keys of the left and right subtrees. Finally, the* $rbt^*(t)$ *holds if: (1) the left and right subtrees are valid red black trees; (2) the keys of the left subtree are no greater than the key in the node, and the keys of the right subtree are no less than the key in the node; (3) if the node is* `red`*, both its children are* `black`*; and (4) the black heights of the left and the right subtrees are equal.*

*We can also express, in our logic, various properties of red black trees, by including the above definitions in a formula like:*

$$(rbt^*(t)\ \wedge\ \neg black^*(t)\ \wedge\ t.key = 20) \rightarrow 10 \notin keys^*(t.r)$$

*The above statement is valid because $c^*(t) =$ `red` implies that $t$ is not `nil`, hence t.key is well defined, and since all the keys in the right are no less then the 20, it follows that 10 cannot be one of them. The validity of the above statement should be checkable using the procedures outlined in this chapter.*

### 5.1.3 Semantics

The Dryad$_{\text{tree}}$ logic is interpreted on (concrete) heaps. Let us fix a finite set of *program variables* $PV$. Concrete heaps are defined as follows ($f : A \rightharpoonup B$ denotes a partial function from $A$ to $B$):

**Definition 5.1.3.** *A concrete heap over a set of directions PF, a set of data-fields DF, and a set of program variables $PV$ is a tuple*

$$(N, nil, pf, df, pv)$$

*where:*

- *$N$ is a finite or infinite set of locations;*
- *$nil \in N$ is a special location representing the null pointer;*
- *$pf : (N \setminus \{nil\}) \times PF \to N$ is a function defining the direction fields;*
- *$df : (N \setminus \{nil\}) \times DF \to \mathbb{Z}$ is a function defining the data-fields;*
- *$pv : PV \rightharpoonup N \cup \mathbb{Z}$ is a partial function mapping program variables to locations or integers, depending on the type of the variable.* □

A concrete heap consists of a finite/infinite set of locations, with a pointer-field function $pf$ that maps locations to locations for each direction $pf \in PF$, a data-field function $df$ mapping locations to integers for each data-field $DF$, along with a unique constant location representing `nil` that has no data-fields or pointer-fields from it. Moreover, the function $pv$ is a partial function that maps program variables to locations and integers.

A Dryad$_{\text{tree}}$ formula with free variables $F$ is interpreted by interpreting the program variables in $F$ according to the function $pv$ and the other variables being given an interpretation (hence, for validity, these other variables are universally quantified, and for satisfiability, they are existentially quantified).

Each term evaluates to either a normal value of the corresponding type, or to `undef`. A location term is evaluated by dereferencing pointers in the heap. If a dereference is undefined, the term evaluates to `undef`. The set of locations that are roots of *PF*-trees are special in that they are the only ones over which recursive definitions are properly defined. A term of the form $i^*(lt)$, $si^*(lt)$ or $msi^*(lt)$ will evaluate to `undef` if $lt$ evaluates to `undef` or is not a root of a tree in the heap; otherwise it will be evaluated inductively using its recursive definition. Other aspects of the logic are interpreted with the usual semantics of first-order logic, unless they contain some subterm evaluating to `undef`, in which case they also evaluate to `undef`.

Each DRYAD formula evaluates to either `true` or `false`. To evaluate a formula $\varphi$, we first convert $\varphi$ to its *negation normal form (NNF)*, and evaluate each atomic formula of the form $p^*(lt)$ first. If $lt$ is not undefined, $p^*(lt)$ will be evaluated inductively using the recursive definition of $p^*$; if $lt$ evaluates to `undef`, $p^*(lt)$ will evaluate to `false` if $p^*(lt)$ appears positively, and will evaluate to `true` otherwise. Intuitively, undefined recursive predicates cannot help in making the formula true over a model. Similarly, atomic formulas involving terms that evaluate to `undef` are set to false or true depending on whether the atomic formula occurs within an even or odd number of negations, respectively. All other relations between integers, sets, and multisets are interpreted in the natural way, and we skip defining their semantics.

We assume that the DRYAD formulas always include a recursively defined predicate *tree* that is defined as:

$$tree^*(x) \stackrel{def}{=} (x = \texttt{nil}, \texttt{true}, \texttt{true})$$

Note that since recursively defined predicates can hold only on trees and since the above formula vacuously holds on any tree, $tree^*(x)$ holds iff $x$ is a root of a *PF*-tree.

## 5.2 Deriving the Verification Condition

The main technical contribution of this chapter is to show how a Hoare-triple corresponding to a basic path in a recursive imperative program (we disallow while-loops and demand all recursion be through recursive function

calls) with proof annotations written in DRYAD, can be expressed as a pair consisting of a finite footprint and a DRYAD formula. The finite footprint is a *symbolic heap* that captures the heap explored by the basic block of the program precisely. The construction of this footprint and formula calls for a careful handling of the mutating footprint defined by a recursive imperative program, calls for a disciplined approach to unrolling recursion, and involves capturing aliasing and separation by exploiting the fact that the manipulated structures are trees. In particular, the procedure keeps track of locations in the footprint corresponding to trees and precisely computes the value of recursive terms on the these. Furthermore, the verification condition is accurately described by unfolding the pre-condition so that it is expressed purely on the *frontier* of the footprint, so as to enable effective use of the formula abstraction mechanism. In order to be accurate, we place several key restrictions on the logical syntax of pre- and post-conditions expressed for functions.

We then consider the problem of solving the validity problem for the verification condition expressed as a footprint and a $\text{DRYAD}_{\text{tree}}$ formula. We turn to abstraction schemes for $\text{DRYAD}_{\text{tree}}$, and show how to abstract $\text{DRYAD}_{\text{tree}}$ formulas into quantifier-free theories of sets/multisets of integers; the latter can then be translated into formulas in the standard quantifier-free theory of integers with uninterpreted functions. The final formula's validity can be proved using standard SMT solvers, and its validity implies the validity of the $\text{DRYAD}_{\text{tree}}$ formula.

### 5.2.1 Programs

We consider imperative programs manipulating heap structures and the data contained in the heap. In this chapter, we assume that programs do not contain *while* loops and all recursion is captured using recursive function calls. Consequently, proof annotations only involve pre- and post-conditions of functions, and there are no loop-invariants.

The imperative programs we analyze will consist of integer operations, heap operations, conditionals and recursion. In order to verify programs with appropriate proof annotations, we need to verify linear blocks of code, called *basic blocks*, which do not have conditionals (conditionals are replaced

with `assume` statements). Basic blocks always start from the *beginning* of a function and either end at an assertion in the program (checking an intermediate assertion), or end at a function call to check whether the pre-condition to calling the function holds, or ends at the end of the program in order to check whether the post-condition holds. Basic blocks can involve recursive and non-recursive function calls.

We define basic blocks using the following grammar, parameterized by a set of directions *PF* and a set of data-fields *DF*:

$$
\begin{array}{lcl}
bb & :- & bb';\ \mid\ bb';\ \texttt{return}\ u;\ \mid\ bb';\ \texttt{return}\ j; \\
bb' & :- & bb';bb' \mid u := v \mid u := \texttt{nil} \mid u := v.dir \mid u.dir := v \mid \\
 & & j := u.f \ \mid\ u.f := j \ \mid\ u := \texttt{new} \ \mid\ j := aexpr \ \mid \\
 & & \texttt{assume}\ (bexpr) \ \mid\ u := f(v, z_1, \ldots, z_n) \ \mid\ j := g(v, z_1, \ldots, z_n) \\
aexpr & :- & j \ \mid\ aexpr + aexpr \ \mid\ aexpr - aexpr \\
bexpr & :- & u = v \mid u = \texttt{nil} \mid aexpr \leq aexpr \mid \neg bexpr \mid bexpr \vee bexpr
\end{array}
$$

Since we deal with tree data-structure manipulating programs, which often involve functions that take as input a tree and return a tree, we make certain crucial assumptions. One crucial restriction we assume for the technical exposition is that all functions take in at most *one* location parameter as input (the rest have to be integers). Basic blocks hence have function calls of the form $f(v, z_1, \ldots z_n)$, where $v$ is the only location parameter. This restriction greatly simplifies the proofs as it is much easier to track one tree. We can relax this assumption, but when several trees are passed as parameters, our decision procedures will implicitly assume a precondition that the trees are all disjoint. This is crucial: our decision procedures cannot track trees that "collide"; they track only equal trees and disjoint trees. This turns out to be a natural property of most data-structure manipulating programs.

## Restrictions

We consider the basic blocks described above annotated with $\text{DRYAD}_{\text{tree}}$ formulas. However, we place stringent restrictions on annotations (pre- and post-function) that we allow in our framework, guaranteeing that the program always manipulates appropriate trees. These restrictions are important for the technique in this chapter and is the price we pay for automation.

Recall that we allow only two kinds of functions, one returning a location $f(v, z_1, \ldots, z_n)$ and one returning an integer $g(v, z_1, \ldots, z_n)$ ($v$ is a location parameter, $z_1, \ldots, z_n$ are integer parameters). We require that $v$ is the root of a *PF*-tree at the point when the function is called, and this is an implicit pre-condition of the function called.

Each function is annotated with its pre- and post-conditions using *annotating formulas*. Annotating terms and formulas are $\text{DRYAD}_{\text{tree}}$ terms and formulas that do not refer to any child or any data field, do not allow any equality between locations and do not allow `ite`-expressions. We denote the pre-condition as a *pre-annotating formula* $\psi(v, z_1, \ldots, z_n)$.

The post-condition annotation is more complex, as it can talk about properties of the heap at the pre-state as well as the post-state. We allow combining terms and formulas obtained from the pre-heap and the post-heap to express the post-condition. Terms and formulas over the post-heap are obtained using $\text{DRYAD}_{\text{tree}}$ annotating terms and formulas that are allowed to refer to a variable *old_v* which points to the location $v$ pointed to in the pre-heap. These terms and formulas can also refer to the variable *ret_loc* or *ret_int* to refer to the location or integer being returned. Terms and formulas over the pre-heap are obtained using $\text{DRYAD}_{\text{tree}}$ annotating terms and formulas referring to *old_v* and $old_z_i$'s, except that all recursive definitions are renamed to have the prefix *old_* . Then a *post-annotating formula* combines terms and formulas expressed over the pre-heap and the post-heap (using the standard operations).

For a function $f(v, z_1, \ldots z_n)$ that returns a location, we assume that the returned location always has a *PF*-tree under it (and this is implicitly assumed to be part of the post-condition). The post-condition for $f$ is either of the form

$$\texttt{havoc}(old_v) \wedge \psi(old_v, old_z_1, \ldots, old_z_n, ret_loc)$$

or of the form

$$\texttt{tree}(old_v) \wedge \texttt{tree}(ret_loc) \wedge old_v \# ret_loc \wedge \psi(old_v, old_z_1, \ldots, old_z_n, ret_loc)$$

where $\psi$ is a post-annotating formula. In the first kind, $\texttt{havoc}(old_v)$ means that the function guarantees nothing about the location pointed to in the pre-

state by the input parameter $v$ (and nothing about the locations accessible from that location) and hence the caller of $f$ cannot assume anything about the location it passed to $f$ after the call returns. In that case, we restrict $\psi$ from referring to $r^*(old_v)$, where $r^*$ is a recursive predicate/function on the post-heap. In the latter kind $old_v \# ret_loc$ means that $f$, at the point of return, assures that the location passed as parameter $v$ now points to a *PF*-tree and this tree is disjoint from the tree rooted at *ret_loc*.

In either case, the formula $\psi$ can relate complex properties of the returned location and the input parameter, including recursive definitions on the old parameter and the new ones. For example, a post-condition of the form $\texttt{havoc}(old_v) \wedge keys^*(old_v) = keys^*(ret_loc)$ says that the keys under the returned location are precisely the same as the keys under the location passed to the function.

For a function $g$ returning an integer, the post-condition is of the form

$$\texttt{tree}(old_v) \wedge \psi(old_v, old_z_1, \ldots, old_z_n, ret_int)$$

or of the form

$$\texttt{havoc}(old_v) \wedge \psi(old_v, old_z_1, \ldots, old_z_n, ret_int)$$

The former says that the location passed as input continues to point to a tree, while the latter says that no property is assured about the location passed as input (same restriction on $\psi$ applies).

The above restriction that the input tree and the returned tree either point to completely disjoint trees or that the input pointer (and nodes accessible from it) are entirely havoc-ed and the returned node is some tree are the only separation and aliasing properties that the post-condition can assert. Our logical mechanism is incapable, for example, of saying the the returned node is a reachable node from the location passed to the function. We have carefully chosen such restrictions in order to simplify tracking tree-ness and separation in the footprint. In practice, most data-structure algorithms fall into these categories (for example, an insert routine would havoc the input tree and return a new tree whose keys are related to the keys of the input tree, while a tree-copying program will return a tree disjoint from the input tree).

### 5.2.2 Describing the Verification Condition in $\text{DRYAD}_{\text{tree}}$

Given a set of recursive definitions, and a Hoare-triple $(\varphi_{pre}, bb, \varphi_{post})$, where $bb$ is a basic block, we now show how to systematically define the verification condition corresponding to it. Note that since we do not have while-loops, basic blocks always start at the beginning of a function and go either till the end of the function (spanning calls to other functions) or go up to a function call (in order to check if the pre-condition for that call holds). In the former case, the post-condition is a post-condition annotation. In the latter case, we need another form:

$$\texttt{tree}(y) \wedge \psi(\bar{x})$$

where $\bar{x}$ is a subset of program variables. The pre-condition of the called function implicitly assumes that the input location is a tree (which is expressed using $\texttt{tree}(y)$ above), and the pre-condition itself is adapted (after substituting formal parameters with actual terms passed to the function) and written as the formula $\psi$.

This verification condition is expressed as a combination of

a) quantifier-free formulas that define properties of the footprint the basic block uncovers on the heap, combined with

b) recursive formulas expressed only on the frontier of the footprint.

This verification condition is formed by unrolling recursive definitions appropriately as the basic block increases its footprint so that recursive properties are translated to properties of the frontier. This allows us to write the (strongest) post-condition of $\varphi_{pre}$ on precisely the same nodes as $\varphi_{post}$ refers to, which then allows us to apply *formula abstractions* to prove the verification condition. Also, recursive calls to functions that process the data-structure recursively are naturally called on the frontier of the footprint, which allows us to summarize the call to the function on the frontier.

The verification condition is derived in two steps, exploiting the two specific tactics for the natural proof strategy. In the first step, we inductively define a *footprint structure*, composed of a *symbolic heap* and a $\text{DRYAD}_{\text{tree}}$ formula, which captures the state of the program that results when the basic block executes from a configuration satisfying the pre-condition. We then incorporate the post-condition and derive the verification condition.

**Symbolic heap.** A symbolic heap is defined as follows:

**Definition 5.2.1.** *A symbolic heap over a set of directions PF, a set of data-fields DF, and a set of program variables PV is a tuple*

$$(C, S, I, c_{nil}, pf, df, pv)$$

*where:*

- $C$ *is a finite set of* concrete nodes*;*
- $S$ *is a finite set of* symbolic tree nodes *with* $C \cap S = \emptyset$*;*
- $I$ *is a set of integer variables;*
- $c_{nil} \in C$ *is a special concrete node representing nil;*
- $pf : (C \setminus \{c_{nil}\}) \times PF \rightharpoonup C \cup S$ *is a partial function mapping every pair of a concrete node and a direction to nodes (concrete or symbolic);*
- $df : (C \setminus \{c_{nil}\}) \times DF \rightharpoonup I$ *is a partial function mapping concrete nodes and data-fields pairs to integer variables;*
- $pv : PV \rightharpoonup C \cup S \cup I$ *is a partial function mapping program variables to nodes or integer variables (location variables are mapped to* $C \cup S$ *and integer variables to* $I$*).* □

Intuitively, a symbolic heap $(C, S, I, c_{nil}, pf, df, pv)$ has two finite sets of nodes: concrete nodes $C$ and symbolic tree nodes $S$, with the understanding that each $s \in S$ stands for a node that may have an arbitrary *PF*-tree under it, and furthermore the separation constraint that for any two symbolic tree nodes $s,\ s' \in S$, the trees under it would not intersect with each other, nor with the nodes in $C$. The tree under a symbolic node is not represented in the symbolic heap at all. One of the concrete nodes ($c_{nil}$) represents the `nil` location.

The function *pf* captures the pointer-field *pf* in the heap that is within the footprint, and maps the set of concrete nodes to concrete and symbolic nodes. The pointer fields of symbolic nodes are not modeled, as they are part of the tree below the node that is not represented in the footprint. The functions *df* and *pv* capture the data-fields (mapping to integer variables) and program variables restricted to the nodes in the symbolic heap.

A symbolic heap hence represents a (typically infinite) set of concrete heaps, namely those in which it can be embedded. We define this formally using the notion of *correspondence* that captures when a concrete heap is represented by a symbolic heap.

**Definition 5.2.2.** *Let $SH = (C, S, I, c_{nil}, pf, df, pv)$ be a symbolic heap and let $CH = (N, nil, pf', df', pv')$ be a concrete heap. Then $CH$ is said to* correspond *to $SH$ if there is a function $h : C \cup S \to N$ such that the following conditions hold:*

- *$h(c_{nil}) = nil$;*
- *for any $n, n' \in C$, if $n \neq n'$, then $h(n) \neq h(n')$;*
- *for any two nodes $n \in C \setminus \{c_{nil}\}$, $n' \in C \cup S$, and for any $pf \in PF$, if $pf(n, pf) = n'$, then $pf'(h(n), pf)) = h(n')$;*
- *for any $s \in S$, $h(s)$ is the root of a PF-tree in CH, and there is no concrete node $c \in C \setminus \{c_{nil}\}$ such that $h(c)$ belongs to this tree;*
- *for any $s, s' \in S$, $s \neq s'$, the PF-trees rooted at $h(s)$ and $h(s')$ (in CH) are disjoint except for the nil node;*
- *for any location variable $v \in PV$, if $pv(v)$ is defined, then $pv'(v) = h(pv(v))$;* □

Intuitively, $h$ above defines a restricted kind of homomorphism between the nodes of the symbolic heap *SH* and a portion of the concrete heap *CH*. Distinct concrete non-nil nodes are required to map to distinct locations in the concrete heap. Symbolic nodes are required to map to trees that are disjoint (save the *nil* location); they can map to the *nil* location as well. The trees rooted at locations corresponding to symbolic nodes must be disjoint from the locations corresponding to concrete nodes. Note that there is no requirement on the integer variables $I$ and the map $pv'$ on integer variables and the maps $df$ and $df'$. Note also that for a concrete node in the symbolic heap $n$, the fields defined from $n$ in the symbolic heap must occur in the concrete heap as well from the corresponding location $h(n)$; however, the fields not defined for $n$ may or may not be defined on $h(n)$.

A *footprint* is a pair $(SH; \varphi)$ where *SH* is a symbolic heap and $\varphi$ is a $\text{DRYAD}_{\text{tree}}$ formula. The semantics of such a footprint is that it represents all concrete heaps that both correspond to *SH* and satisfy $\varphi$.

**Tree-ness of nodes.** The key property of a symbolic heap is that we can determine that certain nodes have *PF*-trees rooted under them (i.e. in *any* concrete heap corresponding to the symbolic heap, the corresponding locations will have a *PF*-tree under them).

For a symbolic heap $SH = (C, S, I, c_{nil}, pf, df, pv)$, let the set of *graph nodes* of *SH* be the smallest set of nodes $V \subseteq C \cup S$ such that:

- $c_{nil} \in V$ and $S \subseteq V$
- For any node $n \in C$, if for every $pf \in PF$, $pf(n, pf)$ is defined and belongs to $V$, then $n \in V$.

Now define *Graph*(*SH*) to be the directed graph $(V, E)$, where $V$ is as above, and $E$ is the set of edges $(u, v)$ such that $pf(u, pf) = v$ for some $pf \in Dir$. Note that, by definition, there are no edges out of $u$ if $u \in S$, as symbolic nodes do not have outgoing fields.

We say that a node $u$ in $V$ is the root of a tree in *Graph*(*SH*) if the set of all nodes reachable from $u$ forms a tree (in the usual graph-theoretic sense).

The following claim follows and is the crux of using the symbolic heap to determine tree-ness of nodes:

**Lemma 5.2.3.** *Let SH be a symbolic heap and let CH be a corresponding concrete heap, defined by a function $h$. If a node $u$ is the root of a tree in Graph(SH), then $h(u)$ also subtends a tree in CH.*

*Proof.* A proof gist is as follows. First, note that symbolic nodes and the node $c_{nil}$ are always roots of trees in *Graph*(*SH*) and the locations in the concrete heap corresponding to them subtend trees (in fact, disjoint trees save the *nil* location). Turning to concrete nodes, we need to argue that if $c$ is a concrete node in *Graph*(*SH*), then $h(c)$ is the root of a *PF*-tree in *CH*. This follows by induction on the height of the tree under $c$ in *Graph*(*SH*), since each of the *PF* children of $c$ in *Graph*(*SH*) must either be the $c_{nil}$ node or a summary node or a concrete node that is a the root of a tree of less height. The corresponding locations in *CH*, by induction hypothesis or by the above observations, have *PF*-trees suspended from them. In fact, by the definition of correspondence, these trees are all disjoint except for the *nil* location (since trees corresponding to summary nodes are all disjoint and disjoint from locations corresponding to concrete nodes, and since concrete nodes in the symbolic heap denote ).

The location corresponding to a concrete node in $Graph(SH)$ that does not have all $PF$-fields defined in $SH$ may or may not have a $PF$-tree subtended from it; this is because the notion of correspondence allows the corresponding location to have more fields defined. In the sequel, when we use symbolic heaps for tracking footprints, such concrete nodes with partially defined $PF$ fields will occur only when processing function calls (where all information about a node may not be known). $\square$

**Initial footprint.** Let the pre-condition be $\varphi_{pre}(u, j_1, \ldots, j_m)$, where $u$ is the only location program variable, there is a $PF$-tree rooted at $u$, and $j_1, \ldots, j_m$ are integer program variables. Then we define the initial symbolic heap:

$$(C_0, S_0, I_0, c_{nil}, pf_0, df_0, pv_0)$$

where $C_0 = \{c_{nil}\}$, $S_0 = \{n_0\}$, $I = \{i_1, \ldots i_m\}$, $pf_0$ and $df_0$ are empty functions (i.e. functions with an empty domain), and $pv_0$ maps $u$ to $n_0$ and $j_1, ..., j_m$ to $i_1, ..., i_m$, respectively. The initial formula $\varphi_0$ is obtained from $\varphi_{pre}(u, j_1, \ldots, j_m)$ by replacing $u$ by $n_0$ and $j_1, ..., j_m$ by $i_1, ..., i_m$, and by adding the conjunct $p^*(c_{nil}) \leftrightarrow p_{base}$ or $f^*(c_{nil}) = f_{base}$ for all recursive predicates and functions. Note that the formula is defined over the variables $S_0 \cup I_0$. Intuitively, we start at the beginning of the function with a single symbolic node that stands for the input parameter, which is a tree, and a concrete node that stands for `nil`. All integer parameters are assigned to distinct variables in $I$.

**Expanding the footprint.** A basic operation on a pair, $(SH; \varphi)$, consisting of a symbolic heap, and a formula is expansion. Let $SH$ be

$$(C, S, I, c_{nil}, pf, df, pv)$$

and $n \in C \cup S$ be a node. We define $expand\big((SH; \varphi), n\big) = (SH'; \varphi')$, where $SH'$ is the tuple

$$(C', S', I', c_{nil}, pf', df', pv')$$

as follows: if $n \in C$ (the node is already expanded), then do nothing by setting $(SH'; \varphi')$ to $(SH; \varphi)$; otherwise:

- $C' = C \cup \{n\}$, where $n$ is the node being expanded

- $S' = S \uplus \{n_{pf} \mid pf \in PF\} \setminus \{n\}$, where each $n_{pf}$ is a fresh new node different from the nodes in $C \cup S$

- $I' = I \uplus \{i_f \mid f \in DF\}$, where each $i_f$ is a fresh new integer variable

- $pf' \mid_{C\setminus\{c_{nil}\}\times PF} = pf$, and $pf'(n, pf) = n_{pf}$ for all $pf \in PF$

- $df' \mid_{C\setminus\{c_{nil}\}\times DF} = df$, and $df'(n, f) = i_f$ for all $f \in DF$

- $pv' = pv$;

The formula $\varphi'$ is obtained from the formula $\varphi$ as follows:

$$\begin{aligned}
\varphi' \;=\; & \varphi[\bar{p}_n, \bar{f}_n / \bar{p}^*(n), \bar{f}^*(n)] \\
\wedge\; & \bigwedge_{p^*} \Big(p_n \leftrightarrow \hat{p}_{ind}(n)\Big) \wedge \bigwedge_{f^*} \Big(f_n = \hat{f}_{ind}(n)\Big) \\
\wedge\; & n \neq c_{nil} \wedge \bigwedge_{n' \in C' \setminus \{c_{nil}\}, pf \in PF} \Big(n' \neq n_{pf}\Big) \\
\wedge\; & \bigwedge_{pf \in PF, s \in S} \Big(n_{pf} = s \rightarrow n_{pf} = c_{nil}\Big) \\
\wedge\; & \bigwedge_{pf_1, pf_2 \in PF, pf_1 \neq pf_2} \Big(n_{pf_1} = n_{pf_2} \rightarrow n_{pf_1} = c_{nil}\Big)
\end{aligned}$$

where $\bar{p}_n$ are fresh Boolean variables, $\bar{f}_n$ are fresh term (integer, set, ...) variables, $p^*(x) \stackrel{def}{=} \texttt{ite}(x = \texttt{nil}, p_{base}, p_{ind}(x))$ ranges over all the recursive predicates, and $f^*(x) \stackrel{def}{=} \texttt{ite}(x = \texttt{nil}, f_{base}, f_{ind}(x))$ ranges over all the recursive functions. Intuitively, The variables $\bar{p}_n$ and $\bar{f}_n$ capture the values of the predicates and functions for the node $n$ in the current symbolic heap. This is possible because the values of the recursive predicates and functions for concrete non-*nil* nodes are determined by the values of the functions and predicates for symbolic nodes and the *nil* node. The formula $\hat{p}_{ind}(n)$ is obtained from $p_{ind}(n)$, by substituting every location term of the form $n.pf$ with $n_{pf}$ for every $pf \in PF$, and substituting every integer term of the form $n.f$ with $i_f$ for every $f \in DF$. The term $\hat{f}_{ind}(n)$ is obtained by the same substitutions.

**Evolving the footprint on basic blocks.** Given a symbolic heap *SH* along with a formula $\varphi$, and a basic block *bb*. We compute the symbolic execution of *bb* using the transformation function $st\big((SH; \varphi), bb\big) = (SH'; \varphi')$. The transformation function *st* is computed transitively; i.e., if *bb* is of the

form $(stmt; bb')$ where $stmt$ is an atomic statement and $bb'$ is a basic block, then

$$st\big((SH;\varphi), bb\big) = st\big(st\big((SH;\varphi), stmt\big), bb'\big)$$

Therefore, it is enough to define the transformation for the various atomic statements. Given $SH = (C, S, I, c_{nil}, pf, df, pv)$, $\varphi$ and an atomic statement $stmt$, we define $st\big((SH;\varphi), stmt\big)$ as follows by cases of $stmt$. Unless some assumptions fail (in which case the transformation is undefined), we describe $st\big((SH;\varphi), stmt\big)$ as $(SH';\varphi')$.

As per our convention, function updates are denoted in the form of $[arg \leftarrow new_val]$. For example, $pv[u \leftarrow n]$ denotes the function $pv$ except that $pv(u)$ maps to $n$. Formula substitutions are denoted in the form of $[new/old]$. For example, $\varphi[df'/df]$ denotes the formula obtained from the formula $\varphi$ by substituting every occurrence of $df$ with $df'$.

The following defines how the footprint evolves across all possible statements *except* function calls:

**(a)** $stmt : u := v$

If $pv(v)$ is undefined, the transformation is undefined; otherwise

$$\begin{aligned} SH' &= (C, S, I, c_{nil}, pf, df, pv[u \leftarrow pv(v)]) \\ \varphi' &\equiv \varphi \end{aligned}$$

**(b)** $stmt : u := \texttt{nil}$

$$\begin{aligned} SH' &= (C, S, I, c_{nil}, pf, df, pv[u \leftarrow c_{nil}]) \\ \varphi' &\equiv \varphi \end{aligned}$$

**(c)** $stmt : u := v.pf$

If $pv(v)$ is undefined, or $pv(v) \in C$ and $pf(pv(v), pf)$ is undefined, the transformation is undefined. Otherwise we expand the symbolic heap:

$$((C'', S'', I'', c_{nil}, pf'', df'', pv''); \varphi'') = expand\big((SH;\varphi), pv(v)\big)$$

Now $pv''(v)$ must be in $C'' \setminus \{c_{nil}\}$, and we set

$$\begin{aligned} SH' &= (C'', S'', I'', c_{nil}, pf'', df'', pv''[u \leftarrow pf''(pv''(v), pf)]) \\ \varphi' &\equiv \varphi'' \end{aligned}$$

**(d)** $stmt : j := v.f$

If $pv(v)$ is undefined, or $pv(v) \in C$ and $pf(pv(v), f)$ is undefined, the transformation is undefined. Otherwise we expand the symbolic heap:

$$((C'', S'', I'', c_{nil}, pf'', df'', pv''); \varphi'') = expand\big((SH; \varphi), pv(v)\big)$$

Now $pv''(v)$ must be in $C'' \setminus \{c_{nil}\}$, and we set

$$\begin{aligned} SH' &= (C'', S'', I'' \uplus \{i\}, c_{nil}, pf'', df'', pv''[j \leftarrow i]) \\ \varphi' &\equiv \varphi'' \wedge i = df''(pv''(v), f) \end{aligned}$$

**(e)** $stmt : u.pf := v$

If $pv(u)$ or $pv(v)$ is undefined, or $pv(u) = c_{nil}$, the transformation is undefined. Otherwise we expand the symbolic heap:

$$((C'', S'', I'', c_{nil}, pf'', df'', pv''); \varphi'') = expand\big((SH; \varphi), pv(u)\big)$$

Now $pv''(u)$ must be in $C'' \setminus \{c_{nil}\}$, and we set

$$\begin{aligned} SH' &= (C'', S'', I'', c_{nil}, pf''[(pv''(u), pf) \leftarrow pv''(v)], df'', pv'') \\ \varphi' &\equiv \varphi'' \end{aligned}$$

**(f)** $stmt : u.f := j$

If $pv(u)$ or $pv(j)$ is undefined, or $pv(u) = c_{nil}$, the transformation is undefined. Otherwise we expand the symbolic heap:

$$((C'', S'', I'', c_{nil}, pf'', df'', pv''); \varphi'') = expand\big((SH; \varphi), pv(u)\big)$$

Now $pv''(u)$ must be in $C'' \setminus \{c_{nil}\}$, and we set

$$\begin{aligned} SH' &= (C'', S'', I'' \uplus \{i\}, c_{nil}, pf'', df''[(pv''(u), f) \leftarrow i], pv'') \\ \varphi' &\equiv \varphi'' \wedge i = pv''(j) \end{aligned}$$

**(g)** $stmt : u :=$ `new`

We assume that, for the new location, every pointer initially points to

nil and every data field initially evaluates to 0.

$$\begin{aligned} SH' &= (C \uplus \{n\}, S, I \uplus \{i_f \mid f \in DF\}, c_{nil}, pf', df', pv[u \leftarrow n]) \\ \varphi' &\equiv \varphi \wedge \bigwedge_{f \in DF} \Big( i_f = 0 \Big) \wedge \bigwedge_{n' \in C \cup S} \Big( n \neq n' \Big) \end{aligned}$$

where $pf'$ and $df'$ are defined as follows:

- $pf' \mid_{C \setminus \{c_{nil}\} \times PF} = pf$, and $pf'(n, pf) = c_{nil}$ for all $pf \in PF$
- $df' \mid_{C \setminus \{c_{nil}\} \times DF} = df$, and $df'(n, f) = i_f$ for all $f \in DF$

**(h)** $stmt : j := aexpr(\bar{k})$

If $pv$ is undefined on any variable in $\bar{k}$, then the transformation is undefined; otherwise

$$\begin{aligned} SH' &= (C, S, I \uplus \{i\}, c_{nil}, pf, df, pv[j \leftarrow i]) \\ \varphi' &\equiv \varphi \wedge i = aexpr[pv(\bar{k})/\bar{k}] \end{aligned}$$

**(i)** $stmt$ : `assume` $bexpr(\bar{v}, \bar{j})$

If $pv$ is undefined on any variable in $pv(\bar{v})$ or in $pv(\bar{j})$, then the transformation is undefined; otherwise

$$\begin{aligned} SH' &= SH \\ \varphi' &\equiv \varphi \wedge bexpr[pv(\bar{v}), pv(\bar{j})/\bar{v}, \bar{j}] \end{aligned}$$

**(j)** $stmt$ : `return` $u$

If $pv(u)$ is undefined, the transformation is undefined; otherwise

$$\begin{aligned} SH' &= (C, S, I, c_{nil}, pf, df, pv[ret \leftarrow pv(u)]) \\ \varphi' &\equiv \varphi \end{aligned}$$

**(k)** $stmt$ : `return` $j$

If $pv(j)$ is undefined, the transformation is undefined; otherwise

$$\begin{aligned} SH' &= (C, S, I \uplus \{i\}, c_{nil}, pf, df, pv[ret_int \leftarrow i]) \\ \varphi' &\equiv \varphi \wedge i = pv(j) \end{aligned}$$

We can show that for any atomic statement that is not a function call, the above computes the strongest post of the footprint:

**Theorem 5.2.4.** *Let $(SH; \varphi)$ be a footprint and let* **stmt** *be any statement that is not a function call. Let $(SH'; \varphi')$ be the footprint obtained from $(SH; \varphi)$ across the statement* **stmt**, *as defined above. Let $\mathcal{C}$ denote the set of all concrete heaps that correspond to SH and satisfy $\varphi$, and let $\mathcal{C}'$ be the set of all heaps that result from executing* **stmt** *from any concrete heap in $\mathcal{C}$. Then $\mathcal{C}'$ is the precise set of concrete heaps that correspond to $SH'$ and satisfy $\varphi'$.*

*Proof.* The proof consists of case analysis for each type of statement:

[ $u := v$ ] The variable assignment makes $u$ points to where $v$ points to. Hence $pv(u)$ is updated with $pv(v)$. Since the heap is unmodified, all the heap domain ($C$ and $S$), pointer fields ($pf$) and data fields ($df$) remain the same. The constraints $\varphi'$ is also unchanged.

[ $u :=$ `nil` ] The variable assignment makes $u$ points to *nil*, so $pv(u)$ is updated with $c_{nil}$. Similar to the above case, the heap and the formula are completely unmodified.

[ $u := v.\boldsymbol{pf}$ ] The dereferencing on $v$ requires an expanding of $(SH; \varphi)$ to $((C'', S'', I'', c_{nil}, pf'', df'', pv''); \varphi'')$, which is sound (the proof is omitted). Moreover, the assignment makes $u$ points to the $pf$ field of $v$, formally $pv''(u)$ is updated with the $pf$ field of $pv''(v)$. Similar to the above case, the heap and the formula are also unmodified from $(SH; \varphi)$ to $(SH'; \varphi')$.

[ $u.\boldsymbol{pf} := v$ ] Similar to the above case, the symbolic heap and the formula $(SH; \varphi)$ is first expanded to $((C'', S'', I'', c_{nil}, pf'', df'', pv''); \varphi'')$. When $u$ points to a valid location in the expanded heap, the mutation makes the $pf$ field of it updated: $pf''(pv''(u), pf)$ is updated with $pv''(v)$. Moreover, the heap domain ($C''$ and $S''$) is unmodified. The other field functions and the formula also remain the same.

[ $j := u.f$ ] Similar expanding to $((C'', S'', I'', c_{nil}, pf'', df'', pv''); \varphi'')$. The assignment makes $u$ points to the $f$ field of $v$, formally $pv''(u)$ is updated with a fresh integer variable $i$, representing the $f$ field of $v$. Hence, $\varphi'$ also extends $\varphi$ with the assertion $i = df''(pv''(v), f)$.

[ $u.f := j$ ] Similar to the $u.pf := v$ case. But again, a fresh integer variable $i$ which represents the current value of $j$, and extend $\varphi$ with the assertion $i = pv''(j)$.

[ $u :=$ `new` ] This statement makes $u$ points to a freshly allocated location, namely $n$. Since the new heap domain is an extension of the old one by adding a new node $n$, we know that $C' = C \uplus \{n\}$. By default, for $n$, each pointer field initially points to *nil*, each data field initially stores 0. These initial values are formalized by the definition of *pf*′ and *df*′, in which each $i_f$ represents the initial value of the $f$ field. the variable store $pv(u)$ is clearly updated with $n$. The remaining portion of the heap ($C$, $S$, *pf*′ and *df*′) is exactly the same as before. Moreover, $\varphi'$ is extended to assert that each data field value $i_f$ is 0, and the new node $n$ is different from all existing nodes in $C \cup S$.

[ $j := aexpr(\bar{k})$ ] The statement assigns the value of $aexpr(\bar{k})$, which is expressible in our logic, to $j$. Hence $pv(j)$ is updated with $i$, a fresh integer variable such that $i = aexpr[pv(\bar{k})/\bar{k}]$.

[ `assume` $bexpr(\bar{v}, \bar{j})$ ] The assumed condition *bexpr*, which can be expressed in our logic, must be true. So $\varphi'$ can simply extend $\varphi$ with a conjunct of *bexpr*, in which each variable is replaced with its value in *pv*. The heap is simply unmodified.

[ `return` $u$ ] When $u$ is defined in *pv*, the statement simply copy its value to the special variable *ret*. Formally the only modification to $SH'$ is updating $pv(ret)$ with $pv(u)$. The formula $\varphi'$ is unmodified.

[ `return` $j$ ] When $j$ is defined in *pv*, the statement simply copy its value to the special variable $ret_{int}$. So $pv(ret)$ in $SH'$ is updated with $i$, which is asserted as $i = pv(j)$ in $\varphi'$.

□

**Handling function calls.** Let us consider the statement $u := f(v, \bar{j})$ on the pair $(SH; \varphi)$. Let $f(w, \bar{k})$ be the function prototype and $\varphi_{post}$ its post-condition. If $pv(v)$ or any element of $pv(\bar{j})$ is undefined, the transformation is undefined. We also assume that the checking of the pre-condition for $f$ is successful; in particular, $pv(v)$ and all the nodes reachable from it are roots of trees.

Recall that certain nodes of the symbolic heap can be determined to point to trees (as discussed earlier). For any node $n \in C \cup S$, let us define

$reach_nodes(SH, n)$ to be the subset of $C \cup S$ that is reachable from $n$ in $Graph(SH)$. Let

$$\begin{aligned} N_C &= (reach_nodes(SH, pv(v)) \cap C) \setminus \{c_{nil}\} \\ N_S &= reach_nodes(SH, pv(v)) \cap S \end{aligned}$$

Intuitively, $N_C$ and $N_S$ are the concrete non-*nil* and the symbolic nodes affected by the call. Let $n_{ret}$ be the node returned by $f$. Let $N'$ be the set of nodes generated by the call: $N' = \{n_{ret}, pv(v)\}$ if $\varphi_{post}$ does not havoc $old_w$, and $N' = \{n_{ret}\}$ otherwise. The resulting symbolic heap is $(C', S', I', c_{nil}, pf', df', pv')$, where:

- $C' = C \setminus N_C$
- $S' = (S \setminus N_S) \cup N'$
- $I' = I$
- $pf' \mid_D = pf \mid_D$, and $pf'(n, pf)$ is undefined for all the pairs $(n, pf) \in (C' \setminus \{nil\} \times PF) \setminus D$, where $D \subseteq (C' \setminus \{nil\}) \times Dir$ is the set of pairs $(n', pf')$ such that $pf(n', pf') \in C' \cup S'$
- $df' = df \mid_{C' \setminus \{c_{nil}\} \times DF}$
- $pv' = pv[u \leftarrow n_{ret}]$

Intuitively, the concrete and symbolic nodes affected by the call are removed from the footprint (and get quantified in the $\text{D}\textsc{ryad}_{\text{tree}}$ formula), with the possible exception of $pv(v)$ (if $\varphi_{post}$ does not havoc $old_w$, $pv(v)$ becomes a symbolic node). The returned node is added to $S$. The *pf* and *df* functions are restricted to the new set of concrete nodes, and all the directions and program variables pointing to quantified nodes become undefined.

Let $\psi_{post}$ be the post-annotating formula in $\varphi_{post}$, we define the following formulas

$$\begin{aligned} \varphi_1 &\equiv \varphi[\overline{pre_call_r_n} / \overline{r^*(n)}] \\ &\wedge \bigwedge_{n \in N_C, r^*} \Big( pre_call_r_n = \hat{r}_{ind}(n) \Big) \\ \varphi_2 &\equiv \psi_{post}[pv(v)/old_w][\overline{pv(j)}/\overline{old_k}][n_{ret}/ret] \\ &\qquad [\overline{pre_call_r}_{pv(v)} / \overline{old_r^*}(pv(v))] \end{aligned}$$

where $n$ ranges over $N_C \cup N_S$, $r^*$ ranges over all the recursive predicates and functions; $pre_call_r_n$ are fresh logical variables; $\hat{r}_{ind}(n)$ is obtained from $r_{ind}(n)$ by replacing $n.pf$ with $pf(n, pf)$ and $n.f$ with $df(n, f)$ for all $pf \in PF, f \in DF$, and then by replacing $r^*(n')$ with $pre_call_r_{n'}$ for all $n' \in N_C \cup N_S$; $\overline{r^*(n)}$ is the vector of all the recursive predicates and functions on all $n \in N_C \cup N_S$. Intuitively, in $\varphi_1$ we add logical variables that capture the values of the recursive predicates and functions for the nodes affected by the call. In $\varphi_2$ we replace the program variables in the $\psi_{post}$ with the actual nodes and integer variables, and we replace the old version of the predicates and functions on $old_w$ with the variables capturing those values. Then the resulting formula is

$$\varphi' \equiv \varphi_1 \wedge \varphi_2$$

The case of $j := g(v, \bar{k})$ is similar.

**Example 5.2.5** (Search in AVL trees)**.** *The above procedure expands the symbolic heap and generates formulas, we present it working on the search routine of an AVL tree. Figure 5.3 shows the* `find` *routine, which searches in an AVL tree $t$ and returns* `true` *if a key $v$ is found. The pre-condition $\varphi_{pre}$, post-condition $\varphi_{post}$, and user-defined recursive sets and predicates are shown in Figure 5.4. In Figure 5.5, we present graphically how the symbolic heap evolves for a particular execution path of the routine. At each point of the basic block, we also formally show the updated symbolic heap SH and the corresponding formula $\varphi$.*

**Incorporating the postcondition.** Finally, after capturing the program state after execution *bb* by a pair $(SH; \varphi)$, we incorporate the post-condition $\varphi_{post}$, which contains the annotating formula $\psi$, and generate a verification condition. We should verify that:

**(1)** the nodes required by $\varphi_{post}$ to be tree roots are indeed tree roots; and

**(2)** For every pair of symbolic nodes $n_1$ and $n_2$, the reachable nodes from $n_1$ and $n_2$ are disjoint; and

**(3)** $(SH; \varphi_{vc}) \rightarrow \psi_{vc}$, that is, the constraints on the current states imply the constraints required by $\psi$.

```
int find(node t, int v)
{
  if (t = NULL) return false;
  tv := t.value;
  if (v = tv)
      return true;
  else if (v < tv) { w := t.left;
      r := find(w, v); }
  else { w := t.right;
      r := find(w, v); }
  return r;
}
```

Figure 5.3: AVL-find routine

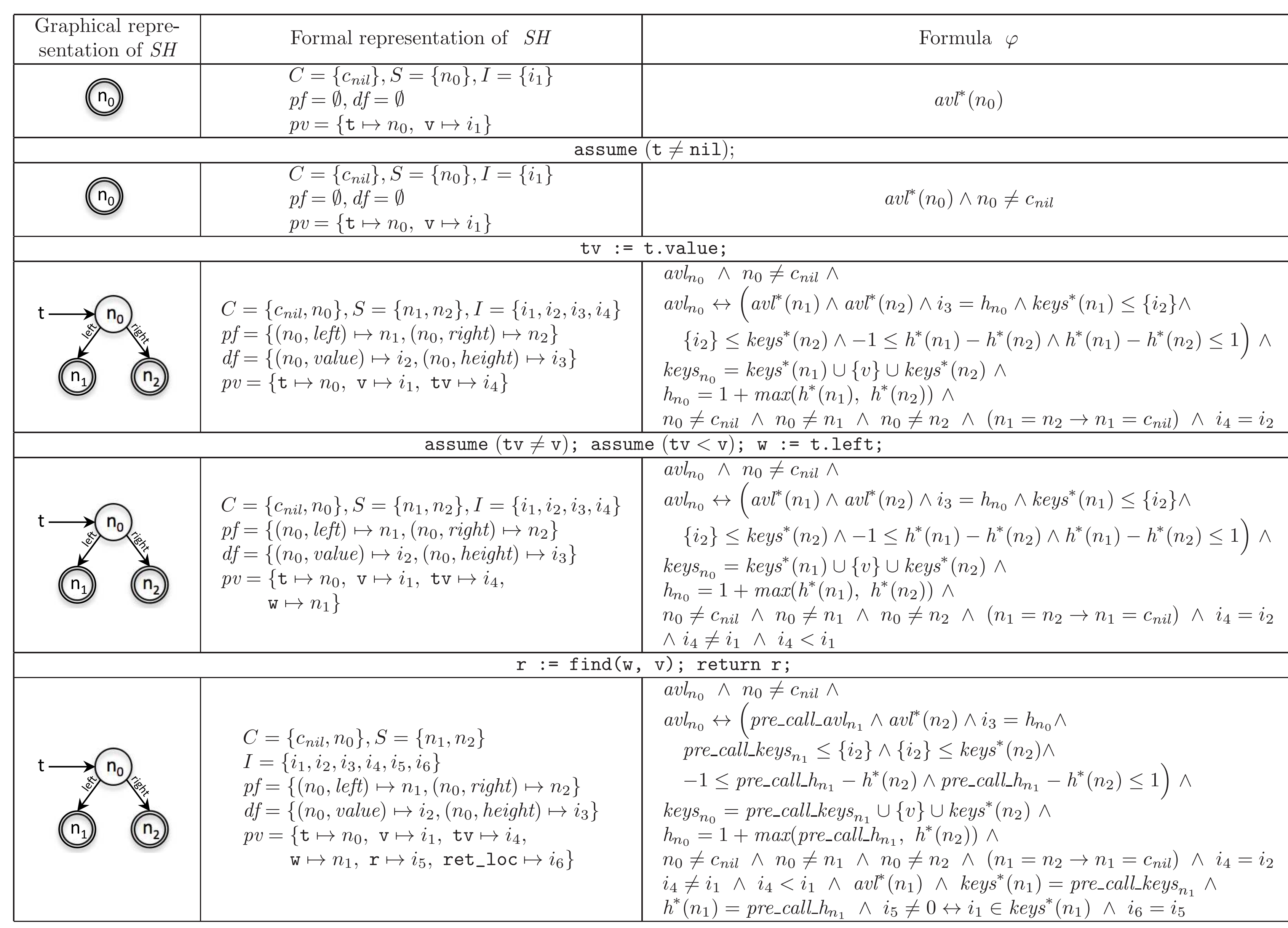
$$
\begin{aligned}
\varphi_{pre} &\equiv avl^*(\mathtt{t}) \\
\varphi_{post} &\equiv avl^*(\mathtt{t}) \wedge keys^*(\mathtt{t}) = keys^*(\mathtt{old_t}) \wedge h^*(\mathtt{t}) = h^*(\mathtt{old_t}) \wedge \\
&\qquad ret \neq 0 \leftrightarrow \mathtt{v} \in keys^*(\mathtt{t}) \\
avl^*(x) &\stackrel{def}{=} \mathtt{ite}(\ x = \mathtt{nil},\ \mathtt{true},\ avl_{ind}(x)\ ) \\
avl_{ind}(x) &\stackrel{def}{=} avl^*(x.left) \wedge avl^*(x.right) \wedge x.hight = h^*(x) \wedge \\
&\qquad keys^*(x.left) \leq \{v.value\} \wedge \{v.value\} \leq keys^*(x.right) \wedge \\
&\qquad -1 \leq h^*(x.left) - h^*(x.right) \wedge \\
&\qquad h^*(x.left) - h^*(x.right) \leq 1 \\
keys^*(x) &\stackrel{def}{=} \mathtt{ite}(\ x = \mathtt{nil},\ \emptyset,\ keys_{ind}(x)\ ) \\
keys_{ind}(x) &\stackrel{def}{=} keys^*(x.left) \cup \{x.value\} \cup keys^*(x.right) \\
h^*(x) &\stackrel{def}{=} \mathtt{ite}(\ x = \mathtt{nil},\ 0,\ h_{ind}(x)\ ) \\
h_{ind}(x) &\stackrel{def}{=} 1 + max(h^*(x.left),\ h^*(x.right))
\end{aligned}
$$

Figure 5.4: Pre/post conditions and recursive definition for AVL-find

The first two are readily checkable. The last one asserts that any concrete heap that corresponds to the symbolic heap $SH$ and satisfies $\varphi_{vc}$ must also satisfy $\psi_{vc}$. Checking the validity of this claim is non-trivial (undecidable) and we examine procedures that can soundly establish this.

## 5.3 A Decidable Fragment of $\textsc{Dryad}_{\text{tree}}$

Given verification conditions of the form $(SH; \varphi_{vc}) \rightarrow \psi_{vc}$, where $SH$ is a symbolic heap and $\varphi_{vc}$ and $\psi_{vc}$ are $\textsc{Dryad}_{\text{sep}}$ formulas, the validity problem



| Graphical representation of $SH$ | Formal representation of $SH$ | Formula $\varphi$ |
|---|---|---|
| $n_0$ | $C = \{c_{nil}\}, S = \{n_0\}, I = \{i_1\}$<br>$pf = \emptyset, df = \emptyset$<br>$pv = \{\mathtt{t} \mapsto n_0,\ \mathtt{v} \mapsto i_1\}$ | $avl^*(n_0)$ |
| `assume (t` $\neq$ `nil);` | | |
| $n_0$ | $C = \{c_{nil}\}, S = \{n_0\}, I = \{i_1\}$<br>$pf = \emptyset, df = \emptyset$<br>$pv = \{\mathtt{t} \mapsto n_0,\ \mathtt{v} \mapsto i_1\}$ | $avl^*(n_0) \wedge n_0 \neq c_{nil}$ |
| `tv := t.value;` | | |
| t, $n_0$, left, right, $n_1$, $n_2$ | $C = \{c_{nil}, n_0\}, S = \{n_1, n_2\}, I = \{i_1, i_2, i_3, i_4\}$<br>$pf = \{(n_0, left) \mapsto n_1, (n_0, right) \mapsto n_2\}$<br>$df = \{(n_0, value) \mapsto i_2, (n_0, height) \mapsto i_3\}$<br>$pv = \{\mathtt{t} \mapsto n_0,\ \mathtt{v} \mapsto i_1,\ \mathtt{tv} \mapsto i_4\}$ | $avl_{n_0}\ \wedge\ n_0 \neq c_{nil}\ \wedge$<br>$avl_{n_0} \leftrightarrow \Big(avl^*(n_1) \wedge avl^*(n_2) \wedge i_3 = h_{n_0} \wedge keys^*(n_1) \leq \{i_2\} \wedge$<br>$\{i_2\} \leq keys^*(n_2) \wedge -1 \leq h^*(n_1) - h^*(n_2) \wedge h^*(n_1) - h^*(n_2) \leq 1\Big)\ \wedge$<br>$keys_{n_0} = keys^*(n_1) \cup \{v\} \cup keys^*(n_2)\ \wedge$<br>$h_{n_0} = 1 + max(h^*(n_1),\ h^*(n_2))\ \wedge$<br>$n_0 \neq c_{nil}\ \wedge\ n_0 \neq n_1\ \wedge\ n_0 \neq n_2\ \wedge\ (n_1 = n_2 \rightarrow n_1 = c_{nil})\ \wedge\ i_4 = i_2$ |
| `assume (tv` $\neq$ `v); assume (tv` $<$ `v); w := t.left;` | | |
| t, $n_0$, left, right, $n_1$, $n_2$ | $C = \{c_{nil}, n_0\}, S = \{n_1, n_2\}, I = \{i_1, i_2, i_3, i_4\}$<br>$pf = \{(n_0, left) \mapsto n_1, (n_0, right) \mapsto n_2\}$<br>$df = \{(n_0, value) \mapsto i_2, (n_0, height) \mapsto i_3\}$<br>$pv = \{\mathtt{t} \mapsto n_0,\ \mathtt{v} \mapsto i_1,\ \mathtt{tv} \mapsto i_4,$<br>$\mathtt{w} \mapsto n_1\}$ | $avl_{n_0}\ \wedge\ n_0 \neq c_{nil}\ \wedge$<br>$avl_{n_0} \leftrightarrow \Big(avl^*(n_1) \wedge avl^*(n_2) \wedge i_3 = h_{n_0} \wedge keys^*(n_1) \leq \{i_2\} \wedge$<br>$\{i_2\} \leq keys^*(n_2) \wedge -1 \leq h^*(n_1) - h^*(n_2) \wedge h^*(n_1) - h^*(n_2) \leq 1\Big)\ \wedge$<br>$keys_{n_0} = keys^*(n_1) \cup \{v\} \cup keys^*(n_2)\ \wedge$<br>$h_{n_0} = 1 + max(h^*(n_1),\ h^*(n_2))\ \wedge$<br>$n_0 \neq c_{nil}\ \wedge\ n_0 \neq n_1\ \wedge\ n_0 \neq n_2\ \wedge\ (n_1 = n_2 \rightarrow n_1 = c_{nil})\ \wedge\ i_4 = i_2$<br>$\wedge\ i_4 \neq i_1\ \wedge\ i_4 < i_1$ |
| `r := find(w, v); return r;` | | |
| t, $n_0$, left, right, $n_1$, $n_2$ | $C = \{c_{nil}, n_0\}, S = \{n_1, n_2\}$<br>$I = \{i_1, i_2, i_3, i_4, i_5, i_6\}$<br>$pf = \{(n_0, left) \mapsto n_1, (n_0, right) \mapsto n_2\}$<br>$df = \{(n_0, value) \mapsto i_2, (n_0, height) \mapsto i_3\}$<br>$pv = \{\mathtt{t} \mapsto n_0,\ \mathtt{v} \mapsto i_1,\ \mathtt{tv} \mapsto i_4,$<br>$\mathtt{w} \mapsto n_1,\ \mathtt{r} \mapsto i_5,\ \mathtt{ret_loc} \mapsto i_6\}$ | $avl_{n_0}\ \wedge\ n_0 \neq c_{nil}\ \wedge$<br>$avl_{n_0} \leftrightarrow \Big(pre_call_avl_{n_1} \wedge avl^*(n_2) \wedge i_3 = h_{n_0} \wedge$<br>$pre_call_keys_{n_1} \leq \{i_2\} \wedge \{i_2\} \leq keys^*(n_2) \wedge$<br>$-1 \leq pre_call_h_{n_1} - h^*(n_2) \wedge pre_call_h_{n_1} - h^*(n_2) \leq 1\Big)\ \wedge$<br>$keys_{n_0} = pre_call_keys_{n_1} \cup \{v\} \cup keys^*(n_2)\ \wedge$<br>$h_{n_0} = 1 + max(pre_call_h_{n_1},\ h^*(n_2))\ \wedge$<br>$n_0 \neq c_{nil}\ \wedge\ n_0 \neq n_1\ \wedge\ n_0 \neq n_2\ \wedge\ (n_1 = n_2 \rightarrow n_1 = c_{nil})\ \wedge\ i_4 = i_2$<br>$i_4 \neq i_1\ \wedge\ i_4 < i_1\ \wedge\ avl^*(n_1)\ \wedge\ keys^*(n_1) = pre_call_keys_{n_1}\ \wedge$<br>$h^*(n_1) = pre_call_h_{n_1}\ \wedge\ i_5 \neq 0 \leftrightarrow i_1 \in keys^*(n_1)\ \wedge\ i_6 = i_5$ |

Figure 5.5: Expanding the symbolic heap and generating the formulas

is in general undecidable. However, a decision procedure is desirable in many situations. For example, when a program does not satisfy its specifications, the decision procedure could *disprove* the program, and confirm it with a counterexample, which helps programmers debug the code. In this section, we identify a decidable, yet practically useful fragment, called $\text{DRYAD}^{\text{dec}}_{\text{tree}}$. We present that if $\varphi_{vc}$ and $\psi_{vc}$ belong to $\text{DRYAD}^{\text{dec}}_{\text{tree}}$, the validity problem is decidable by showing it can be expressed in $\text{STRAND}^{syn}_{dec}$, an expressive logic that combines theories of trees with arithmetic, and that admits efficient decision procedures (see Chapter 4).

### 5.3.1 Definition of $\text{DRYAD}^{\text{dec}}_{\text{tree}}$

Let us fix a set of directions $Dir$ and a set of data-fields $DF$. $\text{DRYAD}^{\text{dec}}_{\text{tree}}$ does not only restrict the syntax of $\text{DRYAD}_{\text{sep}}$, but also restricts the recursive integers/sets/multisets/predicates that the users can define. We first describe the ways allowed in $\text{DRYAD}^{\text{dec}}_{\text{tree}}$ to define recursions as follows:

- Recursive integers are disallowed;
- For each data field $f \in DF$, a recursive set of integers $fs^*$ can be defined as

$$fs^*(x) = \texttt{ite}\Big(x = \texttt{nil},\ \emptyset,\ \{x.f\} \cup \bigcup_{pf \in PF} fs^*(x.pf)\Big)$$

- For each data field $f \in DF$, a recursive multiset of integers $fms^*$ can be defined as

$$fms^*(x) = \texttt{ite}\Big(x = \texttt{nil},\ \emptyset_m,\ \{x.f\}_m \cup \bigcup_{pf \in PF} fms^*(x.pf)\Big)$$

- Recursive predicates can be defined in the form of

$$p^*(x) = \texttt{ite}\Big(x = \texttt{nil},\ \texttt{true},\ \varphi_p(x) \wedge \bigwedge_{pf \in PF} p^*(x.pf)\Big)$$

where $\varphi_p(x)$ is a *local formula* with $x$ as the only free variable. The syntax of local formulas is presented in Figure 5.6. Intuitively, $p^*(x)$ is evaluated to true if and only if every node $y$ in the subtree of $x$ satisfies the local formula

$\varphi_p(y)$, which can be determined by simply accessing the data fields of $y$ and evaluating the recursive sets/multisets for the children of $y$.

The exclusion of recursive integers prevents us from expressing heights/-cardinalities (which are required by a considerable number of data structures). There are however interesting algorithms on inductive data structures, like binary heaps, binary search trees and treaps, whose verification can be expressed in $\text{DRYAD}^{\text{dec}}_{\text{tree}}$. For example, to describe treaps, let $DF$ be $\{key, priority\}$ and $PF$ be $\{l, r\}$, then we can describe the recursive predicate $treap^*$ as follows:

$$
\begin{aligned}
treap^*(x) = \texttt{ite}\Big(x = \texttt{nil},\ \texttt{true},\ & treap^*(x.l) \wedge treap^*(x.r) \\
& \wedge\ keys^*(x.l) \leq \{x.key\} \leq keys^*(x.r) \\
& \wedge\ \{x.priority\} \leq priorities^*(x.l) \\
& \wedge\ \{x.priority\} \leq priorities^*(x.l)\Big)
\end{aligned}
$$

With a set of recursive predicates defined as above, the syntax of $\text{DRYAD}^{\text{dec}}_{\text{tree}}$ is presented in Figure 5.7. Intuitively, $\text{DRYAD}^{\text{dec}}_{\text{tree}}$ does not allow to refer to any child or any data field, for any location, i.e., terms of the form $lt.pf$ or $lt.f$ are disallowed. Difference operations and subset relations between sets/multisets are also disallowed. For example, for the `insert` routine for treaps, one can still express that the returned tree is still a treap. However, one cannot state that the set of keys has the expected property.

$$
\begin{array}{rcl}
\varphi & ::= & \psi \mid sit_1 \leq sit_2 \mid msit_1 \leq msit_2 \\
& & \mid it \notin sit \mid it \notin msit \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \\
\psi & ::= & \texttt{true} \mid it_1 \leq it_2 \mid \neg\psi \\
it & ::= & c \mid x.f \mid it_1 + it_2 \mid it_1 - it_2 \\
sit & ::= & \emptyset \mid \{it\} \mid fs^*(x.dir) \mid sit_1 \cup sit_2 \mid sit_1 \cap sit_2 \\
msit & ::= & \emptyset_m \mid \{it\}_m \mid fms^*(x.dir) \mid msit_1 \cup msit_2 \mid msit_1 \cap msit_2
\end{array}
$$

$$
pf \in PF \qquad f \in DF \qquad c\ :\ Int \text{ Constant}
$$

Figure 5.6: Syntax of local formulas $\varphi_p(x)$

$$q \in \textit{Boolean}\text{ Vars} \qquad x \in \textit{Loc}\text{ Vars} \qquad j \in \textit{Int}\text{ Vars}$$

$$S \in \mathcal{MS}(\textit{Int})\text{ Vars} \qquad MS \in \mathcal{MS}(\textit{Int})\text{ Vars} \qquad c : \textit{Int}\text{ Constant}$$

$$\begin{array}{rcl}
\textit{Int}\text{ Term}: \ it, it_1, it_2, \ldots & ::= & c \mid j \mid it_1 + it_2 \mid it_1 - it_2 \\
\mathcal{S}(\textit{Int})\text{ Term}: \ sit, sit_1, sit_2, \ldots & ::= & \emptyset \mid S \mid \{it\} \mid \textit{fs}^*(x) \mid \\
 & & sit_1 \cup sit_2 \mid sit_1 \cap sit_2 \\
\mathcal{MS}(\textit{Int})\text{ Term}: \ msit, msit_1, \ldots & ::= & \emptyset_m \mid MS \mid \{it\}_m \mid \textit{fms}^*(x) \mid \\
 & & msit_1 \cup msit_2 \mid msit_1 \cap msit_2 \\
\text{Formula}: \ \varphi, \varphi_1, \varphi_2, \ldots & ::= & \texttt{true} \mid q \mid p^*(x) \mid x_1 = x_2 \mid x = \texttt{nil} \\
 & & \mid it_1 \le it_2 \mid sit_1 \le sit_2 \\
 & & \mid msit_1 \le msit_2 \\
 & & \mid it \in sit \mid it \in msit \mid \neg\varphi \mid \varphi_1 \vee \varphi_2
\end{array}$$

Figure 5.7: Syntax of $\text{DRYAD}^{\text{dec}}_{\text{tree}}$

### 5.3.2 Proof of Decidability

It is clear that $(SH; \varphi_{vc}) \to \psi_{vc}$ is valid if and only if $(SH; \varphi_{vc} \wedge \neg\psi_{vc})$ is unsatisfiable. We now prove that the satisfiability of $(SH; \varphi)$ is decidable, where $\varphi \in \text{DRYAD}^{\text{dec}}_{\text{tree}}$. The idea is, to characterize $SH$ with variable assignments as a class of recursively-defined data structures $\mathcal{R}_{SH}$, and characterize $\varphi$ as a STRAND formula $strand(\varphi)$, then the satisfiability of $(SH; \varphi)$ is reduced to the satisfiability of of $strand(\varphi)$ over $\mathcal{R}_{SH}$. We outline the translation as follows.

Let $SH = (C, S, I, c_{nil}, pf, df, pv)$. Consider a concrete heap $CH$ corresponding to $SH$, which consists of two portions: the portion that $SH$ is isomorphic to, including $h(c)$ for every $c \in C$ and the tree under $s$ for every $s \in S$, and the portion not represented in $SH$. With the satisfiability preserved, we can assume that there is a particular node in the the second portion such that any pointer to/from the second portion is to/from this particular node. Let *Vars* be a finite but large enough set of variables of various sorts (Loc/Int/Set/MSet). $CH$ with variable assignments for *Vars* can be encoded as a tree with data fields as follows. The root of the tree has three children:

- the subtree of the leftmost child ($T_1$) models the portion of $CH$ that $SH$ is isomorphic to;
- the subtree of the middle child ($T_2$) models the portion of $CH$ that is not represented by $SH$;

- the subtree of the rightmost child ($T_3$) models the non-location variable assignments for *Vars*.

$T_1$ has two children: the subtree of the left/right child models concrete/symbolic nodes, respectively. The subtree of the left child consists of only the nodes in the leftmost path, which is of length $|C|$, such that each concrete node in $C$ is modeled as a specific node in the path. In the subtree of the right child, the leftmost path is of length $|S|$. For each symbolic node in $S$, the corresponding tree in *CH* is modeled as the right subtree of a specific node in the path. $T_2$ is of arbitrary shape, but each node in $T_2$ should corresponds to a node in *CH* that is not represented by *SH*.

The leftmost path of $T_3$ is of length $|Vars|$, such that each non-location variable assignment is modeled as the right subtree of a specific node in the path. For example, let $i \mapsto 1$ be an integer-variable assignment, then its corresponding subtree consists of a single node $n$ with $n.f = 1$; let $S \mapsto \{1, 2, 3\}$ be a set-variable assignment, then its corresponding subtree consists of three nodes, such that the set of integers in their $f$ fields is $\{1, 2, 3\}$.

Hence, the class concrete heaps corresponding to *SH* can be represented as a class of recursively-defined data structures, called $\mathcal{R}_{SH}$. We introduce an elastic relation $\rightarrow^*$. $x \rightarrow^* y$ means that $y$ is a descendent of $x$. Moreover, for each variable $v \in Vars$, we introduce a new predicate $p_v$, such that $p_v(y)$ is evaluated to true if and only if $y$ is the node that models $v$. These predicates are also definable in $\mathcal{R}_{SH}$.

With the above setting, we now show that there is a mapping *strand* from $\text{DRYAD}^{\text{dec}}_{\text{tree}}$ formulas to $\text{STRAND}^{syn}_{dec}$ formulas over $\mathcal{R}_{SH}$, such that if a concrete heap satisfies $\varphi$ with a set of variable assignments, its corresponding model in $\mathcal{R}_{SH}$ satisfies $strand(\varphi)$, and vice versa. Since efficient decision procedures for $\text{STRAND}^{syn}_{dec}$ exist (see Chapter 4), the satisfiability of $\text{DRYAD}^{\text{dec}}_{\text{tree}}$ is also decidable. We first split the appearances of integer terms in set/multiset terms by substituting every occurrence of $\{it\}$ in $\{it\}$ or $\{it\}_m$ in $\varphi$ with the newly introduced integer variable $i_{it}$, and adding a conjunct $i_{it} = it$. The resulting formula is denoted as $\varphi_{split}$. Then for each variable $v$ appears in $\varphi$, we introduce an existentially quantified location variable with the same name in the mapped $\text{STRAND}^{syn}_{dec}$ formula, and add a conjunct $p_v(v)$, which intuitively says $v$ does appear in the heap (in $T_1$ or $T_2$). Let the occurred

$$
\begin{array}{rcl}
map(\texttt{true}) & = & \texttt{true} \\
map(q) & = & q.value = 0 \\
map(p^*(x)) & = & \forall w \;.\; \big(\; x \not\to^* w \vee map(\varphi_p(w)) \;\big) \\
map(x_1 = x_2) & = & x_1 = x_2 \\
map(x = \texttt{nil}) & = & x = c_{nil} \\
map(it_1 \leq it_2) & = & it_1[\vec{i}.f/\vec{i}] \leq it_2[\vec{i}.f/\vec{i}] \\
map(sit_1 \leq sit_2) & = & \forall w_1, \ldots, w_{m+n} \;.\; \big(\; w_1.f \leq w_{m+1}.f \;\vee \\
& & \qquad\qquad \neg wtns(sit_1)(w_1, \ldots, w_m) \;\vee \\
& & \qquad\qquad \neg wtns(sit_2)(w_{m+1}, \ldots, w_{m+n}) \;\big) \\
map(msit_1 \leq msit_2) & = & \forall w_1, \ldots, w_{m+n} \;.\; \big(\; w_1.f \leq w_{m+1}.f \;\vee \\
& & \qquad\qquad \neg wtns(msit_1)(w_1, \ldots, w_m) \;\vee \\
& & \qquad\qquad \neg wtns(msit_2)(w_{m+1}, \ldots, w_{m+n}) \;\big) \\
map(it \in sit_1) & = & \exists w_1, \ldots, w_m \;.\; \big(\; it[\vec{i}.f/\vec{i}] = w_1.f \;\wedge \\
& & \quad wtns(sit_1)(w_1, \ldots, w_m) \;\big) \\
map(it \in msit_1) & = & \exists w_1, \ldots, w_m \;.\; \big(\; it[\vec{i}.f/\vec{i}] = w_1.f \;\wedge \\
& & \quad wtns(msit_1)(w_1, \ldots, w_m) \;\big) \\
map(\neg\varphi) & = & \neg map(\varphi) \\
map(\varphi_1 \wedge \varphi_2) & = & map(\varphi_1) \wedge map(\varphi_2) \\
map(\varphi_1 \vee \varphi_2) & = & map(\varphi_1) \vee map(\varphi_2)
\end{array}
$$

$$
\begin{array}{rcl}
wtns(\emptyset) & = & \texttt{false} \\
wtns(\emptyset_m) & = & \texttt{false} \\
wtns(\{i\}) & = & w_1 = i \\
wtns(\{i\}_m) & = & w_1 = i \\
wtns(fs^*(x)) & = & x \to^* w_1 \\
wtns(fms^*(x)) & = & x \to^* w_1 \\
wtns(sit_1 \cup sit_2) & = & wtns(sit_1)(w_1, \ldots, w_m) \vee wtns(sit_2)(w_1, \ldots, w_n) \\
wtns(msit_1 \cup msit_2) & = & wtns(msit_1)(w_1, \ldots, w_m) \vee wtns(msit_2)(w_1, \ldots, w_n) \\
wtns(sit_1 \cap sit_2) & = & wtns(sit_1)(w_1, \ldots, w_m) \;\wedge \\
& & wtns(sit_2)(w_{m+1}, \ldots w_{m+n}) \;\wedge w_1.f = w_{m+1}.f \\
wtns(msit_1 \cap msit_2) & = & wtns(msit_1)(w_1, \ldots, w_m) \;\wedge \\
& & wtns(msit_2)(w_{m+1}, \ldots w_{m+n}) \;\wedge w_1.f = w_{m+1}.f
\end{array}
$$

where $m = width(sit_1)$ or $width(msit_1)$
$n = width(sit_2)$ or $width(msit_2)$
$\varphi_p(x)$ is the local formula for $p^*$

$$
\begin{array}{rcl}
width(\emptyset) & = & 0 \\
width(\emptyset_m) & = & 0 \\
width(\{i\}) & = & 1 \\
width(\{i\}_m) & = & 1 \\
width(fs^*(x)) & = & 1 \\
width(fms^*(x)) & = & 1 \\
width(sit_1 \cup sit_2) & = & max\big(width(sit_1), width(sit_2)\big) \\
width(msit_1 \cup msit_2) & = & max\big(width(msit_1), width(msit_2)\big) \\
width(sit_1 \cap sit_2) & = & width(sit_1) + width(sit_2) \\
width(msit_1 \cap msit_2) & = & width(msit_1) + width(msit_2)
\end{array}
$$

Figure 5.8: Inductive definition of $map(\varphi)$

variables be $\vec{v}$, then

$$strand(\varphi) \;=\; \exists\vec{v}.\Big(map(\varphi_{split}) \wedge \bigwedge_{v\in\vec{v}} p_v(v)\Big)$$

where $map(\varphi_{split})$ is defined inductively as shown in Figure 5.8. For each set $sit$ or multiset $msit$, there is an auxiliary formula $wtns(sit)/wtns(msit)$ with free variables $\vec{x}$, saying that $\vec{x}$ witnesses an element in $sit/msit$. We denote $|\vec{x}|$ as $width(sit)/width(msit)$, which can be computed inductively. For simplicity, we assume that $f$ is the only data field.

**Proposition 5.3.1.** *For any* $\text{DRYAD}^{dec}_{tree}$ *formula* $\varphi$*,* $strand(\varphi)$ *is a* $\text{STRAND}^{syn}_{dec}$ *formula.*

*Proof.* Note that $\varphi_{split}$ is quantifier-free, and the translation of each atomic formula in $\varphi_{split}$ introduces only universal or existential quantifiers, $map(\varphi_{split})$ has a $\exists\forall$-prefix after converting to the prenex normal form. Similarly, for each $v$, $p_v(v)$ also admits the $\exists\forall$-prefix. Hence $strand(\varphi)$ is a STRAND formula. Moreover, the only structural relation introduced in the translation is the reachability $\rightarrow^*$, which is elastic. Then by the definition, $strand(\varphi)$ falls in the fragment $\text{STRAND}^{syn}_{dec}$. ☐

Note that for any recursive predicate $p^*$ with local formula $\varphi_p$, $map(\varphi_p)$ is a universal $\text{STRAND}^{syn}_{dec}$ formula. Then for any $\text{DRYAD}^{\text{dec}}_{\text{tree}}$ formula $\varphi$, $map(\varphi)$ is well defined: if $\varphi$ is an atomic formula, by definition it is clear that $map(\varphi)$ is an existential $\text{STRAND}^{syn}_{dec}$ formulas or a universal $\text{STRAND}^{syn}_{dec}$ formula; if $\varphi$ is a Boolean combination of atomic formulas, $map(\varphi)$ is a Boolean combination of corresponding existential/universal $\text{STRAND}^{syn}_{dec}$ formulas, and is still a $\text{STRAND}^{syn}_{dec}$ formula.

**Theorem 5.3.2.** *For any symbolic heap SH and any* $\text{DRYAD}^{dec}_{tree}$ *formula* $\varphi$*,* $(SH;\varphi)$ *is satisfiable if and only if the* $\text{STRAND}^{syn}_{dec}$ *formula* $strand(\varphi)$ *is satisfiable over* $\mathcal{R}_{SH}$*.*

*Proof.* The equisatisfiability is also built inductively by investigating the semantics of the mapping formula for each atomic case. Notice that the $\text{DRYAD}^{\text{dec}}_{\text{tree}}$ formulas are quantifier-free, and the translation preserves all Boolean operators, it suffices to discuss the translation of each case of atomic $\text{DRYAD}^{\text{dec}}_{\text{tree}}$ formulas.

[ `true` ] Obvious.

[ $x_1 = x_2$ ] The translation is identical, so $x_1 = x_2$ is satisfied over $SH$ iff both $x_1$ and $x_2$ are real nodes in the model of $\mathcal{R}_{SH}$, namely $p_v(x_1) \wedge p_v(x_2)$, and $x_1 = x_2$.

[ $x_1 = \texttt{nil}$ ] Similar to the above case.

[ $q$ ] The Boolean variable $q$ in $\text{Dryad}^{\text{dec}}_{\text{tree}}$ is interpreted as a node with integer value of either 0 (true) or 1 (false). Hence $q$ is satisfied in $\text{Dryad}^{\text{dec}}_{\text{tree}}$ iff $q.value = 0$ is satisfied in $\text{Strand}^{syn}_{dec}$.

[ $p^*(x)$ ] Since $p^*(x)$ can only be defined in a restricted form in $\text{Dryad}^{\text{dec}}_{\text{tree}}$ with respect to a local formula $\varphi_p(x)$, intuitively $p^*(x)$ holds if and only if for every node $y$ reachable from $x$, $\varphi_p(y)$ holds. This semantics can be expressed using universal quantifiers in $\text{Strand}^{syn}_{dec}$: $\forall w \;.\; \big(\, x \not\rightarrow^* w \vee map(\varphi_p(w)) \,\big)$ (based on the hypothesis that $map(\varphi_p(w))$ is sound).

[ $\boldsymbol{it}_1 \leq \boldsymbol{it}_2$ ] When both $it_1$ and $it_2$ are integer terms, the translation is similar to the above identity cases. However, each integer variable $i$ is replaced with the data field of the same-name location variable $i$, namely $i.f$.

[ **other formulas containing set/multiset terms** ] When set/multiset terms are involved, there are basically two kinds of relations: $\leq$ and $\in$. Note that $\leq$ can be encoded into $\in$ with quantifiers. For example, $sit_1 \leq sit_2$ can be intuitively translated to

$$\forall w_1, w_2. \; (w_1 \leq w_2 \vee w_1 \notin sit_1 \vee w_2 \notin sit_2)$$

Now to encode $\in$ into Strand, an auxiliary mapping $wtns$ is used. Given an integer set term $sit$ with the width $m$, $wins(sit)$ is a formula with m free integer variables: $w_1, \ldots, w_m$. Intuitively, $wins(sit)$ guesses the auxiliary variables $w_2, \ldots, w_m$ and encodes the formula $w_1 \in sit$. The soundness of such encodings can be proved by induction on the structure of $sit$. It is worth mentioning that in the definition of $wtns(sit_1 \cap sit_1)$, $w_1$ and $w_{m+1}$ are the witnesses of $sit_1$ and $sit_2$, respectively, and the conjunct $w_1.f = w_{m+1}.f$ guarantees that $w_1$ witnesses both $sit_1$ and $sit_2$. In a similar way $wtns(msit_1 \cap msit_1)$ is defined.

□

The above reduction immediately implies the decidability of $\text{DRYAD}^{\text{dec}}_{\text{tree}}$.

**Corollary 5.3.3.** *The satisfiability of* $\text{DRYAD}^{dec}_{tree}$ *is decidable.*

*Proof.* By Theorem 5.3.2, the satisfiability of $\text{DRYAD}^{\text{dec}}_{\text{tree}}$ reduces to the satiability of $\text{STRAND}^{syn}_{dec}$, which is decidable, by Theorem 4.3.4. □

## 5.4 Formula Abstraction

Overall, $\text{DRYAD}^{\text{dec}}_{\text{tree}}$ is the most powerful fragment of $\text{DRYAD}_{\text{tree}}$ that we could find that embeds into a known decidable logic, like $\text{STRAND}^{syn}_{dec}$. However, it is not powerful enough for the heap verification questions that we would like to solve. This motivates the second proof tactics for the natural proof strategy: formula abstraction. We turn to the abstraction schemes for unrestricted verification conditions, and show how to soundly reduce verification conditions to a validity problem of quantifier-free theories of sets/multisets of integers, which is decidable using state-of-the-art SMT solvers.

Before describing how the formula abstraction technique works for $\text{DRYAD}_{\text{tree}}$, we roughly describe its idea and motivation as below.

### 5.4.1 Motivation

When reasoning with formulas that have recursively defined terms, which can be unrolled forever, a key idea is to use *formula abstraction* that makes the terms *uninterpreted*. Intuitively, the idea is to replace recursively defined predicates, sets, etc. by uninterpreted Boolean values, uninterpreted sets, etc.

The idea of formula abstraction is extremely natural, and utilized very often in manual proofs. For instance, let us consider a binary search tree (BST) search routine searching for a key $k$ on the root node $x$. The verification condition of a path of this program, typically, would require checking:

$$(bst(x) \wedge k \in keys(x) \wedge k < x.key) \Rightarrow (k \in keys(x.left))$$

where $bst()$ is a recursive predicate defined on trees that identifies binary search trees, and $keys()$ is a recursively defined set that collects the multiset of keys under a node. Unrolling the definitions of $keys()$ and $bst()$ gives the following formula (below, $i < S$ means that $i$ is less than every element in $S$):

$$bst(x.left) \wedge bst(x.right) \wedge keys(x.left) < x.key \wedge keys(x.right) > x.key \;\wedge$$
$$k \in (keys(x.left) \cup keys(x.right) \cup \{x.key\}) \wedge k < x.key) \Rightarrow (k \in keys(x.left))$$

Now, while the above formula is quite complex, involving recursive definitions that can be unrolled *ad infinitum*, we can prove its validity soundly by viewing $bst()$ and $keys()$ as *uninterpreted* functions that map locations to Booleans and sets, respectively. Doing this gives (modulo some renaming and modulo theory of equality):

$$\big(b_1 \wedge b_2 \wedge K_1 \leq xkey \wedge K_2 > xkey \wedge k \in (K_1 \cup K_2 \cup \{xkey\}) \wedge k < xkey\big) \Rightarrow (k \in K_1)$$

Note that the above formula is a quantifier-free formula over integers and multisets of integers, and furthermore is valid (since $k < xkey$ and $K_2 > xkey$, $k$ must be in $K_1$). Validity of quantifier-free formulas over sets/multisets of integers with addition is *decidable* (they can be translated to quantifier-free formulas over integers and uninterpreted functions), and can be solved using SMT solvers efficiently. Consequently, we can prove that the verification condition is valid, completely automatically. Note that formula abstraction is *sound* but incomplete.

This idea has been explored in the literature. For example, Suter et al. [72] have proposed abstraction schemes for algebraic data-types that soundly (but not necessarily completely) transform logical validity into much simpler decidable problems using formula abstractions, and developed mechanisms for proving *functional programs* correct.

### 5.4.2 Formula Abstraction for $\text{DRYAD}_{\text{tree}}$

To prove a verification condition $(SH; \varphi) \rightarrow \psi$ using formula abstraction, we drop $SH$, and we replace recursive predicates on symbolic nodes by uninter-

preted Boolean functions, replace recursive integer functions as uninterpreted functions that map nodes to integers, and replace recursive set/multiset functions with functions that map nodes to arbitrary sets and multisets. Let us denote the uninterpreted definition using the same name without $^*$. For example $avl^*$ is replaced with $avl$. Notice that the constraints regarding the concrete and symbolic nodes in $SH$ were already added to $\varphi$, during the construction of the verification condition. The formula resulting via abstraction is a formula $\varphi_{\text{abs}} \rightarrow \psi_{\text{abs}}$ such that: (1) if $\varphi_{\text{abs}} \rightarrow \psi_{\text{abs}}$ is valid, then so is $(SH; \varphi) \rightarrow \psi$ (the converse may not necessarily be true); (2) checking $\varphi_{\text{abs}} \rightarrow \psi_{\text{abs}}$ is decidable, and in fact can be reduced to `QF_UFLIA`, the quantifier-free theory of uninterpreted functions and arithmetic.

**Proposition 5.4.1** (Soundness). *Let $SH$ be a symbolic heap and $\varphi$, $\psi$ be* DRYAD$_{tree}$ *formulas, and let $\varphi_{abs}$ and $\psi_{abs}$ be the uninterpreted version of $\varphi$ and $\psi$, respectively. Then if $\varphi_{abs} \rightarrow \psi_{abs}$ is valid, then so is $(SH; \varphi) \rightarrow \psi$.*

*Proof.* Prove by contradiction. Assume $(SH; \varphi) \rightarrow \psi$ is not valid, then there is a concrete heap $CH$ that corresponds to $SH$ and satisfied $\varphi \wedge \neg\psi$. Then we can construct a model $CH'$ satisfying $\varphi_{\text{abs}} \wedge \neg\psi_{\text{abs}}$ as well. $CH'$ consists of the same set of locations $N$ that form $CH$, and for each recursive definition $r$, simply interpret it as the same way that $r^*$ is interpreted in $CH$. The counterexample $CH'$ contradicts the validity of $\varphi_{\text{abs}} \rightarrow \psi_{\text{abs}}$ and concludes the proof. □

We point out that while the formula abstraction is *sound*, it is not *complete*. For example, consider two recursively defined functions: height of a binary tree, $height^*(t)$, and size of the same binary tree (the number of nodes in the respective tree), $size^*(t)$. The two functions respect the following relationship: $size^*(t) < 2^{height^*(t)}$, for any tree $t$. For a symbolic node $t$, the formula $\neg(height^*(t) = 2 \;\wedge\; size^*(t) = 10)$ is valid. However, the abstracted formula is trivially invalid, as the height and the size are abstracted into unrelated uninterpreted functions.

### 5.4.3 Decision procedure for ordered sets and multisets

The validity of the abstracted formula $\varphi_{\text{abs}} \rightarrow \psi_{\text{abs}}$ over the theory of uninterpreted function, linear arithmetic, and sets and multisets of integers, is

decidable. The fact that the quantifier free theory of ordered sets is decidable is well known. In fact, Kuncak et al. [45] showed that the quantifier-free theory of sets with cardinality constraints is NP-complete. Since we do not need cardinality constraints, we use a slightly simplified decision procedure that reduces formulas with sets/multisets using uninterpreted functions that capture the characteristic functions associated with these sets/multisets, which is described as follows.

In this section, we describe a simple decision procedure for quantifier free ordered sets, and we show how to adapt this decision procedure for quantifier free ordered multisets.

Formally, let $(\mathcal{D}, \leq)$ be a domain with a partial order relation. The syntax for formulas of ordered sets over the domain $\mathcal{D}$ is given below:

$$\begin{array}{rcl} \varphi & ::= & \texttt{true} \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid \neg\varphi \\ & \mid & e_1 = e_2 \mid e_1 \leq e_2 \mid e \in S \mid S_1 \subseteq S_2 \mid S_1 \leq S_2 \\ S & ::= & \emptyset \mid \{e\} \mid S_1 \cup S_2 \mid S_1 \cap S_2 \mid S_1 \setminus S_2 \end{array}$$

where $e,\ e_1,\ e_2$ are constants in $\mathcal{D}$. We interpret set membership and set inclusion in the usual way. The interpretation of the order on sets is given by

$$S_1 \leq S_2 \Leftrightarrow (\forall x_1,\ x_2 \in \mathcal{D})\ (x_1 \in S_1 \wedge x_2 \in S_2) \Rightarrow x_1 \leq s_2$$

Note that the ordering on sets is not a partial order relation. In particular, for any sets $S_1$ and $S_2$, both $S_1 \leq \emptyset$ and $\emptyset \leq S_2$ hold trivially, regardless of whether $S_1 \leq S_2$. Set equality $S_1 = S_2$ is just syntactic sugar for $S_1 \subseteq S_2 \wedge S_2 \subseteq S_1$

We associate to each set $S$ its characteristic predicate $\chi_S$, which is defined such that for any $x$ in $\mathcal{D}$, $\chi_S(x)$ holds iff $x \in S$. Then we have the following equivalences for set atomic formulas:

$$\begin{array}{rcl} e \in S & \Leftrightarrow & \chi_S(e) \\ S_1 \subseteq S_2 & \Leftrightarrow & (\forall x \in \mathcal{D})\ \chi_{S_1}(x) \rightarrow \chi_{S_2}(x) \\ S_1 \leq S_2 & \Leftrightarrow & (\forall x_1,\ x_2 \in \mathcal{D})\ \chi_{S_1}(x_1) \wedge \chi_{S_2}(x_2) \rightarrow x_1 \leq x_2 \end{array}$$

and for set terms:

$$
\begin{array}{lll}
(\forall x \in \mathcal{D}) \quad \chi_{S_1 \cup S_2}(x) & \leftrightarrow & \chi_{S_1}(x) \vee \chi_{S_2}(x) \\
(\forall x \in \mathcal{D}) \quad \chi_{S_1 \cap S_2}(x) & \leftrightarrow & \chi_{S_1}(x) \wedge \chi_{S_2}(x) \\
(\forall x \in \mathcal{D}) \quad \chi_{S_1 \setminus S_2}(x) & \leftrightarrow & \chi_{S_1}(x) \wedge \neg \chi_{S_2}(x)
\end{array}
$$

For each quantifier-free ordered sets formula $\varphi$ we construct a quantifier free formula $\varphi_{\mathcal{D}}$ such that the only atomic formulas in $\varphi_{\mathcal{D}}$ are of the form $e_1 = e_2$, $e_1 \leq e_2$, or $\chi_S(e)$, where $e$, $e_1$, $e_2$ are constants in $\mathcal{D}$. We begin by converting $\varphi$ to its negation normal form, such that $\varphi$ is constructed only with conjunction and disjunction from positive or negative atomic formulas. Then we express all the set terms and atomic formulas in terms as described above, and we replace any negative occurrence of a universally quantified formula $(\forall X)\ \psi$ with the positive occurrence $(\exists X)\ \psi$. We call the resulting formula $\varphi_{\exists\forall\mathcal{D}}$. So far we have a formula, $\varphi_{\exists\forall\mathcal{D}}$, which equivalent to $\varphi$, does not use the set atomic formulas anymore, but has both existential and universal quantification.

Next, we proceed to eliminate all the existential quantification from $\varphi_{\exists\forall\mathcal{D}}$. We replace each occurrence of the formula $(\exists x)\ \psi(x)$ with $\psi(e)$, where $e$ is a fresh constant. Similarly, we replace each occurrence of $(\exists x_1,\ x_2)\ \psi(x_1,\ x_2)$ with $\psi(e_1,\ e_2)$ where $e_1$ and $e_2$ are fresh constants. Thus, we have witnesses for each existential formula. We call the resulting formula $\varphi_{\forall\mathcal{D}}$. It is easy to see that $\varphi_{\forall\mathcal{D}}$ is satisfiable iff $\varphi_{\exists\forall\mathcal{D}}$ is satisfiable. Each model of $\varphi_{\forall\mathcal{D}}$ is a model of $\varphi_{\exists\forall\mathcal{D}}$, as the existential quantifiers can be instantiated with the freshly created constants. On the other hand, from a model of $\varphi_{\exists\forall\mathcal{D}}$ we can create a model of $\varphi_{\forall\mathcal{D}}$ by interpreting the extra constants to be the elements use to instantiate the existential quantifiers.

Finally, we proceed to eliminate the universal quantification from $\varphi_{\forall\mathcal{D}}$. Let $\{e_1,\ e_2,\ \ldots, e_n\}$ be the set of constants appearing in the formula. We construct the formula $\varphi_{\mathcal{D}}$ by replacing each occurrence of a formula $(\forall x)\ \psi(x)$ with the formula $\bigwedge_i \psi(e_i)$, and each occurrence of $(\forall xy)\ \psi(x_1,\ x_2)$ with $\bigwedge_{i,\ j} \psi(e_i,\ e_j)$. It follows trivially that any model of $\varphi_{\forall\mathcal{D}}$ is also a model of $\varphi_{\mathcal{D}}$. Let $\mathcal{M}$ be a model of $\varphi_{\mathcal{D}}$. We construct the model $\mathcal{M}'$ from $\mathcal{M}$ by setting the $\chi_S(x)$ to `false` for all the elements of $\mathcal{D}$ that are not the image of a constant (that are not equal to any of the $e_i$), and preserving everything else. It follows that $\mathcal{M}'$ satisfies a formula $(\forall X)\ \psi$ iff $\mathcal{M}$ satisfies it, and

we can conclude that from any model for $\varphi_{\mathcal{D}}$ we can construct a model for $\varphi_{\forall\mathcal{D}}$. Hence $\varphi_{\mathcal{D}}$ is satisfiable iff $\varphi_{\forall\mathcal{D}}$ is satisfiable. Moreover, $\varphi_{\mathcal{D}}$ is satisfiable iff the original formula $\varphi$ is satisfiable. That is, ordered sets over $(\mathcal{D}, \leq)$ is decidable iff $(\mathcal{D}, \leq)$ itself is decidable.

Similarly, we can reduce the decidability of ordered multisets over $(\mathcal{D}, \leq)$ to the decidability of $(\mathcal{D}, \leq)$ combined with Presburger arithmetic. The main difference it that instead of a characteristic function $\chi_S$, we associate with each multiset $S$ a counting function $\nu_S$ such that for any $x$ in $\mathcal{D}$, $\nu_S(x)$ is equal with the multiplicity of $x$ in $S$. Then we have the following equivalences for multiset atomic formulas:

$$\begin{aligned} e \in S &\Leftrightarrow \nu_S(e) > 0 \\ S_1 \subseteq S_2 &\Leftrightarrow (\forall x \in \mathcal{D})\ \nu_{S_1}(x) \leq \nu_{S_2}(x) \\ S_1 \leq S_2 &\Leftrightarrow (\forall x_1,\ x_2 \in \mathcal{D})\ \nu_{S_1}(x_1) > 0 \wedge \nu_{S_2}(x_2) > 0 \rightarrow x_1 \leq x_2 \end{aligned}$$

and for multiset terms:

$$\begin{aligned} (\forall x \in \mathcal{D}) \quad \nu_{S_1 \cup S_2}(x) &= \nu_{S_1}(x) + \nu_{S_2}(x) \\ (\forall x \in \mathcal{D}) \quad \nu_{S_1 \cap S_2}(x) &= min(\nu_{S_1}(x),\ \nu_{S_2}(x)) \\ (\forall x \in \mathcal{D}) \quad \nu_{S_1 \setminus S_2}(x) &= max(\nu_{S_1}(x) - \nu_{S_2}(x),\ 0) \end{aligned}$$

For each ordered multisets formula $\varphi$ we construct the quantifier free formula $\varphi_{\mathcal{D}}$ like we do for sets. The main difference is that for multisets, $\varphi_{\mathcal{D}}$ also contains Presburger arithmetic.

The size of the $\varphi_{\forall\mathcal{D}}$ formula is the same as the size of $\varphi$. The size of the $\varphi_{\mathcal{D}}$ formula is at most $n^2$ times bigger than the size of $\varphi$, where $n$ is the number of constants in $\varphi_{\forall\mathcal{D}}$. The number $n$ of constants is the number of constants in $\varphi$ plus the number of constants added during the existential quantification elimination phase. As each atomic formula contributes with at most two fresh constants, $n$ is at most linear in the size of $\varphi$, and the size of $\varphi_{\mathcal{D}}$ is at most cubic in the size of $\varphi$. However, in our experiments, as described in Section 5.5, the number of constants is small, in the range between 5 and 15, and are handled efficiently by SMT solvers.

**Proposition 5.4.2.** *The validity of the abstracted formula $\varphi_{abs} \rightarrow \psi_{abs}$ is decidable.*

*Proof.* The formula is quantifier-free and falls in the theory of uninterpreted function, linear arithmetic, and the ordered-set theory described above. The

decidability has been shown by giving the decision procedure described above. □

## 5.5 Experiments

We demonstrate the effectiveness and practicality of the $\text{DRYAD}_{\text{tree}}$ logic and the natural proof strategy developed in this chapter by verifying standard operations on several inductive data structures. Each routine was written using recursive functions and annotated with a pre-condition and a post-condition, specifying a set of partial correctness properties including both structural and data requirements. For each basic block of each routine, we manually generated the verification condition $(SH; \varphi)$ following the procedure described in Section 5.2. Then we examined the validity of $\varphi$ using the procedure described in Section 5.4. We employ Z3 [28], a state-of-the-art SMT solver, to check validity of the generated formula $\varphi_{\mathcal{D}}$ formula in the quantifier-free theory of integers and uninterpreted functions `QF_UFLIA`.

### 5.5.1 The Data-Structures, Routines, and Verified Properties

The set of benchmarks is an almost exhaustive list of algorithms on tree-based data-structures covered in a first undergraduate course on data-structures [27].

Lists are trees with a singleton direction set. *Sorted lists* can be recursively defined as either an empty list, or a head node followed by a sorted list with all its keys not less than the key of the head, and hence expressed in DRYAD. The routines `insert` and `delete` insert and delete a node with key $k$ in a sorted list, respectively, in a recursive fashion. The routine `insertion-sort` takes a singly-linked list and sorts it by recursively sorting the tail of the list, and inserting the key of the head into the sorted list by calling `insert`. We check if all these routines return a sorted list with the multiset of keys as expected.

A *binary heap* is recursively defined as either an empty tree, or a binary tree such that the root is of the greatest key and both its left and right subtrees are binary heaps. The routine `max-heapify` is given a binary tree with both its left and right trees are binary heaps. If the binary-heap property is violated by the root, it swaps the root with its greater child, and then recursively

max-heapifies that subtree. We check if the routine returns a binary heap with same keys as that of the input tree.

The *treap* data-structure is a class of binary trees with two data fields for each node: *key* and *priority*. We assume that all priorities are distinct and all keys are distinct. Treaps can also be recursively defined in DRYAD. It is defined as either an empty tree, or a binary tree with both its left and right subtrees as treaps, and the key of the root obeys the binary-search-tree property, and the priority of the root obeys the min-heap order property. The `remove-root` routine deletes the root of the input treap, and joins the two subtrees descending from the left and right children of the deleted node into a single treap. If the left or right subtree of the node to be deleted is empty, the join operation is trivial; otherwise, the left or right child of the deleted node is selected as the new root, and the deletion proceeds recursively. The `delete` routine simply searches the node to be deleted recursively, and deletes it by calling `remove-root`. The `insert` routine recursively inserts the new node into an appropriate subtree to keep the binary-search-tree property, then performs rotations to restore the min-heap order property, if necessary. We check if all these routines return a treap with the set of keys and the set of priorities as expected.

An *AVL tree* is a binary search tree that is balanced: for each node, the absolute difference between the height of the left subtree and the height of the right subtree is at most 1. The recursive predicate $avl^*(t)$ is defined as either the empty tree, or a binary tree such that: (1) the key stored in the root is no less than any key in stored in the left subtree, and no greater than any key stored in the right subtree; (2) the difference between the heights of the left and right subtree is between -1 and 1; and (3) both the left and the right subtree satisfy the $avl^*$ predicate. The main routines for AVL are `insert` and `delete`. The `insert` routine recursively inserts a key into an AVL tree (similar to a binary search tree), and as it returns from recursion it checks the balancedness and performs one or two rotations to restore balance. The `delete` routine recursively deletes a key from an AVL tree (again, similar to a binary search tree), and as it returns from the recursion ensures that the tree is indeed balanced. For both routines, we prove that they return an AVL tree, that the multiset of keys is as expected, and that the height increases by at most 1 (for `insert`), can decrease by at most 1 (for `delete`), or stays the same.

*Red-black trees* are BSTs that are more loosely balanced than the AVL trees, and were described in Section 5.1.2. The height of the left subtree is between half and twice the height of the right subtree. We consider the `insert` and `delete` routines. The `insert` routine recursively inserts a key into a red-black subtree, and colors the new node red, possibly violating the red-black tree property. As it returns from recursion, it performs several rotations to fix the property. If the root of the whole tree is red, it is recolored black, and all the properties are restored. The `delete` routine recursively deletes a key from a red-black tree, possibly violating the red-black tree property. As it returns from recursion, it again performs several rotations to fix it. In the end, it is possible to decrease the black height of the whole tree, and all the properties are restored. For both routines, we prove that they return a red-black tree, that the multiset of keys is as expected, and the black height increases by at most 1 (for `insert`), decreases by at most 1 (for `delete`), or does not change.

The *B-tree* is a data structure that generalizes the binary search tree in that for each non-leaf node the number of children is one more than the number of keys stored in that node. The keys are stored in increasing order, and if the node is not a leaf, the key with index $i$ is no less than any key stored in the child with index $i$ and no more then all the keys stored in the child with index $i+1$. For each node except the root, the number of keys is in some range (typically between $T-1$ and $2T-1$). A B-tree is balanced in that for each node, the heights of all the children are equal. To describe the B-tree in our logic, we need three mutually recursive predicates: one that describes a B-subtree, and two that describe a list of keys and children. The $\textit{b-subtree}^*(t)$ states that the number of keys stored in the node is in the required range, and that keys and children list satisfies either the $\textit{key-child-list}^*(l)$ predicate (if the node is not a leaf) or the $\textit{key-list}^*(l)$ predicate (if the node is a leaf). The $\textit{key-child-list}^*(l)$ states that either the list has only one element, and the child satisfies the $\textit{b-subtree}^*(t)$ predicate, or the list has at least two elements, the key in the head of the list is no less than all the keys stored in the child and no greater than the keys stored in the tail of the list, the child satisfies $\textit{b-subtree}^*(t)$, and the height of the child is equal to the height of the tail (the height of a list is defined as the maximum height of a child). The predicate $\textit{key-list}^*(l)$ is similarly defined. We consider the `find` and `insert` routines. The `find` routine iterates over the list of keys, and recurses into

the appropriate child, until it finds the key or it arrives to a leaf. The `insert` procedure is more complex, as it assumes that the node it is inserting into is non-full, and prior to recursion it might need to split the child. For both routines, we check that the tree after the call is a B-tree, that the multiset of keys has the expected value, and that the height of the tree stays the same, or increases by at most 1 (for `insert`).

As an advanced data structure, the *binomial heap* is described by a set of predicates defined mutually recursively: *binomial-tree*$^*$, *binomial-heap*$^*$ and *full-list*$^*$. We represent a binomial heap as follows. Briefly, a binomial-heap of order $k$ consists of a binary-tree of order $k$ and a binary-heap of order less than $k$. A binomial-tree of order $k$ is an ordered tree defined recursively: the root contains the minimum key, and its children compose a binomial-heap of order $k-1$, satisfying the full-list property. A full-list of order $k$ consists of a tree of order $k$ and a full-list of order $k-1$. The left-child, right-sibling scheme represents each binomial tree within a binomial heap. Each node contains its key; pointers to its leftmost child and to the sibling immediately to its right; and its degree. The roots of a binomial heap form a singly-linked list (also connected by the sibling pointer). We access the binomial heap by a pointer to the first node on the root list.

The `find-minimum` routine expects a nonempty binomial heap, and moves the tree containing the minimum key to the head of the list. It returns the original heap if it is a single tree. Otherwise, it calls `find-minimum` on its tail list, and appends the returned list to the head tree; then if keys of the roots of the first two trees are unordered, swaps the two trees. We check that `find-minimum` returns a binomial tree followed by a binomial heap, such that the root of the tree contains the minimum key, and the head of the binomial heap is either the first or the second root of the original heap. The `merge` routine merges two binomial heaps `x` and `y` into one. If one of the two heaps is empty, it simply returns the other one. Otherwise, if the heads of the two heaps are of the same order, it merges the two head trees into one, merges the two tail lists recursively, and returns the new tree followed by the new heap; if not, say, `x.order > y.order`, then it merges `x.sibling` and `y`, concatenates the head tree of `x` and the new heap in an appropriate way satisfying the binomial-heap property. We check that `merge` returns a binomial heap such that the keys are the union of the two input binomial heaps, and the order increases up to 1. The `delete-minimum` routine is non-recursive. It simply

moves the minimum tree `m` to the head by calling `find-minimum`, and obtains two binomial heaps: a list of the siblings of `m`, and a list of the children of `m`. Finally it merges the two heaps by `merge`. We check that `delete-minimum` returns a binomial heap with the multiset of keys as expected.

### 5.5.2 Experimental Results

Table 5.1 summarizes our experiments, showing the results of verifying 147 basic blocks across these algorithms. All the verified programs can be found at `http://www.cs.illinois.edu/~qiu2/dryad` . The experiments were conducted on a dual-core, 3.2GHz, 8GB machine, running Windows 7 and Z3 2.19. For each data structure, we report the number of integers, sets, multisets and predicates defined recursively. For each routine, we report the number of basic blocks, the number of nodes in the footprint, the time taken by Z3 to determine validity of all generated formulas, and the validity result proved by Z3.

We are encouraged by the fact that all these verification conditions that were deterministically generated by the methodology set forth in this chapter were proved by Z3 efficiently; this proved all these algorithms correct. To the best of our knowledge, this is an efficient terminating automatic mechanism that can prove such a wide variety of data-structure algorithms written using imperative programming correct (in particular, binomial heaps and the B-trees presented here have not been proven automatically correct).

The experimental results show that $\text{D}\textsc{ryad}_{\text{tree}}$ is a very expressive logic that allows us to express natural and recursive properties of several complex inductive tree data structures. Moreover, our sound procedure tends to be able to prove many complex verification conditions.

## 5.6 Related Work

There is a rich literature on program logics for heaps. We discuss the work that is closest to ours. In particular, we omit the rich literature on general interactive theorem provers (like Coq [40]) as well as general software verification tools (like Boogie [5]) that are not particularly adapted for heap verification.

| Data Structure | #Ints | #Sets | #MSets | #Preds | Routine | #BB | Max. #Nodes in Footprint | Total Time (s) | Avg. Time (s) per VC | VC proved valid? |
|---|---|---|---|---|---|---|---|---|---|---|
| Sorted List | 0 | 0 | 1 | 1 | `insert` | 4 | 3 | 0.24 | 0.06 | Yes |
| | | | | | `delete` | 3 | 3 | 0.17 | 0.06 | Yes |
| | | | | | `insertion-sort` | 3 | 4 | 0.11 | 0.04 | Yes |
| Binary Heap | 0 | 0 | 1 | 1 | `max-heapify` | 5 | 8 | 1.89 | 0.38 | Yes |
| Treap | 0 | 2 | 0 | 1 | `insert` | 7 | 6 | 4.06 | 0.58 | Yes |
| | | | | | `delete` | 6 | 4 | 0.81 | 0.14 | Yes |
| | | | | | `remove-root` | 7 | 8 | 2.96 | 0.42 | Yes |
| AVL Tree | 1 | 0 | 1 | 1 | `insert` | 11 | 8 | 1.45 | 0.13 | Yes |
| | | | | | `delete` | 18 | 7 | 2.13 | 0.19 | Yes |
| Red-Black Tree | 1 | 0 | 1 | 1 | `insert` | 19 | 8 | 1.93 | 0.11 | Yes |
| | | | | | `delete` | 24 | 7 | 3.22 | 0.14 | Yes |
| B-Tree | 2 | 0 | 1 | 2 | `insert` | 12 | 6 | 0.40 | 0.03 | Yes |
| | | | | | `find` | 8 | 3 | 0.12 | 0.02 | Yes |
| Binomial Heap | 1 | 0 | 1 | 3 | `delete-minimum` | 3 | 7 | 0.29 | 0.10 | Yes |
| | | | | | `find-minimum` | 4 | 6 | 1.81 | 0.45 | Yes |
| | | | | | `merge` | 13 | 7 | 17.38 | 1.37 | Yes |
| Total | | | | | | 147 | | | | |

Table 5.1: Results of program verification using DRYAD$_{\text{tree}}$
more details at `http://web.engr.illinois.edu/~qiu2/dryad` .

Separation logic [63, 68, 10] is one of the most popular logics for verification of heap structures. Many dialects of separation logic combine separation logic with inductively defined data-structures. While separation logic gives mechanisms to compositionally reason with the footprint and the frame it resides in, proof assistants for separation logic are often heuristic and incomplete [10], though a couple of small decidable fragments are known [53, 9]. A work that comes very close to ours is a paper by Chin et al. [23], where the authors allow user-defined recursive predicates (similar to ours) and build a terminating procedure that reduces the verification condition to standard logical theories. While their procedure is more general than ours (they can handle structures beyond trees), the resulting formulas are quantified, and result in less efficient procedures. Bedrock [24] is a Coq library that aims at mostly automated (but not completely automated) procedures that requires some proof tactics to be given by the user to prove verification conditions.

In manual and semi-automatic approaches to verification of heap manipulating programs [68, 69, 10], the inductive definitions of algebraic data-types is extremely common, and proof tactics unroll these inductive definitions, do extensive unification to try to match terms, and find simple proofs. Our work in this chapter is very much inspired by the kinds of manual heap reasoning that we have seen in the literature.

The work by Zee et al. [81, 82] is one of the first attempts at full functional verification of linked data structures, which includes the development of the JAHOB system that uses higher-order logics to specify correctness properties, and puts together several theorem provers ranging from first-order provers, SMT solvers, and interactive theorem provers to prove properties of algorithms manipulating data-structures. While many proofs required manual guidance, this work showed that proofs can often be derived using simple tactics like unrolling of inductive definitions, unification, abstraction, and employing decision procedures for decidable theories. This work was also an inspiration for our work, but we chose to concentrate on deriving proofs using *completely automatic and terminating procedures*, where unification, unrolling, abstraction, and decidable theories are systematically exploited.

One work that is very close to ours is that of Suter et al. [72] where decision procedures for algebraic data-types are presented with abstraction as the key to obtaining proofs. However, this work focuses on sound and complete decision procedures, and is limited in its ability to prove several complex data

structures correct. Moreover, the work limits itself to functional program correctness; in our opinion, functional programs are very similar to algebraic inductive specifications, leading to much simpler proof procedures.

There is a rich and growing literature on completely automatic sound, complete, and terminating decision procedures for restricted heap logics. The logic LISBQ [46] offers such reasoning with restricted reachability predicates and quantification. While the logic has extremely efficient decision procedures, its expressivity in stating properties of inductive data-structures (even trees) is very limited. There are several other logics in this genre, being less expressive but decidable [15, 12, 65, 66, 59, 67, 2]. STRAND, as shown earlier in this dissertation, is a recent logic that can handle some data-structure properties (at least binary search trees) and admits decidable fragments by combining decidable theories of trees with the theory of arithmetic, but is again extremely restricted in expressiveness. None of these logics can express the verification conditions for full functional verification of the data-structures explored in this chapter.

# CHAPTER 6

# COMBINING SEPARATION AND RECURSION

As mentioned in Chapter 1, it is well known that heap analysis and verification is notoriously difficult. In recent years, *Separation Logic* (SL), especially in combination with recursive definitions, has emerged as a succinct and natural logic to express properties about structure and separation [68, 63]. However, the validation of verification conditions resulting from separation logic invariants are also complex, and has eluded automatic reasoning and exploitation of SMT solvers (even more so than tools such as Boogie that use classical logic). Again, help from the user in proving the verification conditions are currently necessary— the tools VERIFAST [41] and BEDROCK [24], for instance, admit separation logic specifications but require the user to write lower-level lemmas and proof tactics to guide the verification. For example, in verifying an in-place reversal of a linked list[1], BEDROCK would require several lemmas and a hint package be supplied at the level of the code in order for the proof to go through.

On the other hand, as proposed in Chapter 5, the natural proof strategy is a novel attempt to combine expressive logics and automated reasoning. Exploiting natural proofs results in successful verification of a wide variety of tree-manipulating programs. The natural proofs developed in Chapter 5 were too restrictive, however, handling only *single* trees, with no scope for handling multiple or more complex data-structures and their separation.

We believe the two nice techniques complement each other: the succinctness of SL in reasoning with framing and separation, and the efficient decidability of natural proofs. Hence, the aim of this chapter is to provide natural proofs for general properties of structure, data, and separation. Our contributions are:

[1] `http://plv.csail.mit.edu/bedrock/Tutorial.html`

a) we propose $\textsc{Dryad}_{\text{sep}}$, a dialect of separation logic for heaps, with no explicit (classical) quantification but with recursive definitions, to express second-order properties;

b) show that $\textsc{Dryad}_{\text{sep}}$ is both powerful in terms of expressiveness, and that the strongest postcondition of $\textsc{Dryad}_{\text{sep}}$ specifications with respect to bounded code segments can be formulated in $\textsc{Dryad}_{\text{sep}}$;

c) show how $\textsc{Dryad}_{\text{sep}}$ has been designed so that it can be systematically converted to *classical* logic using the theory of sets, allowing us to connect the more natural and succinct specifications to more verbose but classical logic.

**Organization:** In the rest of this chapter, we introduce the design principles of the new logic in Section 6.1. After that we present our logic $\textsc{Dryad}_{\text{sep}}$, a quantifier-free heaplet logic augmented with recursively defined predicates/functions, ini terms of its syntax and its disciplined semantics in Section 6.2 and , respectively. We also clarify the difference between SL and $\textsc{Dryad}_{\text{sep}}$ via examples in Section 6.4. Section 6.5 is dedicated to a formal translation from $\textsc{Dryad}_{\text{sep}}$ to a classical logic extended with set theory.

## 6.1 Logic Design

In this section, we elaborate the two key ingredient of our logic $\textsc{Dryad}_{\text{sep}}$, and explain why they are amenable to our natural proof strategy set forth in Chapter 5.

### 6.1.1 Deterministic Scope

The primary design principle behind separation logic is the decision to express *strict specifications*— logical formulas must naturally refer to *heaplets* (subparts of the heap), and, by default, the smallest heaplets over which the formula needs to refer to. This is in contrast to classical logics (such as FOL) which implicitly refer to the entire heap globally. Strict specifications permit elegant ways to capture how a particular sub-part of the heap changes due to a procedure, implicitly leaving the rest of the heap and its

properties unchanged across a call to this procedure. Separation logic is a particular framework for strict specifications, where formulas are implicitly defined on strictly defined heaplets, and where heaplets can be combined using a *spatial conjunction operator* denoted by $*$. The frame rule in separation logic captures the main advantage of strict specifications: if the Hoare-triple $\{P\}C\{Q\}$ holds for some program $C$, then $\{P * R\}C\{Q * R\}$ also holds (with side-conditions that the modified variables in $C$ are disjoint from the free variables in $R$).

While the above motivation for separation logic based on strict specifications is worthy in itself, separation logic syntax gives another distinct advantage to our goals of building natural proofs for generalized structural properties. Going from handling tree structures in Chapter 5 to more general structures expressed in logic, a primary concern is how the structural property of the heap is expressed. Consider, for example, expressing that the location $x$ is the root of a tree. This is a *second-order* property and formulations of it in classical logic using set or path quantification are quite complex and not easily amenable to automated verification. We prefer *inductive* definitions of structural properties without any explicit quantification. The separation logic syntax with recursive definitions and heaplet semantics allows simple quantifier-free formulas to express structural restrictions; for example. tree-ness can be expressed simply as:

$$tree(x) :: (x = \texttt{nil} \land \texttt{emp}) \lor (x \longmapsto (l, r) * tree(l) * tree(r))$$

We first define a new logic, $\text{DRYAD}_{\text{sep}}$, that permits no explicit quantification, but permits powerful recursive definitions to define integers, sets/multisets of integers, and sets of locations, using least fixed-points. The logic $\text{DRYAD}_{\text{sep}}$ furthermore has a heaplet semantics and allows the spatial conjunction operator $*$. However, a key design feature of $\text{DRYAD}_{\text{sep}}$ is that the heaplet for recursive formulas is essentially *determined* by the syntax as opposed to the semantics. In classical SL, a formula of the form $\alpha * \beta$ says that the heaplet can be partitioned into *any* two disjoint heaplets, one satisfying $\alpha$ and the other $\beta$. In $\text{DRYAD}_{\text{sep}}$, the heaplet for a (complex) formula is *determined* and hence if there is a way to partition the heaplet, there is precisely *one* way to do so. We have found that most uses of separation logic to express properties can be written quite succinctly and easily using $\text{DRYAD}_{\text{sep}}$

(in fact, it is *easier* to write such deterministic-heap specifications). The key advantage is that this eliminates implicit existential quantification the separation operator provides. In a verification condition that combines the pre-condition in the negative and the postcondition in the positive, the classical semantics for SL invariably introduces universal quantification in the satisfiability query for the negation of the verification condition, which in turn is extremely hard to handle.

In DRYAD$_{\text{sep}}$, the semantics of a recursive definition $r(x)$ (such as *tree* above), requires that the heaplet be determined and defined as the set of all locations reachable from the node $x$ through a set of pointer-fields $f_1, \ldots, f_k$ without passing through a set of locations (given by a set of location terms $t_1, \ldots t_n$). While our logical mechanisms can be extended beyond this notion (in deterministic ways), we have found that this captures most common properties required in proving data-structure manipulating programs correct.

The above formulation lends well to the natural proof methodology — it doesn't use quantification (the implicit quantification on $l$ and $r$ are special since they are uniquely determined by $x$) and it is amenable to unfolding across a footprint, and hence amenable to natural proofs, provided we can only handle the separation logic semantics in a decidable theory.

### 6.1.2 Deterministic Translation

The second key step in our paradigm is a technique to bridge the gap from separation logic to classical logic in order to utilize efficient decision procedures supported by SMT solvers. We show that heaplet semantics and separation logic constructs of DRYAD$_{\text{sep}}$ can be effectively translated to classical logic where heaplets are modeled as *sets of locations*. We show that DRYAD$_{\text{sep}}$ formulas can be translated into classical logic with free set variables that capture the heaplets corresponding to the strict semantics. This translation does not, of course, yield a decidable theory yet, as recursive definitions are still present (the recursion-free formulas are in a decidable theory). The carefully designed DRYAD$_{\text{sep}}$ logic with determined heaplet semantics ensures that there is no quantification in the resulting formula in classical logic. The heaplets of recursively defined properties, which are de-

fined using the set of all *reachable* nodes, are translated to recursively defined sets of locations.

## 6.2 Syntax

Let us fix a finite set of *pointer-fields* $PF$ and a finite set of *data-fields* $DF$. A record consists of a set of pointer-fields from $PF$ and a set of data-fields from $DF$. Our logic also presumes that locations refer to entire records rather than particular fields, and that address arithmetic is precluded. We will use the term *locations* hence to refer to these records. We assume that every field is defined at every location, i.e., all memory records have the same layout (to simplify the presentation); our logic can easily be extended with record types.

Let $Bool = \{\texttt{true}, \texttt{false}\}$ stand for the set of Boolean values, $Int$ stand for the set of integers and $Loc$ stand for the universe of locations. For any set $A$, let $\mathcal{S}(A)$ denote the set of all finite subsets of $A$, and let $\mathcal{MS}(A)$ denote the set of all finite multisets with elements in $A$.

The DRYAD logic allows expressing quantifier-free first-order properties over heaps/heaplets augmented with recursively defined notions for a location to express second-order properties, denoted as a function $r : Loc \rightarrow D$. The codomain $D$ can be $Int_L$, $\mathcal{S}(Loc)$, $\mathcal{S}(Int)$, $\mathcal{MS}(Int)_L$ or $Bool$, where $Int_L$ and $\mathcal{MS}(Int)_L$ extend $Int$ and $\mathcal{MS}(Int)$ to lattice domains, respectively, in order to give least fixed-point semantics (explained later in this section). Typical examples of these recursive definitions include the definitions of the height of a tree or the height of black-nodes in the tree rooted at a node (recursively defined integers), the set of nodes reachable from a location following certain pointer fields (recursively defined sets of locations), the set/multiset of keys stored at a particular data-field under nodes reachable from a location (recursively defined set/multiset of integers), and the property that the tree rooted at a node is a binary search tree or a balanced tree or just a tree (recursively defined predicates).

A DRYAD$_{\text{sep}}$ formula $\varphi$ is quantifier-free, but parameterized by a set of recursive definitions $Def^{\Delta}$. The syntax of DRYAD logic is given in Figure 6.1, where the syntax of formulas is followed by the syntax for recursive definitions. Most symbols in DRYAD$_{\text{sep}}$ are common and self-explanatory. Note

$$
\begin{array}{lll}
i^{\Delta} : Loc \rightarrow Int_L & j \in Int_L \text{ Vars} & x \in Loc \text{ Vars} \\
sl^{\Delta} : Loc \rightarrow \mathcal{S}(Loc) & L \in \mathcal{S}(Loc) \text{ Vars} & c : Int_L \text{ Constant} \\
si^{\Delta} : Loc \rightarrow \mathcal{S}(Int) & S \in \mathcal{S}(Int) \text{ Vars} & pf \in PF \\
msi^{\Delta} : Loc \rightarrow \mathcal{MS}(Int)_L & MS \in \mathcal{MS}(Int)_L \text{ Vars} & df \in DF \\
p^{\Delta} : Loc \rightarrow Bool & q \in Bool \text{ Vars} &
\end{array}
$$

$$
\begin{array}{rrcl}
Loc \text{ Term:} & lt & ::= & x \mid \texttt{nil} \\
Int_L \text{ Term:} & it & ::= & c \mid j \mid i^{\Delta}_{\vec{pf},\vec{v}}(lt) \mid it + it \mid it - it \\
\mathcal{S}(Loc) \text{ Term:} & slt & ::= & \emptyset_l \mid L \mid \{lt\} \mid sl^{\Delta}_{\vec{pf},\vec{v}}(lt) \mid \\
 & & & slt \cup slt \mid slt \cap slt \mid slt \setminus slt \\
\mathcal{S}(Int) \text{ Term:} & sit & ::= & \emptyset_s \mid S \mid \{it\} \mid si^{\Delta}_{\vec{pf},\vec{v}}(lt) \mid \\
 & & & sit \cup sit \mid sit \cap sit \mid sit \setminus sit \\
\mathcal{MS}(Int)_L \text{ Term:} & msit & ::= & \emptyset_m \mid MS \mid \{it\}_m \mid msi^{\Delta}_{\vec{pf},\vec{v}}(lt) \mid \\
 & & & msit \cup msit \mid msit \cap msit \mid msit \setminus msit
\end{array}
$$

$$
\begin{array}{rrcl}
\text{Positive Formula:} & \varphi & ::= & \texttt{true} \mid \texttt{false} \mid q \mid \\
 & & & p^{\Delta}_{\vec{pf},\vec{v}}(lt) \mid \texttt{emp} \mid lt \xmapsto{\vec{pf},\vec{df}} (\vec{lt}, \vec{it}) \mid \\
 & & & lt = lt \mid lt \neq lt \mid it \leq it \mid it < it \mid \\
 & & & sit \leq sit \mid sit < sit \mid \\
 & & & msit \leq msit \mid msit < msit \mid \\
 & & & slt \subseteq slt \mid slt \not\subseteq slt \mid lt \in slt \mid lt \notin slt \mid \\
 & & & sit \subseteq sit \mid sit \not\subseteq sit \mid \\
 & & & msit \sqsubseteq msit \mid msit \not\sqsubseteq msit \mid \\
 & & & it \in sit \mid it \notin sit \mid it \in msit \mid it \notin msit \mid \\
 & & & \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \varphi * \varphi \\
\text{Formula:} & \psi & ::= & \varphi \mid \psi \wedge \psi \mid \psi \vee \psi \mid \neg \psi
\end{array}
$$

$$
\begin{array}{rcl}
\text{Recursive function :} \quad f^{\Delta}_{\vec{pf},\vec{v}}(x, \vec{v}) & \stackrel{def}{=} & \big(\ \varphi^f_1(x, \vec{v}, \vec{s}) : t^f_1(x, \vec{s}) \ ; \\
 & & \quad \ldots \ ; \\
 & & \quad \varphi^f_k(x, \vec{v}, \vec{s}) : t^f_k(x, \vec{s}) \ ; \\
 & & \quad \text{default} : t^f_{k+1}(x, \vec{s}) \ \big) \\
\text{Recursive predicate :} \quad p^{\Delta}_{\vec{pf},\vec{v}}(x, \vec{v}) & \stackrel{def}{=} & \varphi^p(x, \vec{v}, \vec{s})
\end{array}
$$

Figure 6.1: Syntax of DRYAD$_{\text{sep}}$

that the inequality ($<$ or $\leq$) between integer sets/multisets indicates that any integer in the left-hand side is less-than/not-greater-than any integer in the right-hand side. It is also noteworthy that the separating conjunction ($*$) from separation logic is also allowed, but only if it is not above any negation ($\neg$). We require that every recursive function/predicate used in

the formula $\varphi$ has a unique definition in $Def^{\Delta}$. Each recursive function is parameterized by a set of pointer fields $\overrightarrow{pf}$ and a set of program variables $\vec{v}$, denoted as $f^{\Delta}_{\overrightarrow{pf},\vec{v}}$. The subscripts are used in defining the semantics of recursive functions in Section 6.3. We usually simply use $f^{\Delta}$ when the subscripts are not relevant in the context. Similarly, recursive predicates are denoted as $p^{\Delta}_{\overrightarrow{pf},\vec{v}}$ or simply $p^{\Delta}$. The recursive functions are defined using the syntax:

$$\big(\varphi^f_1(x,\vec{v},\vec{s}) : t^f_1(x,\vec{s})\ ;\ \ldots\ ;\ \varphi^f_k(x,\vec{v},\vec{s}) : t^f_k(x,\vec{s})\ ;\ \text{default} : t^f_{k+1}(x,\vec{s})\big)$$

where $\varphi^f_u(x,\vec{v},\vec{s})/t^f_u(x,\vec{s})$ is a formula/term in our logic with $\vec{s}$ implicitly existentially quantified. The recursively defined predicates are defined using the syntax: $\varphi^p(x,\vec{v},\vec{s})$, which is a formula in our logic with $\vec{s}$ implicitly existentially quantified. The recursive function syntax above expresses a case-split, with the function evaluating to the first term whose guard evaluates to true. The restrictions on the recursive definitions are:

- Subtraction, set-difference, and negation are disallowed;
- Every variable in $\vec{s}$ should appear in the right hand side of a points-to relation binding it to $x$ *exactly once*.

For examples of recursive functions and predicates, see the definitions $keys^{\Delta}_{\overrightarrow{pf}}(x)$ and $mheap^{\Delta}_{\overrightarrow{pf}}(x)$ in Figure 7.2, respectively. The set of program variables $\vec{v}$ parameterizing the definitions is empty in both these definitions and the set of implicitly existentially-quantified variables $\vec{s}$ is $\{k, l, r\}$.

## 6.3 Semantics

Our logic is interpreted on models that are *program states*:

**Definition 6.3.1.** *A program state is a tuple $C = (R, s, h)$ where*

- $R \subseteq Loc \setminus \{nil\}$ *is a finite set of locations;*
- $s : Vars \rightarrow Int \cup Loc$ *is a store mapping program variables to locations or integers (of appropriate type);*
- $h : R \times (PF \cup DF) \rightarrow Int \cup Loc$ *is a heaplet mapping non-nil locations and each pointer-field/data-field to values of the appropriate type.* □

Note that the set of locations is, in general, larger than the state $R$ and hence $R$ defines a subset of heap locations. The store maps variables to locations (not necessarily in $R$), but the heaplet $h$ gives interpretations for pointer and data-fields only for elements in $R$.

Given a heaplet $h$, for every pointer field $pf$, we denote the projection of $h$ on $R \times (PF \setminus \{pf\} \cup DF)$ as $h \nmid pf$; similarly, for every data-field $df$, we denote the projection of $h$ on $R \times (PF \cup DF \setminus \{df\})$ as $h \nmid df$. Also, for every subset $S \subseteq R$, we denote the projection of $h$ on $S \times (PF \cup DF)$ as $h \mid S$.

A term/formula with free variables $F$ is interpreted by interpreting the free variables in $F$ using the map $s$ from variables to values. The semantics of $\text{DRYAD}_{\text{sep}}$ is similar to that of classical Separation Logic (*SL*). In particular, a term/formula without recursive definitions is interpreted exactly in the same way in $\text{DRYAD}_{\text{sep}}$ and *SL*. Hence we first give the semantics of the non-recursive part, followed by the semantics of recursive definitions.

Before defining the semantics of formulas, we define the *pure* property for terms/formulas. Intuitively, a term/formula is pure if it is independent of the heap. Syntactically, a term/formula is pure if it does not contain `emp`, $\longmapsto$ or any recursive definition. Note that in *SL* all terms are pure, but in $\text{DRYAD}_{\text{sep}}$, a term can be impure if it contains a recursive function $f^{\Delta}$.

Given a term $t$ we define pure($t$) inductively in Figure 6.2. The $f^*$ function refers to any recursive integer/set/multi-set definition; the op and $\sim$ operators stand for appropriate operators. Note that formulas with recursive definitions are not heapless, nor is any formula with separating conjunction $*$. We are now ready to define the semantics under a model $C = (R, s, h)$.

### 6.3.1 Terms

Each $\mathcal{T}$-term evaluates to either a normal value of type $\mathcal{T}$, or to `undef`, which is only used in interpreting recursive functions (will be explained later). As a special value, `undef` will be propagated throughout the formula: if a formula $\varphi$ contains a sub-term that evaluates to `undef`, then $\varphi$ will evaluate to `false` if it appears positively, and will evaluate to `true` otherwise. Intuitively, `undef` cannot help in making the formula true over a model.

The *Loc* terms are evaluated as follows:

$$
\begin{array}{lcl}
\text{pure}(var) & = & \texttt{true} \\
\text{pure}(c) & = & \texttt{true} \\
\text{pure}(f^*(lt)) & = & \texttt{false} \\
\text{pure}(\{t\}) & = & \text{pure}(t) \\
\text{pure}(t \ \text{ op } \ t') & = & \text{pure}(t) \wedge \text{pure}(t')
\end{array}
$$

$$
\begin{array}{lcl}
\text{pure}(\texttt{true}) & = & \texttt{true} \\
\text{pure}(p^*(lt)) & = & \texttt{false} \\
\text{pure}(\texttt{emp}) & = & \texttt{false} \\
\text{pure}(lt \overset{\vec{pf}, \vec{df}}{\longmapsto} (\vec{lt}, \vec{it})) & = & \texttt{false} \\
\text{pure}(t \sim t') & = & \text{pure}(t) \wedge \text{pure}(t') \\
\text{pure}(\varphi \wedge \varphi') & = & \text{pure}(\varphi) \wedge \text{pure}(\varphi') \\
\text{pure}(\varphi \vee \varphi') & = & \text{pure}(\varphi) \wedge \text{pure}(\varphi') \\
\text{pure}(\neg \varphi) & = & \text{pure}(\varphi) \\
\text{pure}(\varphi * \varphi') & = & \texttt{false}
\end{array}
$$

Figure 6.2: The *pure* predicate for terms/formulas

$$
\begin{array}{rcl}
[\![x]\!]_C & = & s(x) \\
[\![\texttt{nil}]\!]_C & = & nil
\end{array}
$$

For any binary operator op, $t$ op $t'$ is evaluated as follows:

$$
[\![t \text{ op } t']\!]_C = \begin{cases}
[\![t]\!]_C \text{ op } [\![t']\!]_C & \text{if } \ t \ \text{ or } \ t' \ \text{ is pure} \\
[\![t]\!]_{C|R_1} \text{ op } [\![t']\!]_{C|R_2} & \text{else if } \ \text{there exist } R_1, R_2 \text{ such that} \\
 & \qquad R = R_1 \cup R_2, [\![t]\!]_{C|R_1} \neq \texttt{undef} \\
 & \quad \text{and } \ [\![t']\!]_{C|R_2} \neq \texttt{undef} \\
\texttt{undef} & \text{otherwise}
\end{cases}
$$

where op is interpreted in the natural way.
For singletons, $\{it\}$ will evaluate to $\emptyset$ if $it$ evaluates to $-\infty$ or $\infty$:

$$
[\![\{it\}]\!]_C = \begin{cases}
\texttt{undef} & \text{if } \ [\![it]\!]_C = \texttt{undef} \\
\emptyset & \text{if } \ [\![it]\!]_C = -\infty \ \text{ or } \ \infty \\
\{[\![it]\!]_C\} & \text{otherwise}
\end{cases}
$$

Similarly, $\{it\}_m$ will evaluate to $\emptyset_m$ if $it$ evaluates to $-\infty$ or $\infty$:

$$[\![\{it\}_m]\!]_C = \begin{cases} \texttt{undef} & \text{if } [\![it]\!]_C = \texttt{undef} \\ \emptyset_m & \text{if } [\![it]\!]_C = -\infty \text{ or } \infty \\ \{[\![it]\!]_C\} & \text{otherwise} \end{cases}$$

and $\{lt\}$ will evaluate as follows:

$$[\![\{lt\}]\!]_C = \begin{cases} \texttt{undef} & \text{if } [\![lt]\!]_C = \texttt{undef} \\ \{[\![lt]\!]_C\} & \text{otherwise} \end{cases}$$

### 6.3.2 Formulas

The formula `true` is always interpreted to be *true*:

$$(R, s, h) \models \texttt{true}$$

The formula `emp` asserts that the heap is empty:

$$(R, s, h) \models \texttt{emp} \quad \text{iff} \quad R = \emptyset$$

The formula $lt \overset{\vec{pf},\vec{df}}{\longmapsto} (\vec{lt}, \vec{it})$ asserts that the heap contains exactly one record consisting of fields $\vec{pf}$ and $\vec{df}$, at address $lt$, with values $\vec{lt}$ and $\vec{it}$, respectively. Formally, the semantics of this formula is given as:

$$\begin{aligned} (R, s, h) \models lt \overset{\vec{pf},\vec{df}}{\longmapsto} (\vec{lt}, \vec{it}) \quad \text{iff} \quad & R = \{[\![lt]\!]_{R,s,h}\} \text{ and} \\ & h([\![lt]\!]_{R,s,h}, pf_i) = [\![lt_i]\!]_{R,s,h} \quad \text{for corresponding } pf_i \text{ and } lt_i, \\ & h([\![lt]\!]_{R,s,h}, df_i) = [\![it_i]\!]_{R,s,h} \quad \text{for corresponding } df_i \text{ and } it_i. \end{aligned}$$

Note that, as in separation logic, the above has a *strict semantics*— the heaplet must be a singleton set and cannot be a larger set.

For binary relations $t \sim t'$ between integers, sets, and multisets, including equality, the pure property plays an important role. Remember that in *SL* all terms are pure. To be consistent with *SL*, if both $t$ and $t'$ are pure, it is interpreted in the normal way. Otherwise, $t \sim t'$ is only defined on the minimum heaplet required by $t$ and $t'$, more concretely the union of the heaplet associated with $t$ and $t'$.

$$(R,s,h) \models t \sim t' \quad \text{iff} \quad t \text{ or } t' \text{ is pure and } [\![t]\!]_C \sim [\![t']\!]_C$$
$$\text{or} \quad t \text{ and } t' \text{ are impure and there exist } R_1, R_2 \text{ s.t. } R = R_1 \cup R_2$$
$$\text{and } [\![t]\!]_{C|R_1} \neq \texttt{undef}, [\![t']\!]_{C|R_2} \neq \texttt{undef} \text{ and } [\![t]\!]_{C|R_1} \sim [\![t']\!]_{C|R_2}$$

where $\sim$ is interpreted in the natural way.

The semantics of the disjoint conjunction operator $*$ is defined as follows. The formula $\varphi_0 * \varphi_1$ asserts that the heap can be split into two disjoint parts in which $\varphi_0$ and $\varphi_1$ hold respectively:

$$(R,s,h) \models \varphi_0 * \varphi_1 \quad \text{iff} \quad \text{there exist } R_0, R_1 \text{ s.t. } R_0 \cap R_1 = \emptyset \text{ and}$$
$$R_0 \cup R_1 = R \text{ and } (R_0, s, h|_{R_0}) \models \varphi_0 \text{ and } (R_1, s, h|_{R_1}) \models \varphi_1$$

Boolean combinations are defined in the standard way:

$$(R,s,h) \models \varphi_0 \wedge \varphi_1 \quad \text{iff} \quad (R,s,h) \models \varphi_0 \text{ and } (R,s,h) \models \varphi_1$$
$$(R,s,h) \models \varphi_0 \vee \varphi_1 \quad \text{iff} \quad (R,s,h) \models \varphi_0 \text{ or } (R,s,h) \models \varphi_1$$
$$(R,s,h) \models \neg\varphi \quad \text{iff} \quad (R,s,h) \not\models \varphi$$

### 6.3.3 Recursive Definitions

The main semantical difference between $\textsc{Dryad}_{\text{sep}}$ and *SL* is on recursive definitions. We would like to *deterministically delineate* the heap domain for any recursive definition, so that the heap domain required by any $\textsc{Dryad}_{\text{sep}}$ formula can be *syntactically determined*. Given a recursive definition $rec^{\Delta}_{\overrightarrow{pf},\vec{v}}$, the subscripts $\overrightarrow{pf}$ and $\vec{v}$ play a role in delineating the heap domain. Intuitively, the heap domain for $rec^{\Delta}_{\overrightarrow{pf},\vec{v}}(l)$ is the set of locations reachable from $l$ using pointer-fields in $\overrightarrow{pf}$, but *without* going through the locations $\vec{v}$. In other words, we want to take the set of locations that lie *in between* $l$ and $\vec{v}$. Precisely, this set is determined by a location $l$ and a program state $(R,s,h)$. We denote it as $\text{reachset}_{\overrightarrow{pf},\vec{v}}(l,(R,s,h))$. Formally it is the smallest set of locations $L$ satisfying the following two conditions:

1. $l$ is in the set $L$ if $l$ is not in $\vec{v}$ and $l \neq nil$;

2. for each $c$ in $L$, with $c \in R$, and for each pointer $pf$, if $h(c, pf)$ is not in $\vec{v}$ and is not $nil$, then $h(c, pf)$ is also in $L$.

**Remark:** Even though the reach set is defined with respect to the edges in the heaplet, we can determine whether $R$ includes all nodes reachable from $l$ without going through $\vec{v}$ in the *global heap* by checking whether $R = \text{reachset}_{\vec{pf},\vec{v}}(l, (R, s, h))$.

For each recursive definition $rec^{\Delta}_{\vec{pf},\vec{v}}$, we usually simply denote $\text{reachset}_{\vec{pf},\vec{v}}$ as $\text{reachset}^{rec}$, as the subscripts are implicitly known.

We first give some intuition on the semantics of recursive definitions. Given a program state $C = (R, s, h)$ and a recursive function/predicate $rec^{\Delta}$, the semantics on a location $l$ depends on whether the heap domain $R$ is exactly the required reach set $\text{reachset}^{rec}(l, (R, s, h))$. If this is not true, we simply interpret it as `undef` or `false`. If the heap domain matches the reach set (i.e., $R = \text{reachset}_{\vec{pf},\vec{v}}(l, (R, s, h))$), the semantics is defined in the natural way (using least fixed-points).

In order to give least fixed-point semantics for recursive definitions in the logic, we extend the primitive data types to lattice domains. *Bool* with the order `false` $\sqsubseteq$ `true` forms a complete lattice, and $\mathcal{S}(Loc)$ and $\mathcal{S}(Int)$ ordered by inclusion, with join as union and meet as intersection, form complete lattices. Integers and multisets are extended to lattices. Let $(Int_L, \leq)$ denote the complete lattice, where $Int_L = Int \cup \{-\infty, \infty\}$, and where the ordering is $\leq$, join is *max*, meet is *min*. Also, $\mathcal{MS}(Int)_L, \sqsubseteq$ denote the complete lattice constructed from $\mathcal{MS}(Int)$, where $\mathcal{MS}(Int)_L = \mathcal{MS}(Int) \cup \{\top\}$, and $\sqsubseteq$ extends the inclusion relation with $S \sqsubseteq \top$ for any $M \in \mathcal{MS}(Int)$. It is easy to see that $(Int_L, \sqsubseteq)$ and $(\mathcal{MS}(Int)_L, \sqsubseteq)$ are complete lattices.

Formally, let *Def* consists of definitions of integer functions $I$, set-of-locations functions *SL*, set-of-integers functions *SI*, multiset-of-integers functions *MSI* and predicates $P$. Since these definitions could rely on each other, we evaluate them altogether as a function vector

$$r^{\Delta} = (\overrightarrow{i^{\Delta}}, \overrightarrow{sl^{\Delta}}, \overrightarrow{si^{\Delta}}, \overrightarrow{msi^{\Delta}}, \overrightarrow{p^{\Delta}})$$

Let $\text{select}_{rec}(r^{\Delta})$, for each recursive definition $rec^{\Delta}$, denote the selection of the coordinate for $rec^{\Delta}$ in $r^{\Delta}$. Then the semantics of each single function $rec^{\Delta}$ is just $\text{select}_{rec}(r^{\Delta})$. Since $(Int_L, \sqsubseteq)$, $(\mathcal{S}(Loc), \sqsubseteq)$, $(\mathcal{S}(Int), \sqsubseteq)$, $(\mathcal{MS}(Int)_L, \sqsubseteq)$

and $(Bool, \sqsubseteq)$ are all complete lattices, for each domain $Loc^k$ and each type $\mathcal{C}$, we can define a structure $(Loc^k \to \mathcal{C}, \sqsubseteq)$, where $\sqsubseteq$ is defined as follows: for any two functions $r, r' : Loc^k \to \mathcal{C}$, $r \sqsubseteq r'$ if and only if for any $(l_1, \ldots, l_k) \in Loc^k$, $r(l_1, \ldots, l_k) \sqsubseteq r'(l_1, \ldots, l_k)$. Then the product of $(Loc^k \to \mathcal{C}_{rec}, \sqsubseteq)$ for all $rec^\Delta$ also forms a complete lattice, where $\sqsubseteq$ is the product order of all the component $\sqsubseteq$. Let each $rec$ be a mapping from $\mathcal{D}_{rec}$ to $\mathcal{C}_{rec}$, then $\big(\prod_{rec}(\mathcal{D}_{rec} \to \mathcal{C}_{rec}),\ \sqsubseteq\big)$ is just the Cartesian product of all the component compete lattices, and is still complete.

**Proposition 6.3.2.** $\big(\prod_{rec}(\mathcal{D}_{rec} \to \mathcal{C}_{rec}),\ \sqsubseteq\big)$ *is a complete lattice.*

*Proof.* By the fact that the Cartesian product of complete lattices is still a complete lattice. ☐

Now, to give the least fixed-point semantics to $r^\Delta$, we first need to define an operator $\mathscr{U}_C$ over $\prod_{rec}(\mathcal{D}_{rec} \to \mathcal{C}_{rec})$. Since the codomain of $\mathscr{U}_C$ is just a product of functions from $\mathcal{D}_{rec} \to \mathcal{C}_{rec}$ for each recursive definition $rec^\Delta$, it suffices to define each component operator

$$\mathscr{U}_{rec} \ :\ \prod_{rec}(\mathcal{D}_{rec} \to \mathcal{C}_{rec}) \ \to\ \big(\mathcal{D}_{rec} \to \mathcal{C}_{rec}\big)$$

Then simply

$$\mathscr{U}_C(V) \ \stackrel{def}{=}\ \prod_{rec}\big(\mathscr{U}_{rec}(V)\big)$$

where $V \in \prod_{rec}(\mathcal{D}_{rec} \to \mathcal{C}_{rec})$.

Let $C = (R, s, h)$ be a computation state, let $r$ be a function vector in $\prod_{rec}(\mathcal{D}_{rec} \to \mathcal{C}_{rec})$, to define $\mathscr{U}_{rec}(r)$, consider two cases: $rec^\Delta$ is a recursive function; or $rec^\Delta$ is a recursive predicate.

Consider a recursively defined function

$$f^\Delta(x, \vec{s}) \ \stackrel{def}{=}\ \big(\varphi^f_1(x, \vec{v}, \vec{s}) : t^f_1(x, \vec{s}) \ ;\ \ \ldots\ ;\\ \varphi^f_k(x, \vec{v}, \vec{s}) : t^f_k(x, \vec{s}) \ ;\ \text{default} : t^f_{k+1}(x, \vec{s})\big)$$

we first translate the colon-separated definition into a nested if-then-else (ITE) operator. Formally,

$$Def_f \ \stackrel{def}{=}\ \text{ITE}\big(\varphi^f_1(x, \vec{v}, \vec{s}), t^f_1(x, \vec{s}),\ \text{ITE}(\varphi^f_2(x, \vec{v}, \vec{s}), t^f_2(x, \vec{s}), \ldots t^f_{k+1}(x, \vec{s}) \ldots)\big)$$

Secondly, we compute the scope $D = \text{reachset}_f(x, \vec{s}, h)$. Now we define

$$\mathscr{U}_f(r)(x, \vec{v}) = [\![ \ Def_f(x, \vec{v}, x(\vec{s})) \ ]\!]_{\left(R \cap D, \ x(\vec{s}), \ h|D\right)}$$

where $x(\vec{s})$ replaces each $s$ with corresponding dereference of $x$.

The above evaluation could involve evaluating some $[\![ f\!f(lt) ]\!]_{C'}$ or $[\![ pp(lt) ]\!]_{C'}$ where $C' = (R', s', h')$ such that $R' \subseteq R$, $s' = s(\vec{v})$, and $h' = h|R'$. $[\![ f\!f(lt) ]\!]_{C'}$ will be evaluated as follows:

$$[\![ f\!f(lt) ]\!]_{R',s',h'} = \begin{cases} \text{select}_{f\!f}\big(r\big)([\![ lt ]\!]'_C) & \text{if} \ \ R' = \text{reachset}_{f\!f}([\![ lt ]\!]_{C'}, h') \\ \texttt{undef} & \text{otherwise} \end{cases}$$

$[\![ pp(\vec{lt}) ]\!]'_C$ will be evaluated similarly:

$$[\![ pp(lt) ]\!]_{R',s',h'} = \begin{cases} \text{select}_{pp}\big(r\big)([\![ lt ]\!]'_C) & \text{if} \ \ R' = \text{reachset}_{pp}([\![ lt ]\!]_{C'}, h') \\ \texttt{false} & \text{otherwise} \end{cases}$$

Similarly, consider a recursively defined predicate $p^{\Delta}(x, \vec{s}) \ \stackrel{def}{=} \ \varphi^p(x, \vec{v}, \vec{s})$, we first compute the scope $D = \text{reachset}_p(x, \vec{s}, h)$. Then we define

$$\mathscr{U}_p(r)(x, \vec{v}) = [\![ \ \varphi^p(x, \vec{v}, x(\vec{s})) \ ]\!]_{\left(R \cap D, \ x(\vec{s}), \ h|D\right)}$$

**Theorem 6.3.3.** *For each computation state $C$, $\mathscr{U}_C$ admits a least fixed-point w.r.t. $\sqsubseteq$.*

*Proof.* Note that we disallow negative operations (subtraction, set-difference and negation) in defining recursive definitions. This syntactical restriction guarantees that for each recursive definition *rec* and for each $r \in \prod_{rec}(\mathcal{D}_{rec} \to \mathcal{C}_{rec})$, $\text{select}_{rec}(r) \sqsubseteq \mathscr{U}_{rec}(r)$. Since $\mathscr{U}_C$ is just the product of all $\mathscr{U}_{rec}$, it is clear that $\text{select}_C(r) \sqsubseteq \mathscr{U}_{rec}(r)$, i.e., $\mathscr{U}_{rec}$ is monotonic w.r.t. $\sqsubseteq$. By Knaster-Tarski theorem, $\mathscr{U}_C$ admits a least fixed-point. ☐

Now we can formally define the semantics of recursive definitions. For any configuration $C$, the semantics of a recursive function $f^{\Delta}$ is defined as:

$$
[\![ f^{\Delta}(lt, \vec{st}) ]\!]_C = \begin{cases} \text{select}_f\big(\mathit{lfp}(\mathscr{U}_C)\big)([\![lt]\!]_C, [\![\vec{st}]\!]_C) \\ \qquad\qquad\qquad \text{if} \quad R = \text{reachset}^f([\![lt]\!]_C, [\![\vec{st}]\!]_C, C) \\ \texttt{undef} \qquad\qquad \text{otherwise} \end{cases}
$$

and the semantics of a recursive predicate $p^{\Delta}$ is defined as

$$
[\![ p^{\Delta}(lt, \vec{st}) ]\!]_C = \begin{cases} \text{select}_p\big(\mathit{lfp}(\mathscr{U}_C)\big)([\![lt]\!]_C, [\![\vec{st}]\!]_C) \\ \qquad\qquad\qquad \text{if} \quad R = \text{reachset}^p([\![lt]\!]_C, [\![\vec{st}]\!]_C, C) \\ \texttt{false} \qquad\qquad \text{otherwise} \end{cases}
$$

## 6.4 Examples

In this section, we give several examples to show how DRYAD$_{\text{sep}}$ can characterize data-structures recursively, and its subtle difference from classical SL.

A max-heap is a binary tree such that for each node $n$ the key stored at $n$ is greater than or equal to the keys stored at each of its children. Heaps are often used to implement priority queues. As below, we express the property that a location $x$ points to a max-heap using recursive definitions $\mathit{keys}^{\Delta}_{\vec{pf}}(x)$ and $\mathit{mheap}^{\Delta}_{\vec{pf}}(x)$, with $\vec{pf} \equiv \{\mathit{left}, \mathit{right}\}$.

$$
\begin{aligned}
\boldsymbol{mheap}^{\Delta}_{\vec{pf}}(x) \quad &\stackrel{def}{=} \quad \Big( \big(x = \texttt{nil} \wedge \texttt{emp}\big) \ \vee \\
&\qquad \big( \ x \stackrel{key,left,right}{\longmapsto} (k,\ l,\ r) \\
&\qquad\quad * \ (\mathit{mheap}^{\Delta}_{\vec{pf}}(l) \wedge \{k\} \geq \mathit{keys}^{\Delta}_{\vec{pf}}(l)) \\
&\qquad\quad * \ (\mathit{mheap}^{\Delta}_{\vec{pf}}(r) \wedge \{k\} \geq \mathit{keys}^{\Delta}_{\vec{pf}}(r)) \ \big) \ \Big) \\
\boldsymbol{keys}^{\Delta}_{\vec{pf}}(x) \quad &\stackrel{def}{=} \quad \Big( x = \texttt{nil} \wedge \texttt{emp} : \emptyset \ ; \\
&\qquad x \stackrel{key,left,right}{\longmapsto} (k,\ l,\ r) * \texttt{true} : \mathit{keys}^{\Delta}_{\vec{pf}}(l) \cup \{k\} \cup \mathit{keys}^{\Delta}_{\vec{pf}}(r) \ ; \\
&\qquad \text{default} : \emptyset \Big)
\end{aligned}
$$

For a location $x$, the recursive definition $\mathit{keys}^{\Delta}_{\vec{pf}}(x)$ returns the set of keys at the nodes of the tree rooted at $x$: if $x$ is *nil*and the heaplet is empty,

then the empty-set; otherwise, the union of the key stored at $x$ and the keys stored in the left and right subtrees of $x$. Similarly, the recursive definition $mheap^{\Delta}_{\vec{pf}}(x)$ states that $x$ points to a max-heap if: $x$ is $\texttt{nil}$ and the heaplet is empty; or $x$ and the heaplets of the left and right subtrees of $x$ are mutually disjoint ($x$ points to a tree) and the key at $x$ is greater than or equal to the keys of the left and right subtrees of $x$.

Note that the definition of a max-heap is precisely defined on the heaplet that includes the underlying tree nodes of the max-heap only, as the heaplet for a recursive definition is the set of all reachable nodes according to the two pointers.

To clarify the difference between $\text{DRYAD}_{\text{sep}}$ and *SL*, consider now this recursive definition:

$$p^{\Delta}_{\{l,r\}}(x) \stackrel{def}{=} (x = \texttt{nil} \wedge \texttt{emp}) \quad \vee \quad \left[(x \xrightarrow{l,r} y, z) \; * \; \left(p^{\Delta}_{\{l,r\}}(y) \vee p^{\Delta}_{\{l,r\}}(z)\right)\right]$$

Consider a global heap that has a tree rooted at $x$ with pointer fields $l$ and $r$. The above recursive formula, in separation logic, will be true on any heaplet that contains the nodes of a path in this tree from $x$ to *nil*. However, in $\text{DRYAD}_{\text{sep}}$, we require that the heaplet must satisfy the heap constraints of the formula and also be the precise set of locations reachable from $x$ using the pointer fields $l$ and $r$. Consequently, if the tree pointed to by $x$ has more than one path, the $\text{DRYAD}_{\text{sep}}$ formula will be *false for any heaplet*.

The above example shows the advantage of $\text{DRYAD}_{\text{sep}}$. while nondetermined heaplets are inherent in SL, in $\text{DRYAD}_{\text{sep}}$, the heap domain is precisely determined, and we can avoid quantification. This quantifier-free feature is very useful when we check a verification condition along the lines of $p^*(x) \wedge \ldots \Rightarrow p^*(t)$ (a precondition implying a postcondition), the corresponding satisfiability query would be $p^*(x) \wedge \ldots \wedge \neg p^*(t)$. The negated formula $\neg p^*(t)$ hence requires a *universal* quantification over paths in the tree if we went with the usual separation logic semantics, which is very hard to handle automatically. In contrast, we have not found natural examples where an undetermined heaplet semantics helps in specifying properties of heaps.

Consequently, in our semantics, the heap domain for recursive definitions is syntactically determined to be a fixed set, based only on its signature,

making Dryad$_{\text{sep}}$ amenable to being translated to quantifier-free classical logic.

Dryad$_{\text{sep}}$ can express structures beyond trees. The main restriction we do impose is that we allow only *unary* recursive definitions, as this allows us to find simpler natural proofs since there is only one way to unfold the definition across a footprint. However, Dryad$_{\text{sep}}$ can express structures like cyclic lists and doubly-linked lists.

A cyclic-list is captured as $(v \mapsto y) * lseg^{\Delta}_{next,v}(y)$. Here, $v$ is a program variable which denotes the head of the cyclic-list and $lseg^{\Delta}_{next,v}(y)$ captures the list segment from $y$ back to the head $v$, where the subscripts *next* and $v$ indicate that the heaplet of the list segment is the locations that can be reached using the field *next*, but without going through $v$:

$$lseg^{\Delta}_{next,v}(y) \stackrel{def}{=} (y = v \wedge \texttt{emp}) \vee \big( (y \stackrel{next}{\mapsto} z) * lseg^{\Delta}_{next,v}(z) \big)$$

Another interesting example is a doubly-linked list. We define a doubly-linked list as the following unary predicate:

$$\begin{aligned} dll^{\Delta}_{next}(x) \quad = \quad & (x = \texttt{nil} \wedge \texttt{emp}) \;\; \vee \;\; (x \stackrel{next}{\mapsto} \texttt{nil}) \;\; \vee \\ & \big( \, x \stackrel{next}{\mapsto} y \; * \; \big((y \stackrel{prev}{\mapsto} x * \texttt{true}) \; \wedge \; dll^{\Delta}_{next}(y)\big) \, \big) \end{aligned}$$

The first two disjuncts in the definition cover the base case when $x$ is *nil* or the location next to $x$ is *nil*; otherwise, let $y$ be the location next to $x$, then the *prev* pointer at $y$ points to $x$ and location $y$ is recursively defined as a doubly-linked list.

## 6.5 Translating to A Logic over the Global Heap

We now show one of the main contributions of this chapter— a translation from Dryad$_{\text{sep}}$ logic to classical logic with recursive predicates and functions, but over the global heap. The formulation of separation logic primitives in the global heap allows us to express complex structural properties, like disjointness of heaplets and tree-ness, using recursive definitions over *sets of locations*, which are defined locally, and are amenable to unfolding across the footprint and hence amenable to natural proofs.

For example, consider the formula $mheap^{\Delta}(x) * mheap^{\Delta}(y)$, where $mheap^{\Delta}$ is defined in Section 7.5. Since the heaplets for $mheap^{\Delta}(x)$ and $mheap^{\Delta}(y)$ are precise, it can get translated to an equivalent formula with a *free* set variable $G$ that denotes the global heap over which the formula is evaluated:

$$\begin{aligned} mheap(x) \wedge mheap(y) \wedge (reach^{mheap}(x) \cap reach^{mheap}(y) = \emptyset) \\ \wedge\ (reach^{mheap}(x) \cup reach^{mheap}(y) = G) \end{aligned}$$

where *mheap* and $reach^{mheap}$ are corresponding recursive definitions in classical logic, which will be defined later in this section. Note that we use italics and remove the $^{\Delta}$ superscript to show the difference from their counterpart in $\text{DRYAD}_{\text{sep}}$.

Since these theories can be handled by present SMT solvers, this translation leads to a robust way of deciding separation logic formulas in an automatic and a terminating manner by making calls to the underlying theory solvers.

### 6.5.1 Preprocessing

We assume the $\text{DRYAD}_{\text{sep}}$ formula to be translated is in *disjunctive normal form*, i.e., $\vee$ operators should be above all $*$ and $\wedge$ operators. This is not a real restriction as one can always push the disjunction out. This normal form ensures that for all occurrences of the separation operator in a formula, there exists a unique way of splitting the heap so as to satisfy the $*$ separated sub-formulas. Also, it ensures that this unique heap-split can be determined syntactically from the structure of those sub-formulas.

In our translation, we model the heaplets associated with a formula or a term as a set of locations and all operations on these heaplets are modeled as set operations like set union, set intersections, etc. over set-of-location variables. For example the separating conjunction $P * Q$ is translated to the following set constraint: the intersection of the sets associated with the heaplets in the formulas $P$ and $Q$ is empty. Given a formula $\varphi$ in $\text{DRYAD}_{\text{sep}}$ and its associated heap domain modeled by a set variable $G$, we define an inductive translation $T$ into a classical logic formula $T(\varphi,\ G)$ in the quantifier-free theory of finite sets, integers and uninterpreted functions. The translated

| Construct | Domain-exact | Scope |
|---|---|---|
| $var/const$ | `false` | $\emptyset$ |
| $\{t\}/\{t\}_m$ | dom-ext(t) | scope(t) |
| $t$ op $t'$ | $\text{dom-ext}(t) \vee \text{dom-ext}(t')$ | $\text{scope}(t) \cup \text{scope}(t')$ |
| $f^{\Delta}(lt)$ | `true` | $\text{reachset}^{f}(lt)$ |
| `true/false` | `false` | $\emptyset$ |
| `emp` | `true` | $\emptyset$ |
| $lt \overset{\vec{pf},\vec{df}}{\longmapsto} (\vec{lt}, \vec{it})$ | `true` | $\{lt\}$ |
| $p^{\Delta}(lt)$ | `true` | $\text{reachset}^{p}(lt)$ |
| $t \sim t'$ | $\text{dom-ext}(t) \vee \text{dom-ext}(t')$ | $\text{scope}(t) \cup \text{scope}(t')$ |
| $\varphi \wedge \varphi'$ | $\text{dom-ext}(\varphi) \vee \text{dom-ext}(\varphi')$ | $\text{scope}(\varphi) \cup \text{scope}(\varphi')$ |
| $\varphi * \varphi'$ | $\text{dom-ext}(\varphi) \wedge \text{dom-ext}(\varphi')$ | $\text{scope}(\varphi) \cup \text{scope}(\varphi')$ |

Table 6.1: Domain-exact property and Scope function

formula is not interpreted on a heaplet, but interpreted on a global heap (i.e., with the heap domain *Loc*).

The translation uses an auxiliary *domain-exact* property and an auxiliary *scope* function. The domain-exact property indicates whether a term evaluates to a well-defined value or a positive formula evaluates to true on a fixed heap domain or not. This is different from the property *pure*; a pure formula or term is not domain-exact but the reverse implication is not true, in general. For example, the formula $(lt \longmapsto it) * true$ is not domain-exact but is also not pure. The scope function maps a term to the minimum heap domain required to evaluate it to a normal value, and maps a positive formula to the minimum heap domain required to evaluate it to *true*. The domain-exact property and the scope function are defined inductively in Table 6.1. Note that both are defined only for terms and formulas without disjunction and negation. A formula is assumed in its disjunctive normal form.

### 6.5.2 Translating Terms and Formulas

We describe the logic translation of $\text{D}\text{RYAD}_{\text{sep}}$ terms and formulas in Figure 6.3 and Figure 6.4, respectively. The ITE expression used in the translation is short for "if-then-else". It is just a conditional expression defined as follows: $\text{ITE}(\phi, t_1, t_2)$ evaluates to $t_1$ if $\phi$ is true, otherwise evaluates to $t_2$.

In general, our translation restricts an impure term/formula to be evaluated only on the syntactically determined heap domain according to the se-

$$
\begin{array}{rcl}
T(var \;/\; const,\ G) & \equiv & var \;/\; const \\
T(\{t\} \;/\; \{t\}_m,\ G) & \equiv & \{t\} \;/\; \{t\}_m \\
T(t \ \ \mathrm{op}\ \ t',\ G) & \equiv & T(t,G)\ \ \mathrm{op}\ \ T(t',G) \\
T(f^{\Delta}(lt),\ G) & \equiv & \mathrm{ITE}\left(reach^{f}(lt) = G,\ f(lt),\ \mathtt{undef}\right) \\
T(\mathtt{true} \;/\; \mathtt{false},\ G) & \equiv & \mathtt{true} \;/\; \mathtt{false} \\
T(\mathtt{emp},\ G) & \equiv & G = \emptyset \\
T(lt \overset{\vec{pf},\vec{df}}{\longmapsto} (\vec{lt},\vec{it}),\ G) & \equiv & G = \{lt\} \wedge \bigwedge_{pf_i}\ pf_i\big(T(lt,G)\big) = T(lt_i,G) \\
 & & \qquad \wedge \bigwedge_{df_i}\ df_i\big(T(lt,G)\big) = T(it_i,G) \\
T(p^{\Delta}(lt),\ G) & \equiv & p(lt) \wedge G = reach^{p}(lt) \\
T(t \sim t',\ G) & \equiv & \begin{cases} t \sim t' & \text{if}\ \ t \sim t' \text{ is not domain-exact} \\ t \sim t' \wedge G = \mathrm{scope}(t \sim t') & \text{otherwise} \end{cases} \\
T(\varphi \wedge \varphi',\ G) & \equiv & T(\varphi,G) \wedge T(\varphi',G) \\
T(\varphi \vee \varphi',\ G) & \equiv & T(\varphi,G) \vee T(\varphi',G) \\
T(\neg\varphi,\ G) & \equiv & \neg T(\varphi,G)
\end{array}
$$

Figure 6.3: Translation of Dryad$_{\mathrm{sep}}$ terms

$$
T(\varphi * \varphi',\ G) \ \equiv \ \begin{cases}
T\big(\varphi,\ \mathrm{scope}(\varphi)\big)\ \wedge\ T\big(\varphi',\ \mathrm{scope}(\varphi')\big) \\
\quad \wedge\ \mathrm{scope}(\varphi) \cup \mathrm{scope}(\varphi') = G \\
\quad \wedge\ \mathrm{scope}(\varphi) \cap \mathrm{scope}(\varphi') = \emptyset \\
\qquad\qquad\qquad\qquad \text{if both } \varphi \text{ and } \varphi' \text{ are domain-exact} \\
T\big(\varphi,\ \mathrm{scope}(\varphi)\big)\ \wedge\ T\big(\varphi',\ G \setminus \mathrm{scope}(\varphi)\big) \\
\quad \wedge\ \mathrm{scope}(\varphi) \subseteq G \qquad \text{if only } \varphi \text{ is domain-exact} \\
T\big(\varphi',\ \mathrm{scope}(\varphi')\big)\ \wedge\ T\big(\varphi,\ G \setminus \mathrm{scope}(\varphi')\big) \\
\quad \wedge\ \mathrm{scope}(\varphi') \subseteq G \qquad \text{if only } \varphi' \text{ is domain-exact} \\
T\big(\varphi,\ \mathrm{scope}(\varphi)\big)\ \wedge\ T\big(\varphi',\ \mathrm{scope}(\varphi')\big) \\
\quad \wedge\ \mathrm{scope}(\varphi) \cup \mathrm{scope}(\varphi') \subseteq G \\
\quad \wedge\ \mathrm{scope}(\varphi) \cap \mathrm{scope}(\varphi') = \emptyset \\
\qquad\qquad\qquad\qquad \text{if neither } \varphi \text{ nor } \varphi' \text{ is domain-exact}
\end{cases}
$$

Figure 6.4: Translation of Dryad$_{\mathrm{sep}}$ formulas

mantics of $\textsc{Dryad}_{\text{sep}}$. In particular, when evaluating a recursive formula or predicate $p^{\Delta}$, we ensure that the heaplet is precisely the reach set $reach^{p}(lt)$. For the formula `emp` evaluated on heaplet G, the translation asserts that G is the empty set of locations. Similarly, for the atomic formula $lt \xrightarrow{\vec{pf},\vec{df}} (\vec{lt}, \vec{it})$, the classical logic formula asserts that G is the singleton set consisting of the location $lt$. For the atomic formula $t \sim t'$, the translation to classical logic depends on whether this formula is tight or not. If the formula is not tight, then it can be evaluated on any heaplet. Otherwise, the heaplet is restricted to the scope of $t \sim t'$. If the formula $\varphi \vee \varphi'$ is evaluated on a heaplet $G$, then either the sub-formula $\varphi$ is true on the heaplet $G$ or $\varphi'$ is true on $G$. The case when the formula is a conjunction $\varphi \wedge \varphi'$ is handled similarly. When the formula has negation as its top-level operator $\neg\varphi$, the formula is true over a heaplet $G$ if the positive sub-formula $\varphi$ evaluates to false over the same heaplet $G$.

For a formula $\varphi * \varphi'$, translation to classical logic depends on whether the sub-formulas $\varphi$ and $\varphi'$ are domain-exact or not. If a sub-formula is domain-exact then it is evaluated on its scope. If it is not domain-exact, then it is evaluated on the rest of the heaplet. Specifically, if neither $\varphi$ nor $\varphi'$ is tight, the translated classical logic formula requires the scopes of $\varphi$ and $\varphi'$ to be disjoint from each other such that their disjoint union is the heaplet $G$.

### 6.5.3 Translating Recursive Definitions

Recursive definitions in $\textsc{Dryad}_{\text{sep}}$ are also translated to recursive definitions in classical logic. Translating a recursive definition $rec^{\Delta}$ uses the corresponding definitions $rec$ and $reach^{rec}$, both of which are defined recursively in classical logic. The set $reach^{rec}$ represents the domain of the required heaplet for evaluating $rec^{\Delta}$, and the $\Delta$-eliminated definition $rec$ captures the value of $rec^{\Delta}$ when the heaplet is restricted to $reach^{rec}$. Formally, suppose $rec^{\Delta}$ is a recursive definition w.r.t. pointer fields $\vec{pf}$ and stopping locations $\vec{v}$, then $reach^{rec}$ is recursively defined as the least fixed-point of

$$reach^{rec}(x, \vec{v}) \stackrel{def}{=} \text{ITE}\Big( x = \texttt{nil} \vee x \in \vec{v},\ \emptyset,\ \{x\} \cup \bigcup_{pf \in \vec{pf}} \big(reach^{rec}(pf(x))\big) \Big)$$

For each recursive predicate $p^\Delta$ defined as $p^\Delta(x) \stackrel{def}{=} \varphi^p(x, \vec{v}, \vec{s})$, we define

$$p(x, \vec{v}) \stackrel{def}{=} T\big(\ \varphi^p(x, \vec{v}, \vec{s}),\ reach^p(x, \vec{v})\ \big)$$

Similarly, for each recursive function $f^\Delta$ defined as

$$f^\Delta(x, \vec{v}) \stackrel{def}{=} \big(\varphi_1^f(x, \vec{v}, \vec{s}) : t_1^f(x, \vec{s})\ ;\ \varphi_2^f(x, \vec{v}, \vec{s}) : t_2^f(x, \vec{s})\ ;\ \ldots\ ;\ \text{default} : t_{k+1}^f(x, \vec{s})\big)$$

we define

$$f(x, \vec{v}) \stackrel{def}{=} \text{ITE}\Big(\ T\big(\varphi_1^f(x, \vec{v}, \vec{s}), reach^f(x)\big),\ \ t_1^{f-\Delta}(x, \vec{s})\ \ \text{ITE}\Big(\ T\big(\varphi_2^f(x, \vec{v}, \vec{s}), reach^f(x)\big),\ \ t_2^{f-\Delta}(x, \vec{s})\ \ \ldots,\ t_{k+1}^{f-\Delta}(x, \vec{s})\ \Big)\ldots\Big)\Big)$$

where $t_i^{f-\Delta}(x, \vec{s})$ is just the classical logic counterpart of $t_i^f(x, \vec{s})$, when interpreted in a heap domain within $reach^f(x)$. Formally it is short for

$$\text{ITE}\Big(\text{scope}(t_i^f(x, \vec{s})) \subseteq reach^f(x),\ T\big(t_i^f(x, \vec{s}),\ \text{scope}(t_i^f(x, \vec{s}))\big),\ \texttt{undef}\Big)$$

Now for each set of recursive definitions $Def^\Delta$ in $\text{DRYAD}_{\text{sep}}$, we can translate it to a set of recursive definitions $Def$ in classical logic. The semantics of the definitions in $Def$ is the least fixed point of the combination of the equations in each definition, similar to the semantics of $Def^\Delta$. The following lemma shows that each definition in $Def$ precisely interprets the heaplet semantics (defined only on the reachable locations) of the corresponding definition in $Def^\Delta$.

**Lemma 6.5.1.** *Let $Def^\Delta$ be a set of recursive definitions in $\text{DRYAD}_{sep}$, and let Def be the set of translated recursive definitions in classical logic. Let $p^\Delta$ be an arbitrary recursive predicate and $f^\Delta$ be an arbitrary recursive function in $Def^\Delta$. Then for every program state $(Loc, s, h)$, for every locations $x$ and $\vec{v}$, $(Loc, s, h) \models p(x, \vec{v})$ if and only if $(R^p, s, h|_{R^p}) \models p^\Delta(x, \vec{v})$, and $[\![f(x, \vec{v})]\!]_{(Loc,s,h)} = [\![f^\Delta(x, \vec{v})]\!]_{(R^f,s,h|_{R^f})}$, where $R^p = reach^p(x, \vec{v})$, $R^f = reach^f(x, \vec{v})$.*

*Proof.* Since both the definitions in $\mathit{Def}$ and $\mathit{Def}^{\Delta}$ has the least-fixed-point semantics, the proof intuitively consists of two steps: first show the semantics at the bottom of the two lattices are equivalent; then show the equivalence is preserved in each iteration of applying the recursive operator.

For the first step, notice that for each predicate $p$, $p(x, \vec{v})$ and $p^{\Delta}(x, \vec{v})$ are always interpreted as `false` (no matter $p^{\Delta}$ interpreted on which heaplet). Similarly, for each function $f$, both $p(x, \vec{v})$ and $p^{\Delta}(x, \vec{v})$ (interpreted on $R^f$) are defined as the bottom of the corresponding lattice.

For each iteration that transits the predicate $p$ to $p'$, and transits $p^{\Delta}$ to $p'^{\Delta}$. Assume that the $p$ and $p^{\Delta}$ are equivalent. Then for every locations $x$ and $\vec{v}$, $(\mathit{Loc}, s, h) \models p'(x, \vec{v})$ iff $\vec{v}$, $(\mathit{Loc}, s, h) \models T\big(\ \varphi^p(x, \vec{v}, x(\vec{s})),\ \mathit{reach}^p(x, \vec{v})\ \big)$. Moreover, $(R^p, s, h|_{R^p}) \models p'^{\Delta}(x, \vec{v})$ iff $(R^p, s, h|_{R^p}) \models \varphi^p(x, \vec{v}, x(\vec{s}))$. Hence it suffices to prove inductively that $(\mathit{Loc}, s, h) \models T\big(\ \varphi^p(x, \vec{v}, x(\vec{s})),\ \mathit{reach}^p(x, \vec{v})\ \big)$ iff $(R^p, s, h|_{R^p}) \models \varphi^p(x, \vec{v}, x(\vec{s}))$.

Similarly, assuming that the $f$ and $f^{\Delta}$ are equivalent, the same iteration transits them to $f'$ and $f'^{\Delta}$, which should also be equivalent. Formally, let the transitions be $f'^{\Delta}(x, \vec{v}) = \mathrm{DryadDef}^f$ and $f'(x, \vec{v}) = \mathrm{GlobalDef}^f$, then the goal is to show $[\![\mathrm{DryadDef}^f(x, \vec{v})]\!]_{(\mathit{Loc},s,h)} = [\![\mathrm{GlobalDef}^f(x, \vec{v})]\!]_{(R^f,s,h|_{R^f})}$

With the assumption that $p$ and $p^{\Delta}$ ($f$ and $f^{\Delta}$) are equivalent, both the two goals can be proved together, by induction on the structure of $\varphi^p$ ($\mathrm{DryadDef}^f$). For each construct from DRYAD$_{\mathrm{sep}}$, its heaplet semantics is equivalenty translated to the classical logic. We omit the details and leave the reader to verify. In particular, when $\mathrm{DryadDef}^f$ is of the form $\big(\varphi_1^f(x, \vec{v}, \vec{s}) : t_1^f(x, \vec{s})\ ;\ \ldots\ ;\ \mathrm{default} : \ldots\big)$, by induction, $(R^p, s, h|_{R^p}) \models t_1^f(x, \vec{s})$ if and only if $(\mathit{Loc}, s, h) \models t_1^{f-\Delta}(x, \vec{s})$. When it is the case,

$$
\begin{aligned}
& [\![\mathrm{DryadDef}^f(x, \vec{v})]\!]_{(\mathit{Loc},s,h)} \\
= \ & [\![T\big(\varphi_1^f(x, \vec{v}, \vec{s}), \mathit{reach}^f(x)\big)]\!]_{(\mathit{Loc},s,h)} \\
= \ & [\![\varphi_1^f(x, \vec{v}, \vec{s})]\!]_{(R^f,s,h|_{R^f})} \\
= \ & [\![\mathrm{GlobalDef}^f(x, \vec{v})]\!]_{(R^f,s,h|_{R^f})}
\end{aligned}
$$

Otherwise, we can similarly proceed to prove the equivalence between the second cases, so on and so forth. □

We have shown the translation of the recursive definitions is sound, now it is straightforward to show the main result of this chapter: the translation of arbitrary DRYAD$_{\mathrm{sep}}$ formula to the classical logic is sound.

**Theorem 6.5.2.** *Let $\varphi$ be a* $\textsc{Dryad}_{sep}$ *formula w.r.t. a set of recursive definitions* $Def^{\Delta}$*. For every program state* $(Loc, s, h)$ *and for every heaplet* $G \subseteq Loc \setminus \{nil\}$*,* $(Loc, s, h) \models T(\varphi,\ G)$ *w.r.t.* $Def$ *if and only if* $(G, s, h|_G) \models \varphi$ *w.r.t.* $Def^{\Delta}$*.*

*Proof.* The proof is by induction on the structure of $\varphi$. For the basic constructs, the proof is very similar to the inductive proof in the second step of proving Lemma 6.5.1. We only prove the recursive predicate case here.

When $\varphi$ is a recursive predicate $p^{\Delta}(lt)$, $T(p^{\Delta}, G) = p(lt) \wedge G = reach^{p}(lt)$, so $(Loc, s, h) \models T(p^{\Delta}(lt),\ G)$ iff $(Loc, s, h) \models p(lt)$ and $(Loc, s, h) \models G = reach^{p}(lt)$. By Lemma 6.5.1 and the induction hypothesis, the condition is equivalent to $(R^{p}, s, h|_{R^{p}}) \models p^{\Delta}(lt)$, where $R^{p} = reach^{p}(lt) = G$. In other word, $(G, s, h|_G) \models p^{\Delta}(lt)$. $\square$

# CHAPTER 7

# NATURAL PROOFS FOR STRUCTURE, DATA, AND SEPARATION

The natural proof methodology has been proposed in Chapter 5, but was exclusively built for tree data-structures. In particular, this work could only handle *recursive* programs, i.e., no while-loops, and even for tree data-structures, imposed a large number of restrictions on pre/post conditions for methods— the input to a procedure had to be only a single tree, the method can only return a single tree, and even then must havoc the input tree given to it. The lack of handling of multiple structures means that even simple programs like mergesort (that merges two lists), cannot be handled, and simple programs that manipulate two lists or two trees cannot be reasoned with. Also, structures such as doubly-linked lists, trees with parent pointers, etc. are out of scope of this work.

On the other hand, in Chapter 6, we have carefully designed $\text{D}\textsc{ryad}_{\text{sep}}$, a dialect of separation logic, which seems an ideal logic with the scope for handling real-world programs handling multiple or more complex data-structures and their separation.

Therefore, the goal of this chapter is to exploit $\text{D}\textsc{ryad}_{\text{sep}}$ in verifying practical programs with natural proofs. Technically, we aim at handling user-defined structures expressible in separation logic, multiple structures and their separation, programs with while-loops, etc., because of our *logical* treatment of separation logic using classical logic.

We develop natural proofs for $\text{D}\textsc{ryad}_{\text{sep}}$ by showing a natural proof mechanism for the equivalent formulas in classical logic, utilizing in part the translation defined in Section 6.5. We exploit the two tactics for natural proofs set forth in Chapter 5: in the first step, utilize the idea of unfolding across the footprint to strengthen the verification condition; in the second step, prove the validity of the VC soundly using the technique of formula abstraction. The resulting formula falls in a logic over sets and integers, which is then

decided using the theory of uninterpreted functions and arrays using SMT solvers.

The basic proof tactic that we follow is not just dependent on the formula embodying the verification condition, but also on the precise footprint touched by the program segment being verified. The key feature is that heaplets and separation logic constructs, which get translated to recursively defined sets of locations, are unfolded along with other user-defined recursive definitions and formula-abstracted using this uniform natural proof strategy.

While our proof strategy is roughly as above, there are many technical details that are complex. For example, the heaplets defined by pre/post conditions intrinsically specify the modified locations of the heap, which have to be considered when processing procedure calls in order to ensure which recursively defined metrics on locations continue to hold after a procedure call. Also, the final decidable theories that we compile our conditions down to does require a bit of quantification, but it turns out to be in the *array property fragment* which admits automatic decision procedures.

Our proof mechanisms are essentially a class of decidable proof tactics that result in sound but incomplete validation procedures. To show that this class of natural proofs is effective in practice, we provide a prototype implementation of our technique, which handles a low-level programming language with pre-conditions and post-conditions written in $\text{DRYAD}_{\text{sep}}$. We show, using a large class of correct programs manipulating lists, trees, cyclic lists, and doubly linked lists as well as *multiple* data-structures of these kinds, that the natural proof mechanism succeeds in proving the verifications conditions automatically. These programs are drawn from a range of sources, from textbook data-structure routines (binary search trees, red-black trees, etc.) to routines from Glib low-level C-routines used in GTK+/Gnome to implement file-systems, from the Schorr-Waite garbage collection algorithm, to several programs from a recent secure framework developed for mobile applications [54]. Our work is by far the only one that we know of that can handle such a large class of programs, completely automatically. Our experience has been that the user-provided contracts and invariants are easily expressible in $\text{DRYAD}_{\text{sep}}$, and the automatic natural proof mechanisms work extremely fast. In fact, contrary to our own expectations, we also found that the tool is useful in *debugging*: in several cases, when the annotations sup-

$$
\begin{array}{lcl}
P & :- & P \,;P \ \mid\ stmt \\
stmt & :- & u := v \mid u := \texttt{nil} \mid u := v.pf \mid u.pf := v \\
 & & \mid\ j := u.df \ \mid\ u.df := j \ \mid\ j := aexpr \\
 & & \mid\ u := \texttt{new} \ \mid\ \texttt{free}\ u \ \mid\ \texttt{assume}\ bexpr \\
 & & \mid\ u := f(\vec{v}, \vec{z}) \ \mid\ j := g(\vec{v}, \vec{z}) \\
aexpr & :- & int \ \mid\ j \ \mid\ aexpr + aexpr \ \mid\ aexpr - aexpr \\
bexpr & :- & u = v \mid u = \texttt{nil} \mid aexpr \leq aexpr \\
 & & \mid\ \neg bexpr \mid bexpr \vee bexpr
\end{array}
$$

Figure 7.1: Syntax of programs

plied were incorrect, the model provided by the SMT solver for the natural proof was useful in detecting errors and correcting the invariants/program.

**Organization:** Section 7.1 defines a simple programming language and the corresponding Hoare-triple with respect to $\text{DRYAD}_{\text{sep}}$. Section 7.2 shows that the VC for the Hoare-triple can be captured by a $\text{DRYAD}_{\text{sep}}$. Section 7.3 and 7.4 formally presents the natural proofs for $\text{DRYAD}_{\text{sep}}$ in two steps: *unfolding across the footprint* and *formula abstraction*, respectively. After that, we give some intuition to the reader through a case study of verifying the `heapify` routine for max-heap in Section 7.5. Section 7.6 evaluates natural proofs for $\text{DRYAD}_{\text{sep}}$ through a wide variety of open-source programs, and Section 7.7 compares our work with other state-of-the-art techniques in verification.

## 7.1 Programs and Hoare-triples

We consider straight-line program segments that do destructive pointer-updates, data-updates and procedure calls. Parameterized by a set of pointer fields $PF$ and a set of data-fields $DF$, the syntax of the programs is defined in Figure 7.1, where $pf \in PF$, $f \in DF$, $u$ and $v$ are program variables of type location, $j$ and $z$ are program variables of type integer, $int$ is an integer constant. To simplify the presentation, we assume all program variables are local and are either pre-assigned or assigned once in the program.

We allow two kinds of recursive procedures, one returning a location $f(\vec{v}, \vec{z})$ and one returning an integer $g(\vec{v}, \vec{z})$. Each procedure/program is annotated with its pre- and post-conditions in DRYAD. The pre-condition is denoted as

a formula $\psi_{pre}(\vec{v}, \vec{z}, \vec{c})$, where $\vec{v}$ and $\vec{z}$ are variables as the formal parameters/program variables, $\vec{c}$ is a set of implicitly existentially quantified complimentary variables (e.g., variable $K$ in the pre-condition $\varphi_{pre}$ in Figure 7.2). The post-condition is denoted as a formula $\psi_{post}(ret, \vec{v}, \vec{z}, \vec{c})$, where $ret$ is the variable representing the returned value, of corresponding type, $\vec{v}$ and $\vec{z}$ are program variables, $\vec{c}$ is a set of complimentary variables that have appeared in the pre-condition $\psi_{pre}$.

Now consider a Hoare-triple, i.e., a straight-line program with its pre- and post-conditions: $\{\psi_{pre}\}\ P\ \{\psi_{post}\}$, we define its partial correctness without considering memory errors[1]: $P$ is partially correct iff for every normal execution (memory-error free) of $P$, which transits state $C$ to state $C'$, if $C \models \psi_{pre}$, then $C' \models \psi_{post}$.

Given a Hoare-triple $\{\psi_{pre}\}\ P\ \{\psi_{post}\}$ as defined above, a set of recursive definitions and a set of annotated procedure declarations are presented here. Assume that $P$ consists of $n$ statements, then consider a normal execution $\mathcal{E}$, which can be represented as a sequence of program states $(C_0, \ldots, C_n)$, where each $C_i = (R_i, s_i, h_i)$ represents the program state after executing the first $i$ statements. The verification condition is just a formula interpreted on a state sequence $(C_0, \ldots, C_n)$. Let $pf_i : Loc \to Loc$ be the function mapping every location $l$ to its $pf$ pointer, i.e., $pf_i(l) = h_i(l, pf)$ for every location $l$. Similarly, $df_i : Loc \to Int$ is defined such that $df_i(l) = h_i(l, df)$ for every $l$. Recall that every program variable is either pre-assigned or assigned once in the program, each $s_i$ is an expantion of the previous one, and $s_n$ is the store with all program variables defined. Hence we simply use $v$ to denote $s_n(v)$. Moreover, every recursive predicate/function is also indexed by $i$. For example, $p_i$ is the recursive predicate such that $p_i(l)$ is true iff $C_i \models T(p^{\Delta}(l), \text{reachset}^{p}(l))$. Now for every formula $\varphi$ and every index $i$, we can give the index $i$ to all the pointer fields, data fields and recursive definitions. We denote the indexed formula as $\varphi[i]$.

[1]We exclude memory errors in order to simplify the presentation. Memory errors can be handled using a similar VC generation for assertions that negate the conditions for memory errors to occur.

## 7.2 Generating the Verification Condition

We now algorithmically derive the verification condition $\psi_{\mathrm{VC}}$ corresponding to it in classical logic with recursive definitions on the global heap.

Assume there are $m$ procedure calls in $P$, then $P$ can be divided into $m+1$ basic segments (subprograms without procedure calls):

$$S_0 \;;\; g_1 \;;\; S_1 \;;\; \dots \;;\; g_m \;;\; S_m$$

where $S_d$ is the $d+1$-th basic segment and $g_d$ is the $d$-th procedure call.

For each $d \in [m]$, let the $d$-th procedure call in $P$ be the $t_d$-th statement (we also extend the index $d$ to $-1$, $0$ and $m+1$ such that $t_{-1} = t_0 = 0$ and $t_{m+1} = n+1$). Note that $\mathcal{E}$ requires that a portion of the state $C_{t_d-1}$ satisfies the precondition of the call, and a portion of the state $C_{t_d}$ satisfies the postcondition of the call. We denote the two required portions $C_{t_d-1} \mid \mathit{Call}_d$ and $C_{t_d} \mid \mathit{Return}_d$, respectively, where $\mathit{Call}_d \subseteq R_{t_d-1}$ and $\mathit{Return}_d \subseteq R_{t_d}$ are two sets of records.

Let all the location variables appearing in $P$ be *LVars*. We call a location variable $v$ *dereferenced* if $v$ appears on the left-hand side of a dereferencing operator "." in $P$. We call a location variable $v$ *modified* if $v$ appears in a statement of the form $v.pf := u$ or $v.df := j$ in $P$. Then we can extract the set of dereferenced variables *Deref* and the set of modified variables *Mod*. Note that a modified variable is always dereferenced, i.e., $\mathit{Mod} \subseteq \mathit{Deref}$. For each basic segment $S_d$, let the dereferenced and modified variables within the segment be $\mathit{Deref}_{t_d}$ and $\mathit{Mod}_{t_d}$, respectively.

For the $d$-th procedure call, let the pre- and post-condition associated with the procedure be $\psi^d_{pre}(\vec{v}, \vec{z}, \vec{c})$ and $\psi^d_{post}(ret, \vec{v}, \vec{z}, \vec{c})$, respectively. Since $\mathcal{E}$ is a normal execution, we have $C_{t_d-1} \models T(\psi^d_{pre}(\vec{v_d}, \vec{z_d}, \vec{c_d}), \mathit{Call}_d)$ and $C_{t_d} \models T(\psi^d_{post}(u, \vec{v_d}, \vec{z_d}, \vec{c_d}), \mathit{Return}_d)$ (assume the procedure call returns a location to $u$), where $\vec{v_d}$ and $\vec{z_d}$ are the actual parameters of the procedure call, $\vec{c_d}$ are the complimentary variables with fresh names.

Now we are ready to define the verification condition corresponding to $P$. We first derive a formula expressing that $\mathcal{E}$ does not involve null pointer dereference:

$$\textsc{NoNullDereference} \equiv \bigwedge_{v \in \mathit{Deref}} v \neq \mathit{nil}$$

For each $i \in [n]$, we show the effect as per statement on the verification condition generated as follows.

$[\ u := v\ ]$

$$\varphi_i \equiv u = v \wedge R_i = R_{i-1} \wedge \mathit{FieldsUnmod}(PF \cup DF, i, i-1)$$

$[\ u := \texttt{nil}\ ]$

$$\varphi_i \equiv u = nil \wedge R_i = R_{i-1} \wedge \mathit{FieldsUnmod}(PF \cup DF, i, i-1)$$

$[\ u := v.pf\ ]$

$$\begin{aligned} \varphi_i \equiv\ & v \in R_{i-1} \wedge u = pf_{i-1}(v) \wedge R_i = R_{i-1} \\ & \wedge \mathit{FieldsUnmod}(PF \cup DF, i, i-1) \end{aligned}$$

$[\ u.pf := v\ ]$

$$\begin{aligned} \varphi_i \equiv\ & u \in R_{i-1} \wedge pf_i = pf_{i-1}\{v \leftarrow u\} \wedge R_i = R_{i-1} \\ & \wedge \mathit{FieldsUnmod}(PF \cup (DF \setminus \{pf\}), i, i-1) \end{aligned}$$

$[\ j := u.df\ ]$

$$\begin{aligned} \varphi_i \equiv\ & u \in R_{i-1} \wedge j = df_{i-1}(u) \wedge R_i = R_{i-1} \\ & \wedge \mathit{FieldsUnmod}(PF \cup DF, i, i-1) \end{aligned}$$

$[\ u.df := j\ ]$

$$\begin{aligned} \varphi_i \equiv\ & u \in R_{i-1} \wedge df_i = df_{i-1}\{j \leftarrow u\} \wedge R_i = R_{i-1} \\ & \wedge \mathit{FieldsUnmod}((PF \setminus \{df\}) \cup DF, i, i-1) \end{aligned}$$

$[\ j := aexpr\ ]$

$$\varphi_i \equiv j = aexpr \wedge R_i = R_{i-1} \wedge \mathit{FieldsUnmod}(PF \cup DF, i, i-1)$$

[ $u := \texttt{new}$ ]

$$\varphi_i \;\equiv\; new_i \neq nil \wedge u = new_i \wedge new_i \notin R_{i-1} \wedge R_i = R_{i-1} \cup \{new_i\}$$
$$\wedge \bigwedge_{pf} \big(pf_i = pf_{i-1}\{nil \leftarrow new_i\}\big) \wedge \bigwedge_{df} \big(df_i = df_{i-1}\{0 \leftarrow new_i\}\big)$$

[ $\texttt{free}\ u$ ]

$$\varphi_i \;\equiv\; u \in R_{i-1} \wedge R_i = R_{i-1} \setminus \{u\} \wedge FieldsUnmod(PF \cup DF, i, i-1)$$

[ $\texttt{assume}\ bexpr$ ]

$$\varphi_i \;\equiv\; bexpr \wedge R_i = R_{i-1} \wedge FieldsUnmod(PF \cup DF, i, i-1)$$

[ $u := f(\vec{v}, \vec{z})$ ]

$$\varphi_i \;\equiv\; T\big(\psi^d_{pre}(\vec{v}, \vec{z}, \vec{c_d}), Call_d\big)[i-1] \wedge T\big(\psi^d_{post}(u, \vec{v}, \vec{z}, \vec{c_d}), Return_d\big)[i]$$
$$\wedge\ (R_{i-1} \setminus Call_d) \cap Return_d = \emptyset \wedge R_i = (R_{i-1} \setminus Call_d) \cup Return_d$$
$$\text{where } d \text{ is the index such that } t_d = i$$

[ $j := g(\vec{v}, \vec{z})$ ]

$$\varphi_i \;\equiv\; T\big(\psi^d_{pre}(\vec{v}, \vec{z}, \vec{c_d}), Call_d\big)[i-1] \wedge T\big(\psi^d_{post}(j, \vec{v}, \vec{z}, \vec{c_d}), Return_d\big)[i]$$
$$\wedge\ (R_{i-1} \setminus Call_d) \cap Return_d = \emptyset \wedge R_i = (R_{i-1} \setminus Call_d) \cup Return_d$$
$$\text{where } d \text{ is the index such that } t_d = i$$

where $FieldsUnmod(F, i, j)$ is short for $\bigwedge_{field \in F}(field_i = field_j)$.

As shown above, each statement's strongest post condition is captured in the logic, and for procedure calls, the heaplet manipulated by the procedure is carefully taken into account to update the heap at the caller. The conjunction of these formulas captures the modification made in $\mathcal{E}$:

$$\textsc{Modification} \;\equiv\; \bigwedge_{i \in [n]} \varphi_i$$

Finally, we can define two formulas to capture the pre- and post-conditions:

$$\textsc{Pre} \equiv T(\psi_{pre}, R_0)[0]$$

$$\textsc{Post} \equiv T(\psi_{post}, R_n)[n]$$

Now the validity of $\{\psi_{pre}\}\ P\ \{\psi_{post}\}$ can be captured by the following formula:

$$\psi_{\mathrm{VC}} \equiv \big(\textsc{Pre} \wedge \textsc{NoNullDereference} \wedge \textsc{Modification}\big) \rightarrow \textsc{Post}$$

**Theorem 7.2.1.** *Given a Hoare-triple $\{\psi_{pre}\}\ P\ \{\psi_{post}\}$, assume that each procedure call in $P$ satisfies its associated pre- and post-conditions. Then the triple is valid if the formula $\psi_{VC}$ derived above is valid. Moreover, when $P$ contains no procedure calls, the triple is valid iff $\psi_{VC}$ is valid.*

*Proof.* We prove the soundness by contradiction. Assume the Hoare-triple $\{\psi_{pre}\}\ P\ \{\psi_{post}\}$ is not valid. Assume $P$ consists of $n$ statements, then there is an execution $\mathcal{E}$, which can be represented as a state sequence $(C_0, \ldots, C_n)$ where each $C_i = (R_i, s_i, h_i)$, such that $(C_0, R_0)$ satisfies $\psi_{pre}[0]$, $(C_n, R_n)$ satisfies $\psi_{post}[n]$, and the whole execution is memory error free. Then by the definitions of $\textsc{Pre}$, $\textsc{Post}$ and $\textsc{NoNullDereference}$, and the definition of $\psi_{\mathrm{VC}}$, $\mathcal{E} \models \textsc{Pre} \wedge \textsc{NoNullDereference} \wedge \textsc{Post}$, it suffices to show that $\mathcal{E} \models \textsc{Modification}$, in which case $\mathcal{E}$ dissatisfies $\psi_{\mathrm{VC}}$. The contradiction will conclude the proof.

Since $\textsc{Modification} \equiv \bigwedge_{i\in[n]} \varphi_i$, we just need to prove $\mathcal{E} \models \varphi_\rangle$ for each $i \in [n]$, by case analysis on the type of the $i$-th statement in $P$:

[ $u := v$ ]

$$\varphi_i \equiv u = v \wedge R_i = R_{i-1} \wedge \mathit{FieldsUnmod}(PF \cup DF, i, i-1)$$

> The variable assignment makes $u$ points to where $v$ points to. Hence $u = v$. Since the heap is unmodified from $C_{i-1}$ to $C_i$, the heap domain remains the same ($R_i = R_{i-1}$), and all the field functions remain the same ($\mathit{FieldsUnmod}(PF \cup DF, i, i-1)$).

[ $u := \texttt{nil}$ ]

$$\varphi_i \equiv u = nil \wedge R_i = R_{i-1} \wedge \mathit{FieldsUnmod}(PF \cup DF, i, i-1)$$

The variable assignment makes $u$ points to *nil*, so $u = nil$. Similar to the above case, the heap is also unmodified from $C_{i-1}$ to $C_i$.

[ $u := v.pf$ ]

$$\begin{aligned}\varphi_i \equiv\ & v \in R_{i-1} \wedge u = \mathit{pf}_{i-1}(v) \wedge R_i = R_{i-1} \\ & \wedge \mathit{FieldsUnmod}(PF \cup DF, i, i-1)\end{aligned}$$

The dereferencing on $v$ implies that $v$ points to a valid location at timestamp $i-1$, i.e., $v \in R_{i-1}$. Moreover, the assignment makes $u$ points to the *pf* field of $v$ at timestamp $i-1$, formally $u = \mathit{pf}_{i-1}(v)$. Similar to the above case, the heap is also unmodified from $C_{i-1}$ to $C_i$.

[ $u.pf := v$ ]

$$\begin{aligned}\varphi_i \equiv\ & u \in R_{i-1} \wedge \mathit{pf}_i = \mathit{pf}_{i-1}\{v \leftarrow u\} \wedge R_i = R_{i-1} \\ & \wedge \mathit{FieldsUnmod}(PF \cup (DF \setminus \{\mathit{pf}\}), i, i-1)\end{aligned}$$

Similar to the above case, $u$ points to a valid location at timestamp $i-1$ ($u \in R_{i-1}$). the mutation makes the *pf* field at timestamp $i$ updated from that at timestamp $i-1$: $\mathit{pf}_i = \mathit{pf}_{i-1}\{v \leftarrow u\}$. Moreover, the heap domain is unmodified, so $R_i = R_{i-1}$. The other field functions also remain the same, which is captured by $\mathit{FieldsUnmod}(PF \cup (DF \setminus \{\mathit{pf}\}), i, i-1)$.

[ $j := u.df$ ]

$$\begin{aligned}\varphi_i \equiv\ & u \in R_{i-1} \wedge j = \mathit{df}_{i-1}(u) \wedge R_i = R_{i-1} \\ & \wedge \mathit{FieldsUnmod}(PF \cup DF, i, i-1)\end{aligned}$$

Similar to the $u := v.pf$ case.

[ $u.df := j$ ]

$$\varphi_i \;\equiv\; u \in R_{i-1} \wedge df_i = df_{i-1}\{j \leftarrow u\} \wedge R_i = R_{i-1} \\ \wedge\ FieldsUnmod((PF \setminus \{df\}) \cup DF, i, i-1)$$

Similar to the $u.pf := v$ case.

[ $j := aexpr$ ]

$$\varphi_i \;\equiv\; j = aexpr \wedge R_i = R_{i-1} \wedge FieldsUnmod(PF \cup DF, i, i-1)$$

The statement assigns the value of *aexpr*, which is expressible in our logic, to $j$. Hence $j = aexpr$. The rest is similar to other variable assignment cases.

[ $u :=$ `new` ]

$$\varphi_i \;\equiv\; new_i \neq nil \wedge u = new_i \wedge new_i \notin R_{i-1} \wedge R_i = R_{i-1} \cup \{new_i\} \\ \wedge \bigwedge_{pf} \big(pf_i = pf_{i-1}\{nil \leftarrow new_i\}\big) \wedge \bigwedge_{df} \big(df_i = df_{i-1}\{0 \leftarrow new_i\}\big)$$

This statement makes $u$ points to a freshly allocated location, namely $new_i$ in $\mathcal{E}$. So it is clear that $new_i \neq nil \wedge u = new_i$. Since the heap domain at timestamp $i$ is an extension of that at timestamp $i-1$ by adding $new_i$, we know that $new_i \notin R_{i-1} \wedge R_i = R_{i-1} \cup \{new_i\}$. By default, for $new_i$, each pointer field initially points to *nil*, each data field initially stores 0. The remaining portion of the heap is exactly the same as $C_{i-1}$. Hence

$$\bigwedge_{pf} \big(pf_i = pf_{i-1}\{nil \leftarrow new_i\}\big) \wedge \bigwedge_{df} \big(df_i = df_{i-1}\{0 \leftarrow new_i\}\big)$$

[ `free` $u$ ]

$$\varphi_i \;\equiv\; u \in R_{i-1} \wedge R_i = R_{i-1} \setminus \{u\} \wedge FieldsUnmod(PF \cup DF, i, i-1)$$

This statement removes the location pointed by $u$ from the heap. So the old heap contains this location, and the new heap can be obtained

by subtracting it from the old heap: $u \in R_{i-1} \wedge R_i = R_{i-1} \setminus \{u\}$. Since the domain is shrinked, the field function can be simply unchanged.

[ `assume` *bexpr* ]

$$\varphi_i \ \equiv\ \mathit{bexpr} \wedge R_i = R_{i-1} \wedge \mathit{FieldsUnmod}(PF \cup DF, i, i-1)$$

The assumed condition *bexpr*, which can be expressed in our logic, must be true. The heap is simply unmodified.

[ $u := f(\vec{v}, \vec{z})$ ]

$$\begin{aligned}\varphi_i \ \equiv\ & T\big(\psi^d_{pre}(\vec{v}, \vec{z}, \vec{c_d}), \mathit{Call}_d\big)[i-1] \wedge T\big(\psi^d_{post}(u, \vec{v}, \vec{z}, \vec{c_d}), \mathit{Return}_d\big)[i] \\ & \wedge (R_{i-1} \setminus \mathit{Call}_d) \cap \mathit{Return}_d = \emptyset \wedge R_i = (R_{i-1} \setminus \mathit{Call}_d) \cup \mathit{Return}_d \\ & \qquad\qquad \text{where } d \text{ is the index such that } t_d = i\end{aligned}$$

As assumed, this call is the $d$-th procedure call in $P$, and $C_{i-1}$ satisfies the associated precondition by the heaplet defined by $\mathit{Call}_d$, and $C_i$ satisfies the associated postcondition by the heaplet defined by $\mathit{Return}_d$. Then formally we have

$$T\big(\psi^d_{pre}(\vec{v}, \vec{z}, \vec{c_d}), \mathit{Call}_d\big)[i-1] \wedge T\big(\psi^d_{post}(u, \vec{v}, \vec{z}, \vec{c_d}), \mathit{Return}_d\big)[i]$$

being satisfied by $\mathcal{E}$.

Due to the framing property of the separation semantics, the portion of $C_{i-1}$ that is not required by $\psi_{pre}$ remains unchanged, and is disjoint from $\mathit{Return}_d$ (since the returned location is assigned to $u$, the variable *ret* can be replaced with $u$). This property can be expressed as

$$(R_{i-1} \setminus \mathit{Call}_d) \cap \mathit{Return}_d = \emptyset \wedge R_i = (R_{i-1} \setminus \mathit{Call}_d) \cup \mathit{Return}_d$$

[ $j := g(\vec{v}, \vec{z})$ ]

$\varphi_i$ is defined in the same way as the above case,
except replacing $u$ with $j$.

The proof is also similar to the case of [[ $u := f(\vec{v}, \vec{z})$]].

□

## 7.3 Unfolding Across the Footprint

The verification condition obtained above is a quantifier-free formula involving recursive definitions and the reachable sets of the form $reach^p(x)$, which are also defined recursively. While these recursive definitions can be unfolded *ad infinitum*, we exploit a proof tactic called *unfolding across the footprint*. Intuitively, the footprint is the set of locations explored by the program *explicitly* (not including procedure calls). More precisely, a location is in the footprint if it is dereferenced explicitly in the program. The idea is to unfold the recursive definitions over the footprint of the program, so that recursive definitions on the footprint nodes are related, as precisely as possible, to the recursive definitions on frontier nodes. This will enable effective use of the formula abstraction mechanism, as when recursive definitions on frontier nodes are made uninterpreted, the unfolding formulas ensure tight conditions that the frontier nodes have to satisfy.

Furthermore, to enable effective frame reasoning, it is also necessary to strengthen the verification condition with a set of instances of the frame rule. More concretely, we need to capture the fact that a recursive definition (or a field) on a location is unchanged during a segment or procedure call of the program, if the reachable locations (or only the location itself) are not affected by the segment or procedure call.

We incorporate the above facts formally into the verification condition. Let us introduce a macro function *fp* that identifies the location variables that are in (or aliased to something in) the footprint. The footprint of $P$, $FP$, is the set of dereferenced variables in $P$ (we call a location variable dereferenced if it appears on the left-hand side of a dereferencing operator "." in $P$). Then $fp(u) \equiv \bigvee_{v \in FP}(u = v)$.

Now we state the unfoldings and framings using a formula; we denote it as UnfoldAndFrame. Assume there are $m$ procedure calls in $P$, then $P$ can be divided into $m+1$ basic segments (subprograms without procedure calls): $S_0\ ;\ g_1\ ;\ S_1\ ;\ \ldots\ ;\ g_m\ ;\ S_m$ where $S_d$ is the $(d+1)$-th basic segment and $g_d$ is the $d$-th procedure call. Then

$$\textsc{UnfoldAndFrame} \equiv \bigwedge_{rec} \bigwedge_{0 \le d \le m} \bigwedge_{u \in LVars \cup \{nil\}} \Bigg[ \Bigg( \Big( fp(u) \vee u = nil \Big) \Rightarrow \Big( \textsc{Unfold}^{rec}_{d}(u) \wedge \textsc{FieldUnchanged}_{d}(u) \Big) \Bigg) \wedge \Bigg( \Big( \neg fp(u) \vee u = nil \Big) \Rightarrow \textsc{RecUnchanged}^{rec}_{d}(u) \Bigg) \Bigg]$$

The formula enumerates every recursive definition *rec* and every index $d$, and for each location $u$ that is either pointed to by a location variable or is *nil*, the formula checks if $u$ is in the footprint, and then unfolds it or frames it accordingly. If $u$ is in the footprint, then we unfold *rec* for the timestamps before and after $S_d$ (represented by the formula $\textsc{Unfold}^{rec}_{d}(u)$ ); moreover, all fields of $u$ are unchanged if it is not affected during calling $g_d$ (represented by the formula $\textsc{FieldUnchanged}_{d}(u)$ ). If $u$ is not in the footprint, i.e., in the frontier, then *rec* and its corresponding reach set $reach^{rec}$ are unchanged after executing $S_d$, if $S_d$ does not modify any location in $reach^{rec}$; they are also unchanged if $reach^{rec}$ is not affected by calling $g_d$. These frame assertions are represented by the formula $\textsc{RecUnchanged}^{rec}_{d}(u)$.

Now we formulate the subformulas mentioned above. Let $u$ be a location variable in $LVars \cup \{nil\}$ and let $i$ be an timestamp such that $1 \le i \le n$. For each recursive definition $rec^{\Delta}$ whose $\Delta$-eliminated version defined as $rec(x) \stackrel{def}{=} \mathrm{def}^{rec}(x, \vec{t}, \vec{v})$ and whose reach set defined as $reach^{rec}(x) \stackrel{def}{=} \mathrm{reachdef}^{\,rec}(x)$, we can derive a formula $\textsc{Unfold}^{rec}(i, u)$ for unfolding both $rec^{\Delta}$ and its corresponding reach set on $u$ at timestamp $i$, provided that $u$ is allocated at the current timestamp ($u \in R_i$). Note that in $def^{\,rec}(x, \vec{t}, \vec{v})$, $x$ will be renamed as $u$, and $\vec{t}$ will not be renamed as they are program variables, but $\vec{v}$ are existentially quantified and should be replaced with fresh variable names. Due to the restrictions on the recursive definitions, every $v$ is unique and can be determined by dereferencing $u$ on the corresponding pointer fields, say $pf^{\,rec,v}$. Hence we can replace each $v$ in $\vec{v}$ distinctly as $u_rec_v_i$. Let the renamed formula be $\mathrm{def}^{rec}(u, \vec{t}, \vec{v}_{fresh})$, then we can derive

$$\textsc{UnfoldAt}^{rec}(i,u) \;\equiv\; \Big( \; reach_i^{rec}(u) = \mathrm{reachdef}_i^{\;rec}(u) \; \Big) \;\wedge\; \Bigg( u \in R_i \rightarrow$$

$$\bigg( \Big( rec_i(u) \leftrightarrow \mathrm{def}_i^{rec}(u, \vec{t}, \vec{v}_{fresh}) \Big) \wedge \bigwedge_{v \in \vec{v}} \Big( pf_i^{\;rec,v}(u) = u_rec_v_i \Big) \; \bigg) \Bigg)$$

Now the footprint unfolding is just unfolding $u$ at the beginning and end of each program segment (for the $d$-th segment, the timestamp $t_d$ and $t_{d+1}-1$, respectively):

$$\textsc{Unfold}_d^{rec}(u) \;\equiv\; \textsc{UnfoldAt}^{rec}(t_d, u) \;\wedge\; \textsc{UnfoldAt}^{rec}(t_{d+1}-1, u)$$

In $\textsc{UnfoldAndFrame}$, we simply make the conjunction of all these unfoldings, for each dereferenced variable, for every timestamp at the beginning and the end and the program, as well as before/after each procedure call. As a special location, *nil* is also unfolded with respect to each recursive definition and each timestamp.

The formula $\textsc{FieldUnchanged}_d(u)$ describes that, in the $d$-th procedure call, if the location $u$ is not *nil*, then for each field *pf* (or *df*), $pf_{t_d-1}(u)$ and $pf_{t_d}(u)$ are the same if $u$ itself is not affected during the call:

$$\textsc{FieldUnchanged}_d(u) \;\equiv\; \bigg( \; \big(u \neq nil \wedge u \notin Call_d\big) \rightarrow$$

$$\Big( \bigwedge_{pf} \big(pf_{t_d-1}(u) = pf_{t_d}(u)\big) \wedge \bigwedge_{df} \big(df_{t_d-1}(u) = df_{t_d}(u)\big)\Big) \; \bigg)$$

Finally, to define $\textsc{RecUnchanged}_d^{rec}(u)$, we first define a formula expressing that a recursive definition and its corresponding reach set on a location are unchanged between two timestamps:

$$\textsc{UnchangedBetween}^{rec}(u, i, i') \;\equiv\; rec_i(u) = rec_{i'}(u) \wedge reach_i^{rec}(u) = reach_{i'}^{rec}(u)$$

For each non-footprint location variable $u$ and for each recursive predicate $rec^{\Delta}$, the formula $\textsc{RecUnchanged}_d$ just captures the fact that $rec(u)$ and $reach^{rec}(u)$ are unchanged in two cases:

1. in the $d$-th segment of the program (between timestamp $t_d$ and $t_{d+1}-1$), they are unchanged if reach set is not modified; or

2. in the $d$-th procedure call (between the timestamp $t_d - 1$ and $t_d$) if the reach set is not affected during the call.

Moreover, it also incorporates the fact that the reach set on $u$ contains $u$ itself. Formally,

$$\begin{aligned}
&\textsc{RecUnchanged}_d^{rec}(u) \;\equiv \\
&\quad \Big( \; (reach_{t_d}^{rec}(u) \cap Mod_{t_d} = \emptyset) \rightarrow \textsc{UnchangedBetween}^{rec}(u, t_d, t_{d+1} - 1) \; \Big) \\
&\wedge \quad \Big( \; (reach_{t_d}^{rec}(u) \cap Call_d = \emptyset) \rightarrow \textsc{UnchangedBetween}^{rec}(u, t_d - 1, t_d) \; \Big) \\
&\wedge \quad \Big( u \neq nil \rightarrow ( \; reach_{t_d}^{rec}(u) \cap reach_{t_{d+1}-1}^{rec}(u) \; ) \Big)
\end{aligned}$$

For each variable $u$ that is not in the footprint and for the special location $\{nil\}$, we apply $\textsc{RecUnchanged}_d^{rec}(u)$.

Now we can strengthen the verification condition by incorporating the derived formula above:

$$\psi'_{\mathrm{VC}} \;\equiv\; \psi_{\mathrm{VC}} \wedge \textsc{UnfoldAndFrame}$$

Since the incorporated formula is implied by the verification condition, we can reduce the validity of $\psi_{\mathrm{VC}}$ to the validity of $\psi'_{\mathrm{VC}}$.

**Theorem 7.3.1.** *Given a Hoare-triple $\{\psi_{pre}\}$ $P$ $\{\psi_{post}\}$, its verification condition $\psi_{VC}$ is valid if and only if $\psi'_{VC}$ is valid.*

*Proof.* Notice that $\psi'_{\mathrm{VC}}$ strengthens $\psi_{\mathrm{VC}}$ with a set of conjuncts represented as $\textsc{UnfoldAndFrame}$, it suffices to show that $\textsc{UnfoldAndFrame}$ is a tautology for arbitrary execution of the program. In other word, for every recursive definition $rec$, for every timestamp $d$, and for every location $u$, we need to show:

- $\Big(fp(u) \vee u = nil\Big) \Rightarrow \Big(\textsc{Unfold}_d^{rec}(u) \wedge \textsc{FieldUnchanged}_d(u)\Big)$

- $\Big( \; \neg fp(u) \vee u = nil\Big) \Rightarrow \textsc{RecUnchanged}_d^{rec}(u)$

For the first assertion, if $u$ is dereferenced or is *nil*, then the unfolding formula $\textsc{Unfold}_d^{rec}(u)$ is true as every $rec_d$ and its corresponding $\text{reachset}_d^{rec}$ are fixpoints; and the formula $\textsc{FieldUnchanged}_d(u)$ is true as no field will be modified if $u$ is not affected during the call.

For the second assertion, if $u$ is not dereferenced (including the *nil* case), then the formula $\textsc{RecUnchanged}_d^{rec}(u)$ is true, as it just enumerates the possible cases that $rec_d$ and $\text{reachset}_d^{rec}$ are unchanged. ☐

## 7.4 Formula Abstraction

While checking the validity of the strengthened verification condition $\psi'_{\text{VC}}$ is still undecidable, as we argued before, it is often sufficient to prove it by assuming that the recursive definitions are arbitrary, or uninterpreted. Moreover, the uninterpreted formula falls in the array property fragment [19], whose satisfiability is decidable and is supported by modern SMT solvers such as Z3 [28]. This tactic roughly corresponds to applying *unification* in proof systems.

To prove $\psi'_{\text{VC}}$, we first replace each recursive predicate $rec_d$ with an uninterpreted predicate $\hat{rec}_d$, and replacing the corresponding reach-set function $reach_d^{rec}$ with an uninterpreted function $\hat{reach}_d^{rec}$. Let the result formula be $\psi_{\text{VC}}^{\text{abs}}$. This conversion, called *formula abstraction*, is sound:

**Proposition 7.4.1.** *if $\psi_{VC}^{abs}$ is valid, so is $\psi'_{VC}$.*

*Proof.* We prove by contradiction. Assume that $\psi_{\text{VC}}^{\text{abs}}$ is valid but $\psi'_{\text{VC}}$ is not, then there is a model $\mathcal{M}$ satisfying $\psi'_{\text{VC}}$. We can construct a similar model $\mathcal{M}'$ that consists of the same elements from $\mathcal{M}$. For each recursive definition $rec_d$, $\mathcal{M}'$ interprets the uninterpreted $\hat{rec}_d$ and $\hat{reach}_d^{rec}$ just using the interpretation of $rec_d$ and $reach_d^{rec}$ in $\mathcal{M}$. Then $\mathcal{M}'$ satisfies $\psi_{\text{VC}}^{\text{abs}}$ and contradicts the assumption. ☐

When a proof for $\psi_{\text{VC}}^{\text{abs}}$ is found, we call it a natural proof for $\psi'_{\text{VC}}$ (and also for $\psi_{\text{VC}}$).

**Remark:** Obviously the natural proof scheme is incomplete since the formula abstraction throws the semantics of the recursive definitions away and

treats them as arbitrary. When the natural proof fails, there exists some spurious counterexample that satisfies $\psi'_{\mathrm{VC}}$ but not $\psi^{\mathrm{abs}}_{\mathrm{VC}}$, as the counterexample interprets some recursive definition incorrectly. The incorrect interpretation may fall into one of the two following cases.

In the first case, the interpretation is not a fixed point of the propagation function with respect to the recursive definitions. In this case, the interpretation must be incorrect on some non-footprint locations, as we have precisely unfolded the recursive definitions on the footprint. The impreciseness on non-footprint location is reasonably expected, and is not guaranteed in the natural proof scheme.

The second case is more subtle, which is when the interpretation is a non-least fixed point. In this case, the interpretation may still be incorrect even if all locations are in the footprint. While we don't have any specific mechanism to avoid this situation, in practice, the user may exclude this case by giving recursive definitions such that the fixed point is unique. First, it is noteworthy that we can define a recursive set-of-locations as an overapproximation of the reach set. Moreover, using this recursive function we can precisely describe whether a location is within a cycle or not. Secondly, typical recursive definitions recursively reduce the definition on a location to that of its neighbors, till *nil* is reached. For these definitions, we can modify them so that it is defined recursively on a location $l$ only if $l$ is not in a cycle (which is expressible in DRYAD), otherwise a default value is given. It is not hard to prove inductively that the fixed point is unique for the modified definitions.

### 7.4.1 Deciding the Abstracted Formula

The formula abstraction step is the only step that introduces incompleteness in our framework, but helps us transform the verification condition to a decidable theory. Formula abstraction (combined with unfolding recursive definitions across the footprint) discovers *recursive proofs* where the recursion is structural recursion on the definitions of the data-structures. The use of these tactics comes from the observation that such programs often have such recursive proofs (see [72] also for use of formula abstractions).

Our goal now is to check the satisfiability of $\neg\psi^{\mathrm{abs}}_{\mathrm{VC}}$ in a decidable theory. Note that $\neg\psi^{\mathrm{abs}}_{\mathrm{VC}}$ is mostly expressible in the quantifier-free theory of arrays,

maps, uninterpreted functions, and integers: *Loc* can be viewed as an uninterpreted sort; each pointer field *pf* can be viewed as an array with both indices and elements of sort *Loc*; each data field *df* can be viewed as an array with indices of sort *Loc* and elements of sort *Int*; each integer set (or multiset) variable $S$ can be viewed as an array with indices of sort *Int* and elements of sort *Bool* (or *Int*). Moreover, each array update operation of the form $array\{elem \leftarrow key\}$ can be viewed as a read-over-write operation in the array property fragment, and each set-operation (union, intersection, etc.) can be viewed as a mapping function applying a Boolean operation ($\wedge$, $\vee$, etc.) to the range of arrays.

The only construct in $\neg\psi_{\text{VC}}^{\text{abs}}$ that escapes the quantifier-free formulation is the $\leq$ relation between integer sets/multisets; but this can be encoded using the *Array Property Fragment* (APF), which is decidable [19]. We explain the encoding as follows.

For each atomic formula of the form $S_1 < S_2$, if $S_1$ and $S_2$ are sets of integers, we can be replace the formula with a universally quantified formula as follows:

$$\forall i_1, i_2.\ \big(\ i_1 \leq i_2 \rightarrow (\neg S_2[i_1] \vee \neg S_1[i_2])\ \big)$$

Similarly, if $S_1$ and $S_2$ are integer multisets, we can replace the formula with

$$\forall i_1, i_2.\ \big(\ i_1 \leq i_2 \rightarrow (S_2[i_1] = 0 \vee S_1[i_2] = 0)\ \big)$$

The formula $S_1 \leq S_2$ where $S_1$ and $S_2$ are sets of integers can also be translated to

$$\forall i.\ \big(\ (S_1[i] \rightarrow i \leq k) \wedge (S_2[i] \rightarrow k \leq i)\ \big)$$

where $k$ is an additional existential integer variable, serving as the pivot for splitting $S_1$ and $S_2$. Similarly, when $S_1$ and $S_2$ are integer multisets, the formula is translated to

$$\forall i.\ \big(\ (S_1[i] > 0 \rightarrow i \leq k) \wedge (S_2[i] > 0 \rightarrow k \leq i)\ \big)$$

Moreover, the negation of the above relations between sets/multisets can always be expressed using two existential integer variables $k_1, k_2$ that witness

the violation of the inequality. For instance, $S_1 \not\leq S_2$ can be expressed as $k_1 \in S_1 \wedge k_2 \in S_2 \wedge k_2 \leq k_1$.

We hence obtain a formula $\psi^{APF}$ in the array property fragment combined with the theory of uninterpreted functions, maps, and arithmetic.

**Theorem 7.4.2.** *Given a Hoare-triple $\{\psi_{pre}\}\ P\ \{\psi_{post}\}$, if the derived array formula $\psi^{APF}$ is unsatisfiable, then the Hoare-triple is valid.*

*Proof.* By Theorem 7.3.1, the Hoare-triple is valid iff $\psi'_{\mathrm{VC}}$ is valid, we can simply show the validity of $\psi'_{\mathrm{VC}}$. Moreover, by Lemma 7.4.1, the formula abstraction is obviously sound, i.e., if $\psi^{\mathrm{abs}}_{\mathrm{VC}}$ is valid, so is $\psi'_{\mathrm{VC}}$. Hence, it suffices to show that $\psi^{\mathrm{abs}}_{\mathrm{VC}}$ is valid, or its negation is unsatisfiable.

Now the only difference between $\neg\psi^{\mathrm{abs}}_{\mathrm{VC}}$ and $\psi^{APF}$ is the encoding of *leq* between integer sets/multisets into APF, which is precise. The reader can easily verify that $\neg\psi^{\mathrm{abs}}_{\mathrm{VC}}$ and $\psi^{APF}$ are equivalent and conclude the proof. □

### 7.4.2 User-Provided Axioms

While natural proofs are often effective in finding recursive proofs that unfold recursive definitions and do unification, they are not geared towards finding *relationships* between various recursive definitions themselves. We hence require the user to provide certain obvious relationships between the different recursive definitions as axioms. For example, $lseg(x, y) * list(y) \Rightarrow list(x)$ is such an axiom saying that a list segment concatenated with a list yields a list. Note that these axioms are *not* program-dependent, and hence are not program-specific tactics that the user provides. These axioms are necessary typically to relate partial data-structure properties (like list segments) to complete ones (like lists), especially in iterative programs (as opposed to recursive ones), and we can fix them for each class of data-structures. We also allow the use of the separating implication, $-*$, from separation logic while specifying these axioms. User-defined axioms are instantiated, using the natural proof philosophy, on precisely the footprint nodes uniformly, and get translated to quantifier-free formulas.

```
void heapify(loc x) {
  if (x.left = nil)
    s := x.right;
  else if (x.right = nil)
    s := x.left;
  else {
    lx := x.left
    rx := x.right;
    if (lx.key < rx.key)
      s := x.right;
    else
      s := x.left;
  }
  if (s =/= nil)
    if (s.key > x.key) {
      t := s.key;
      s.key := x.key;
      x.key := t;
      heapify(s);
    }
}
```

$$\varphi_{pre} \equiv \Big(x \overset{key,left,right}{\longmapsto} (k,\ l,\ r) * mheap^{\Delta}_{\vec{pf}}(l) * mheap^{\Delta}_{\vec{pf}}(r)\Big) \wedge keys^{\Delta}_{\vec{pf}}(x) = K$$

$$\varphi_{post} \equiv mheap^{\Delta}_{\vec{pf}}(x) \wedge keys^{\Delta}_{\vec{pf}}(x) = K$$

`assume` $x.left_0 \neq nil$
`assume` $x.right_0 \neq nil$
$lx := x.left_0$
$rx := x.right_0$
`assume` $lx.key_0 < rx.key_0$
$s := x.right_0$
`assume` $s \neq nil$
`assume` $s.key_0 > x.key_0$
$t := s.key_0$
$s.key_1 := x.key_0$
$x.key_2 := t$
`heapify`$(s)$

$$\boldsymbol{mheap}^{\Delta}_{\vec{pf}}(x) \stackrel{def}{=} \Big( x = \texttt{nil} \wedge \texttt{emp} \vee \big(x \overset{key,left,right}{\longmapsto} (k,\ l,\ r) * (mheap^{\Delta}_{\vec{pf}}(l) \wedge \{k\} \geq keys^{\Delta}_{\vec{pf}}(l)) * (mheap^{\Delta}_{\vec{pf}}(r) \wedge \{k\} \geq keys^{\Delta}_{\vec{pf}}(r))\big)\Big)$$

$$\boldsymbol{keys}^{\Delta}_{\vec{pf}}(x) \stackrel{def}{=} \Big( x = \texttt{nil} \wedge \texttt{emp} : \emptyset \ ; \ x \overset{key,left,right}{\longmapsto} (k,\ l,\ r) * \texttt{true} : keys^{\Delta}_{\vec{pf}}(l) \cup \{k\} \cup keys^{\Delta}_{\vec{pf}}(r) \ ; \ \text{default} : \emptyset \Big)$$

Figure 7.2: Case study: Heapify

## 7.5 Case Study

In this section we give intuition into our verification approach through a case study. We consider the max-heap data-structure. Recall that a max-heap is a binary tree such that for each node $n$ the key stored at $n$ is greater than or equal to the keys stored at each of its children. We have defined max-heap in DRYAD$_{sep}$ in Section 6.4. In Figure 7.2, in the lower right corner, we give the recursive definitions for $keys^{\Delta}_{\vec{pf}}(x)$ and $mheap^{\Delta}_{\vec{pf}}(x)$ again.

The method `heapify` in Figure 7.2 is at the heart of the procedure for deleting the root from a max-heap (removing the node with the maximum priority). If the max-heap property is violated at a node $x$ while satisfied by

its descendants, then `heapify` restores the max-heap property at $x$. It does so by recursively descending into the tree, swapping the key of the root with the key at its left or right child, whichever is greater.

The figure also presents the pre and post conditions of the method, $\varphi_{pre}$ and $\varphi_{post}$ respectively. The precondition $\varphi_{pre}$ binds the free variable $K$ to the set of keys of $x$. The postcondition states that after the procedure call, $x$ satisfies the max-heap property and the set of keys of $x$ is unchanged (same as $K$).

### 7.5.1 Translation

One of the main aspects of our approach is to reduce reasoning about heaplet semantics and separation logic constructs to reasoning about sets of locations. We use set operations like union, intersection and membership to describe separation constraints on a heaplet satisfying a formula. This translation from $\text{DRYAD}_{\text{sep}}$ formulas, like those in Figure 7.2, to formulas in classical logic with recursive predicates and functions has been formally presented in Chapter 6. Intuitively, we associate a set of locations to each (spatial) atomic formula, which is the domain of the heaplet satisfying that formula. $\text{DRYAD}_{\text{sep}}$ requires that this heaplet is *syntactically* determined for each formula. In this example, the heaplet associated to the formula $x \mapsto \ldots$ is the singleton $\{x\}$; for recursive definitions like $\mathit{mheap}_{\vec{pf}}^{\Delta}(x)$ and $\mathit{keys}_{\vec{pf}}^{\Delta}(x)$, the domain of the heaplet is $\mathit{reach}_{\{\mathit{left},\mathit{right}\}}(x)$, which intuitively is the set of locations reachable from $x$ using the pointer fields *left* and *right*, and can be defined recursively.

As shown in Figure 7.2, $\varphi_{pre}$ is a conjunction of two formulas. If $G_{pre}$ is the domain of the heaplet associated to $\varphi_{pre}$, then the first conjunct requires $G_{pre}$ to be the disjoint union of the sets $\{x\}$, $\mathit{reach}_{\{\mathit{left},\mathit{right}\}}(\mathit{left}(x))$ and $\mathit{reach}_{\{\mathit{left},\mathit{right}\}}(\mathit{right}(x))$, that is,

$$G_{pre} = \{x\} \;\cup\; \mathit{reach}_{\{\mathit{left},\mathit{right}\}}(\mathit{left}(x)) \cup\; \mathit{reach}_{\{\mathit{left},\mathit{right}\}}(\mathit{right}(x))$$

The second conjunct requires $G_{pre} = \mathit{reach}_{\{\mathit{left},\mathit{right}\}}(x)$. From these heaplet constraints, we can translate $\varphi_{pre}$ to the following formula in classical logic

*over the global heap*:

$$
\begin{aligned}
& G_{pre} = \{x\} \cup reach_{\{left,right\}}(left(x)) \cup reach_{\{left,right\}}(right(x)) \\
\wedge\ \ & x \notin reach_{\{left,right\}}(left(x)) \wedge x \notin reach_{\{left,right\}}(right(x)) \\
\wedge\ \ & reach_{\{left,right\}}(left(x)) \cap reach_{\{left,right\}}(right(x)) = \emptyset \wedge x \neq nil \\
\wedge\ \ & mheap(left(x)) \wedge mheap(right(x)) \\
\wedge\ \ & G_{pre} = reach_{\{left,right\}}(x) \wedge keys(x) = K
\end{aligned}
$$

Similarly, we translate $\varphi_{post}$ to

$$G_{post} = reach_{\{left,right\}}(x) \wedge mheap(x) \wedge keys(x) = K$$

Note that the recursive definitions *mheap* and *keys* without the "Δ" superscript are in the classical logic (without the heaplet constraint). Hence the recursive predicate *mheap* satisfies

$$
\begin{aligned}
mheap(x) \leftrightarrow\ & x = nil \wedge reach_{\{left,right\}}(x) = \emptyset) \\
\vee\ & \Big(x \neq nil \wedge x \notin reach_{\{left,right\}}(left(x)) \wedge x \notin reach_{\{left,right\}}(right(x)) \\
& \wedge reach_{\{left,right\}}(left(x)) \cap reach_{\{left,right\}}(right(x)) = \emptyset \\
& \wedge \big(reach_{\{left,right\}}(x) = \{x\} \cup reach_{\{left,right\}}(left(x)) \\
& \qquad \cup reach_{\{left,right\}}(right(x))\big) \\
& \wedge mheap(left(x)) \wedge \{key(x)\} \geq keys(left(x)) \\
& \wedge mheap(right(x)) \wedge \{key(x)\} \geq keys(right(x))\Big)
\end{aligned}
$$

Notice that in the translation of $mheap^{\Delta}_{\overrightarrow{pf}}$, we express all heaplet constraints on reachset set directly, since in DRYAD$_{sep}$ the domain of the heaplet associated with a recursive predicate is syntactically determined to be the reachset set. The recursive funtion *keys* is also defined recursively in the same fashion and skipped in this section.

All these conditions involving heaplets, reach-sets, scoping, modified-sets, etc., can be formulated in logic using set theory, and Section 6.5 describes this in detail. Finally, the verification condition is written using a logic over the global heap, but referring only to the footprint, and the recursive definitions are formula-abstracted, resulting in a formula in a decidable theory, whose proof is then attempted.

### 7.5.2 VC Generation

To illustrate how natural proofs work in the `heapify` example, let us take a closeup look at one particular paths. The right side of Figure 7.2 presents a basic path from method `heapify`, corresponding to the case when both children of $x$ are not *nil* and the key of the right child is greater than the keys of the left child and the root. The subscript of a pointer/data field denotes the timestamp. Corresponding to the case when both children of $x$ are not *nil* and the key of the left child is greater than the key of the right child and the root. A key insight is that any basic path touches a finite number of locations and may call some recursive procedures. We refer to the touched locations as the *footprint*, and to the adjacent locations which are not part of the footprint as the *frontier*. For this example, let subscript 0, 1, 2 indicate the recursive definitions at the start, just before the call to `heapify`, and after the function call returns, respectively. Then the footprint is $\{\ x,\ lx,\ rx\ \}$ ($s$ is known to be equal with $rx$) and the frontier is $\{\ left_0(lx),\ right_0(lx),\ left_0(rx),\ right_0(rx)\ \}$.

We capture the effect of the path until the call to `heapify` by

$$
\begin{aligned}
& left_0(x) \neq nil \wedge right_0(x) \neq nil \wedge lx = left_0(x) \wedge rx = right_0(x) \\
\wedge\ \ & key_0(lx) < key_0(rx) \wedge\ \ s = right_0(x) \wedge s \neq nil \\
\wedge\ \ & key_0(s) > key_0(x) \wedge t = key_0(s) \\
\wedge\ \ & key_1 = key_0\{s \leftarrow key_0(x)\} \wedge key_2 = key_1\{x \leftarrow t\}
\end{aligned}
$$

To get the verification condition for the entire path, we conjunct the precondition $\varphi_{pre}$, the effect of the path before the function call, the effect of the function call and the formulae that computes the recursive definitions for the footprint in terms of the values at the frontier locations, and check if it implies the postcondition $\varphi_{post}$. This procedure has been formally described in Section 7.2.

### 7.5.3 Natural Proofs

Once we have expressed the verification condition in classical logic with recursive definitions over the global heap, we prove it using the *natural proof* methodology. A key issue is tracking the recursive predicates and functions across a basic path. We need to evaluate these recursive definitions at the beginning and the end of the path, and also before and after every procedure

call to incorporate the effect of the procedure call. We "compute" these by expressing the definitions of them on nodes within the footprint using their recursive definitions. Furthermore, at frontier locations, we know that if the corresponding reach set from that location hasn't changed due to the basic path working on the footprint (i.e., the reach sets from the location do not involve footprint nodes), then the recursive definitions on the frontier does not change. Similarly, if a procedure is called and its pre-condition defines a heaplet that is disjoint from the reach-set of a location, then we can retain the value for a recursive definition for that location across a call to the procedure; otherwise it will have to be updated conservatively taking into account the pre/post condition of the called procedure.

Technically, we unfold the recursive definitions $mheap(x)$, $keys(x)$ and $reach_{\{left,right\}}(x)$ for $x$, $lx$ and $rx$ (the footprint), thus evaluating them in terms of their values on $left_0(lx)$, $right_0(lx)$, $left_0(rx)$ and $right_0(rx)$ (the frontier). The call to `heapify` preserves the recursive definitions on locations reachable from $lx$, and modifies those on $rx$ according to the pre/post condition.

If $mheap_i$, $keys_i$ and $reach_i$ are the recursive definitions at the $i^{th}$ instance, then we can compute the values of $mheap_i$, $keys_i$ and $reach_i$ for the locations in the footprint by instantiating their definitions given above, in terms of the values of the recursive definitions at the frontier. For frontier locations, we notice that the basic path till the call to `heapify` only modifies the fields of locations $x$ and $s$. If neither of $x$ and $s$ is reachable from a frontier location $l$, then the value of the recursive definitions at $l$ remains unchanged across the path. Then for any frontier location $l$

$$\begin{aligned}&\{x, s\} \cap reach_0(l) = \emptyset \\ &\Rightarrow mheap_1(l) = mheap_0(l) \wedge keys_1(l) = keys_0(l) \wedge reach_1(l) = reach_0(l)\end{aligned}$$

In our example, the left-hand-side of the implication is true for all frontier locations because $\varphi_{pre}$ states that $x$ points to a tree.

To handle the call to `heapify`, let $G_1$ and $G_2$ be the domains of the heaplets of $\varphi_{pre}$ and $\varphi_{post}$ when instantiated with the actual call argument $s$. Also, let $key_3$, $left_1$ and $right_1$ be the data and the pointer fields after the call. If a footprint location $l$ is not in the domain of the precondition i.e. $G_1$, then it is not affected by the function call. Also it is not present in the domain of

the postcondition $G_2$:

$$\begin{aligned} & l \notin G_1 \\ \Rightarrow & l \notin G_2 \wedge key_3(l) = key_2(l) \wedge \; left_1(l) = left_0(l) \wedge right_1(l) = right_0(l) \end{aligned}$$

Similarly, if the set of reachable nodes of a frontier location $l$ does not intersect with the domain of the call, then the values of its recursive definitions are unchanged across the call:

$$\begin{aligned} & G_1 \cap \mathrm{reachset}_0(l) = \emptyset \\ & \Rightarrow mheap_2(l) = mheap_1(l) \wedge keys_2(l) = keys_1(l) \wedge reach_2(l) = reach_1(l) \end{aligned}$$

Finally, we abstract the recursive definitions on the frontier with uninterpreted functions, and encode the relations between integer sets into APF:

$$\begin{aligned} & left_0(x) \neq nil \wedge right_0(x) \neq nil \wedge lx = left_0(x) \wedge rx = right_0(x) \\ \wedge \; & \forall i_1, i_2. \left( \, i_1 \leq i_2 \rightarrow (\neg key_0(rx)[i_1] \vee \neg key_0(lx)[i_2]) \, \right) \\ \wedge \; & \; s = right_0(x) \wedge s \neq nil \\ \wedge \; & \forall i_1, i_2. \left( \, i_1 \leq i_2 \rightarrow (\neg key_0(s)[i_1] \vee \neg key_0(x)[i_2]) \, \right) \\ \wedge \; & t = key_0(s) \\ \wedge \; & key_1 = key_0\{s \leftarrow key_0(x)\} \wedge key_2 = key_1\{x \leftarrow t\} \end{aligned}$$

We decide the resulted formula (which is in a decidable logic) using an SMT solver. This process has been described in detail in Section 7.4.

## 7.6 Experimental Evaluation

We have implemented a prototype of the natural proof methodology for $\textsc{Dryad}_{\text{sep}}$ presented in this chapter. The prototype verifier takes as input a set of user-defined recursive definitions, a set of procedure declarations with contracts, and a set of straight-line programs (or *basic blocks*) annotated with a pre-condition and a post-condition specifying a set of partial correctness properties including structural, data and separation requirements. Both the contracts and pre-/post-conditions are written in $\textsc{Dryad}_{\text{sep}}$. For each basic block, the verifier automatically generates the abstracted formula $\psi^{APF}$ as described in Section 7.3 and 7.4, and passes $\psi^{APF}$ to Z3 [28], a state-of-the-art SMT solver, to check the satisfiability in the decidable theory of array

property fragment. The front-end of our verifier is based on ANTLR and our tool is around 4000 lines of C# code.

### 7.6.1 Standard Routines

Using the verifier, we successfully proved the partial correctness of 59 routines over a large class of programs involving heap data structures like sorted lists, doubly-linked lists, cyclic lists and trees. data-structures including sorted lists, doubly-linked lists, cyclic lists and trees. Experimental details are available at `http://web.engr.illinois.edu/~qiu2/dryad` .

We conducted the experiments on a machine with a dual-core, 2.4GHz CPU and 6GB RAM. The first part of our experimental results is tabulated in Table 7.1. In general, for every routine, we checked the properties formalizing the complete verification of the routines— capturing the precise structure of the resulting heap-structure, the precise change to the data stored in the nodes and the precise heaplet modified by the respective routines.

For every routine, the suffix `rec` or `iter` indicates if the routine was implemented recursively or iteratively using while loops. The names for most of the routines are self-descriptive. Routines like `find`, `insert`, `delete`, `append`, etc. are the natural implementations of the corresponding data structure operations. The routine `delete_all` for singly-linked lists, sorted lists and doubly-linked lists recursively deletes all occurrences of a particular key in the input list. The max-heap routine `heapify` accepts an almost max-heap in which the heap property is violated only at the root, both of whose children are max-heaps, and recursively descends the tree to restore the max-heap property. The routine `remove_root` for binary search trees and treaps is an auxiliary routine which is called in `delete`. Similarly, the routines `leftmost` for AVL-trees and RB-trees and `delete_fix` and `insert_fix` for RB-trees are also auxiliary routines.

Schorr-Waite is a well-known graph marking algorithm which marks all the reachable nodes of the graph using very little additional space. The algorithm achieves this by manipulating the pointers in the graph such that the stack of nodes along the path from the root is encoded in the graph itself. The Schorr-Waite algorithm is used in garbage collectors and it is traditionally considered as a challenging problem for verification [39]. The

| Data-structure | Routine | Time (s) / Routine |
|---|---|---|
| Singly-Linked List | find_rec, insert_front, insert_back_rec, delete_all_rec, copy_rec, append_rec, reverse_iter | < 1s |
| Sorted List | find_rec, insert_rec, merge_rec, delete_all_rec, insert_sort_rec, reverse_iter, find_last_iter | < 1s |
| | insert_iter | 1.4 |
| | quick_sort_iter | 64.8 |
| Doubly-Linked List | insert_front, insert_back_rec, delete_all_rec, append_rec, mid_insert, mid_delete, meld | < 1s |
| Cyclic List | insert_front, insert_back_rec, delete_front, delete_back_rec | < 1s |
| Max-Heap | heapify_rec | 8.8 |
| BST | find_rec, find_iter, insert_rec, delete_rec, remove_root_rec | < 1s |
| | insert_iter | 72.4 |
| | find_leftmost_iter | 4.7 |
| | remove_root_iter | 65.6 |
| | delete_iter | 225.2 |
| Treap | find_rec, delete_rec | < 1s |
| | insert_rec | 12.7 |
| | remove_root_rec | 9.5 |
| AVL-Tree | balance, leftmost_rec | < 1s |
| | insert_rec | 4.1 |
| | delete_rec | 13.9 |
| RB-Tree | insert_rec | 73.9 |
| | insert_left_fix_rec | 8.1 |
| | insert_right_fix_rec | 5.1 |
| | delete_rec | 12.1 |
| | delete_left_fix_rec | 7.6 |
| | delete_right_fix_rec | 5.5 |
| | leftmost_rec | < 1s |
| Binomial Heap | find_min_rec | 1.1 |
| | merge_rec | 152.7 |
| Schorr-Waite (for trees) | marking_iter | < 1s |
| Tree Traversals | inorder_tree_to_list_rec | 2.4 |
| | inorder_tree_to_list_iter | 42.7 |
| | preorder_rec, postorder_rec | < 1s |
| | inorder_rec | 3.76 |

Table 7.1: Results of verifying data-structure algorithms more details at `http://web.engr.illinois.edu/~qiu2/dryad` .

routine `marking` is an implementation of Schorr-Waite for trees [49] and we check the property that the resulting output tree is well-marked.

The routines `inorder_tree_to_list` construct a list consisting of the keys of the input tree, which is traversed inorder. The iterative version of this algorithm achieves this by maintaining a worklist/stack of sub-trees which remain to be processed at any given time. The routines `inorder`, `preorder` and `postorder` number the nodes of an input tree according to the inorder, preorder and postorder traversal algorithm, respectively.

### 7.6.2 Open-Source Libraries

Additionally, we pit our natural proofs methodology against real-world programs and successfully verified, in total, 47 routines from different projects including the list and queue implementations in the Glib open source library, the OpenBSD library, the Linux kernel and the memory regions and the page cache implementations from two different operating systems.

Table 7.2 shows the results of applying natural proofs to the verification of various other real world programs and libraries. Glib is the low-level C library that forms the basis of the GTK+ toolkit and the GNOME desktop environment, apart from other open source projects. Using our prototype verifier, we efficiently verified Glib implementation of various routines for manipulating singly-linked and doubly-linked lists. We also verified the queue library which forms part of the OpenBSD operating system.

ExpressOS is an operating-system/browser implementation which provides security guarantees to the user via formal verification [54]. The module `cachePage` maintains a cache of the recently used disc pages. The cache is implemented as a priority queue based on a sorted list. We prove that the methods `add_cachepage` and `lookup_prev` (both called whenever a disc page is accessed) maintain the sortedness property of the cache page.

In an OS kernel, a process address space consists of a set of intervals of linear addresses represented as a *memory region*. In the ExpressOS implementation, a memory region is implemented as a sorted doubly-linked list where each node of the list with a *start* and an *end* address represents an interval included in the address space. We also verified some key components of the Linux implementation of a memory region, present in the file *mmap.c*.

| Example | Routine | Time (s) / Routine |
|---|---|---|
| **glib/gslist.c** Singly Linked-List LOC: 1.1K | free, prepend, concat, insert_before, remove_all, remove_link, delete_link, copy, reverse, nth, nth_data, find, position, index, last, length | < 1s |
| | append | 4.9 |
| | insert_at_pos | 11.4 |
| | remove | 3.1 |
| | insert_sorted_list | 16.6 |
| | merge_sorted_lists | 6.1 |
| | merge_sort | 3.0 |
| **glib/glist.c** Doubly Linked-List LOC: 0.3K | free, prepend, reverse, nth, nth_data, position, find, index, last, length | < 1s |
| **OpenBSD/queue.h** Queue LOC: 0.1K | simpleq_init, simpleq_remove_after | < 1s |
| | simpleq_insert_head | 1.6 |
| | simpleq_insert_tail | 3.6 |
| | simpleq_insert_after | 18.3 |
| | simpleq_remove_head | 2.1 |
| **ExpressOS/cachePage.c** LOC: 0.1K | lookup_prev | 2.4 |
| | add_cachepage | 6.4 |
| **ExpressOS/ memoryRegion.c** LOC: 0.1K | memory_region_init | < 1s |
| | create_user_space_region | 3.6 |
| | split_memory_region | 5.8 |
| **linux/mmap.c** LOC: 0.1K | find_vma, remove_vma, remove_vma_list | < 1s |
| | insert_vm_struct | 11.6 |

Table 7.2: Results of verifying open-source libraries
more details at `http://web.engr.illinois.edu/~qiu2/dryad` .

In Linux, a memory region is represented as a red-black tree where each node, again, represents an address interval. We proved methods which find, remove and insert a *vma_struct* (*vma* is short for virtual memory address) into a memory region.

### 7.6.3 Evaluation

Note that our natural proof mechanism presented in Chapter 5 can only prove tree properties, and further, is restricted to extremely stringent conditions of pre- and post-conditions (for example, two trees cannot be given as parameters for a procedure, a procedure cannot modify the input node and return another tree, etc.).

However, on the tree data-structure examples, the algorithm in Chapter 5 perform better as it has been honed to work only for trees, where graph algorithms on footprints check the tree property as opposed to the logical mechanism dealing with it. The technique in this chapter is more general, and can handle arbitrary separation property expressed in $\text{DRYAD}_{\text{sep}}$.

The results for $\text{DRYAD}_{\text{tree}}$ do suggest that for certain structural properties, moving their check to a simpler graph algorithm outside of logic may be more efficient in practice. Another big difference is that $\text{DRYAD}_{\text{tree}}$ can only handle functional recursion and does not support while-loops. In contrast, $\text{DRYAD}_{\text{sep}}$ supports both recursive and iterative programs. In fact this paper extends the natural proof methodology to partial heap-structures like list segments or partial trees which is essential for the verification of many while-loop programs. It is important to note that most of the routines in Figure 7.2 are implemented iteratively.

It also worth mentioning that in the process of experiments, we did make some unintentional mistakes, in writing both the basic blocks and the annotations. For example, forgetting to free the deleted node, or using $\wedge$ instead of $*$ in the specification between two disjoint heaplets, were common mistakes. In these cases, Z3 provided counter-examples to the verification condition that captured the essence of the bugs, and turned out to be very helpful for us to debug the specification. These debugging hints are usually not available in other incomplete proof systems.

Our experiments show that the natural proof methodology set forth in this paper is successful in efficiently proving full-functional correctness of a large variety of algorithms. Most of the VCs generated for the above examples were discharged by Z3 in a few seconds. To the best of our knowledge, this is the first automatic mechanism that can prove such a wide variety of algorithms correct, handling such complex properties of structure, data and separation.

## 7.7 Related Work

There is a rich literature on analysis of heaps in software. We omit discussing literature on general interactive theorem provers (like ISABELLE [62]) that require considerable manual guidance. We also omit a lot of work on analyzing shape properties of the heap [53, 77, 22, 30, 8], as they do not handle complex functional properties.

There are several proof systems and assistants for separation logic [68, 63] that incorporate proof heuristics and are incomplete. However, [9] gives a small decidable fragment of separation logic on lists which has been further extended in [17] to include a restricted form of arithmetic. Symbolic execution with separation logic has been used in [11, 10, 14] to prove structural specifications for various list and tree programs. These tools come hardwired with a collection of axioms and their symbolic execution engines check the entailment between two formulas modulo these axioms. VERIFAST [41], on the other hand, chooses flexibility of writing richer specifications over complete automation, but requires the user to provide inductive lemmas and proof tactics to aid verification. Similarly, BEDROCK [24] is a Coq library that aims at mostly automated (but not completely automated) procedures that requires some proof tactics to be given by the user to prove verification conditions. The idea of using regions (sets of locations) for describing heaps in our work also extends to describing frames for function calls, and the use for the latter is similar to implicit dynamic frames [71] in the literature. The crucial difference in our framework is that the implicit dynamic frames are syntactically determined, and amenable to quantifier-free reasoning. A work that comes very close to ours is a paper by Chin et al. [23], where the authors allow user-defined recursive predicates (similar to ours) and build a terminating procedure that reduces the verification condition to standard

logical theories. However, their procedure does not search for a proof in a well-defined simple and decidable class, unlike our natural proof mechanism; in fact, the resulting formulas are quantified and incompatible with decidable logics handled by SMT solvers.

In all of the above cited work and other manual and semi-automatic approaches to verification of heap-manipulating programs like [69], inductive definitions of algebraic data-types is extremely common for capturing second-order data-structure properties. Most of these approaches use proof tactics which unroll inductive definitions and do extensive unification to try to match terms to find simple proofs. Our notion of natural proofs is very much inspired by such kinds of semi-automatic and manual heap reasoning that we have seen in the literature.

There is also a variety of verification tools based on classical logics and SMT solvers. DAFNY [47] and VCC [26] compile to BOOGIE [5] and generate VCs that are passed to SMT solvers. This approach requires significant ghost annotations, and annotations that explicitly express and manipulate frames. The JAHOB system [81, 82] is one of the first attempts at full functional verification of linked data structures, which integrates a variety of theorem provers, including SMT solvers, and makes the process mostly automated. However, complex specifications combining structure, data and separation usually require more complex provers such as MONA [42], or even interactive theorem provers such as ISABELLE [62] in the worst case. The success of the proof search also relies on users' manual guidance.

The idea of unfolding recursive definitions and formula abstraction also features in the work by Suter et al. [72, 73], where a procedure for algebraic data-types is presented. However, this work focuses on soundness and completeness, and is not terminating for several complex data structures like red-black trees. Moreover, the work limits itself to functional program correctness; in our opinion, functional programs are very similar to algebraic inductive specifications, leading to much simpler proof procedures.

There is also a rich literature on completely automatic decision procedures for restricted heap logics, some of which combine structure-logic and arbitrary data-logic. These logics are usually FOLs with restricted quantifiers, and usually are decided using SMT solvers. The logics LISBQ [46] and CSL [15, 16] offer such reasoning with restricted reachability predicates and quantification; see also the logics in [12, 65, 66, 59, 67, 3]. STRAND

presented in this dissertation is a relatively expressive logic that can handle some data-structure properties (like BSTs) and admits decidable fragments, but is again not expressive enough for more complex properties of inductive data-structures. None of these logics can express the class of VCs for full functional verification explored in this paper.

## 7.8 Annotation Synthesis

In this chapter, we have presented how to build an automatic verifier, which encodes the natural proof strategy for $\text{DRYAD}_{\text{sep}}$ into logical formulas that can be handled by SMT solvers. However, we believe natural proofs, as a novel and fundamental methodology, should not only lead to dedicated, stand-alone verifiers, but also benefit existing verification frameworks and tools. For example, in recent years, a number of tools have been developed with the aim of making the verification process mostly automated. These tools varies in the logics they support (e.g., classical logic [26, 47] or SL [41]), in their underlying proving engine (e.g., SMT solvers [26, 47] or MONA/ISABELLE [81, 82]), but they all rely on user-provided lemmas and/or ghost code. The programmer may provide hints or proof heuristics through these additional annotations. This task could be very challenging and tedious for ordinary programmers, and make the verification process much less automatic. See why through an motivating example as follows.

### 7.8.1 An Motivating Example

Let us consider an BST example verified by VERIFAST[2]:

```
#include "stdlib.h"
#include "assert.h"

struct tree{
  int value;
  struct tree *left;
```

[2]http://people.cs.kuleuven.be/~bart.jacobs/verifast/examples/sorted_bintree.c.html

```
  struct tree *right;
};

predicate tree(struct tree *t,bintree b)
  requires switch(b){
    case empty: return t==0;
    case tree(a,bl,br): return t->value |-> ?v &*&
        t->left |-> ?l &*& t->right |-> ?r &*&
        malloc_block_tree(t) &*& tree(l,bl) &*& tree(r,br)
        &*& t!=0 &*& a==v ;
  }&*& inorder(b)==true;

inductive bintree = |empty |tree(int,bintree,bintree);

fixpoint bool t_contains(bintree b, int v) {
  ...
}
fixpoint bintree tree_add(bintree b, int x) {
  ...
}
fixpoint int max(bintree b){
  ...
}
fixpoint int min(bintree b){
  ...
}
fixpoint bool inorder(bintree b){
  ...
}

lemma void max_conj_add(bintree l,int v,int x)
  requires x < v &*& (max(l)<v||l==empty) &*& inorder(l)==true;
  ensures max(tree_add(l,x))<v &*& inorder(l)==true;
{
  switch(l){
    case empty:
```

```
    case tree(a,b,c):if(x < a){
              max_conj_add(b,a,x);
          }
             if(a < x){
              max_conj_add(c,v,x);
          }
  }
}
lemma void min_conj_add(bintree r,int v,int x)
  requires v < x &*& (v < min(r)||r==empty) &*&
              inorder(r)==true;
  ensures v < min(tree_add(r,x)) &*& inorder(r)==true;
{
  switch(r){
    case empty:
    case tree(a,b,c):
          if(a < x){
              min_conj_add(c,a,x);
          }
          if(x < a){
              min_conj_add(b,v,x);
          }
  }
}

lemma void tree_add_inorder(bintree b, int x)
    requires inorder(b)==true &*& t_contains(b,x)==false;
    ensures inorder(tree_add(b,x))==true &*&
                t_contains(tree_add(b,x),x)==true;
{
  switch (b) {
    case empty:
    case tree(v,l,r):
          if(x < v){
              max_conj_add(l,v,x);
              tree_add_inorder(l,x);
```

```
        }
        if(v < x){
            min_conj_add(r,v,x);
            tree_add_inorder(r,x);
        }
  }
}

void add(struct tree *t, int x)
//@ requires tree(t,?b) &*& b!=empty &*&
false==t_contains(b,x) &*& inorder(b)==true;
//@ ensures tree(t,tree_add(b,x)) &*&
inorder(tree_add(b,x))==true;
 {
  //@ open tree(t,b);
  int v=t->value;
  struct tree *l=t->left;
  //@ open tree(l,?bl);
  //@ close tree(l,bl);
  struct tree *r=t->right;
  //@ open tree(r,?br);
  //@ close tree(r,br);
  if(x < v){
    if(l!=0){
      add(l,x);
      //@ tree_add_inorder(b,x);
      //@ close tree(t,tree(v,tree_add(bl,x),br));
    }else{
      struct tree *temp=init_tree(x);
      t->left=temp;
      //@ open tree(l,bl);
      //@ close tree(t,tree(v,tree(x,empty,empty),br));
      //@ tree_add_inorder(b,x);
    }
  }else{
    if(v < x){
```

```
      if(r!=0){
        add(r,x);
        //@ tree_add_inorder(b,x);
        //@ close tree(t,tree(v,bl,tree_add(br,x)));
      }else{
        struct tree *temp=init_tree(x);
        t->right=temp;
        //@ open tree(r,br);
        //@ close tree(t,tree(v,bl,tree(x,empty,empty)));
      }
    }
  }
}
```

The BST consists of structured nodes of type `tree`. The `add` function inserts a new node with value `x` into a tree with root pointed by `t`. The `requires` and `ensures` clauses specify the partial correctness of `add`: the expected inputs are a non-empty tree pointed by `t` and an integer `x`, and the keys in the tree form an inorder `bintree` structure `b` that does not contain `x`; upon the end of the program, the keys stored under `t` still form an inorder `bintree` structure `tree(t,tree_add(b,x))`. The above specification relies on the definition of a recursive predicate `tree(t,b)` and an inductive data type `bintree`.

To verify this particular function, the programmer instruments 14 ghost annotations in 3 categories: the `open` clause, the `close` clause, and the `tree_add_inorder` lemma. Intuitively, the `open` and `close` manually unfold or fold the recursive definitions with respect to a particular node, respectively. The `tree_add_inorder` lemma asserts the relation between `b` and `tree_add(b,x)`. This lemma is described as a function and proved by verifying the lemma function. Furthermore, proving `tree_add_inorder` in turn relies on two additional lemmas: `max_conj_add` and `min_conj_add`. In addition, these lemmas use 5 pure functions which are recursively defined: `t_contains`, `tree_add`, `max`, `min` and `inorder`. There definitions are omitted here.

The VeriFast tool successfully verifies this program against such complex specifications, in a completely automatic fashion. Nevertheless, the price to

pay is all the ghost annotations, lemmas and functions, which is literally at least three times longer than the minimal portion that implements the functionality. Writing these extra annotations also far exceeds the capability of any ordinary programmer without specific training.

Therefore, our goal is to extricate programmers from the extra burden of developing their own proof heuristics and encoding them into the daunting ghost annotations/lemmas. In particular, we hope the natural proofs, as a pre-defined tactic that is practically useful for many correct programs, can be automatically encoded as a set of ghost annotations, and leave only the necessary coding task to the programmer: the program itself as well as pre-/post-conditions.

### 7.8.2 Recursive Definitions for VCC

In this section, we consider $\text{D}\textsc{ryad}_{\text{sep}}$, and implement natural proofs for C programs by automatically encoding natural proof tactics into carefully crafted ghost-code annotations on a C program using classical logical annotations that ensure verification conditions fall into decidable logics. These automatic annotations then help the tool VCC to carry out an automatic proof of the C program, completely freeing the engineer from guiding proofs. We implement our annotation synthesis as an extension of VCC, and the preliminary results show that a class of challenging heap-manipulating programs can be automatically verified.

We now show how the "VCC + Natural Proof" mechanism works through a simple example: reversing a sorted list. First, the programmer writes the program as well as the $\text{D}\textsc{ryad}_{\text{sep}}$ annotations:

```
#include "sll.dryad"

Node * reverse_sorted(Node * l)
    _(requires _dryad_srtl^(l))
    _(ensures _dryad_rsrtl^(\result))
    _(ensures _dryad_keys^(\result) == \old(_dryad_keys^(l)))
{
    Node * r = NULL;
```

```
    while(l != NULL)
        _(invariant _dryad_srtl^(l) * _dryad_rsrtl^(r))
        _(invariant \old(_dryad_keys^(l)) ==
            \intset_union(_dryad_keys^(l), _dryad_keys^(r)))
    {
        Node * t = l->next;

        l->next = r;
        r = l;
        l = t;
        }
    return r;
}
```

The annotations above are written in DRYAD$_{sep}$ and just consist of necessary parts: pre-/post-conditions, and loop invariants. The `_dryad_srtl^` and `_dryad_rsrtl^` are recursive predicates for sorted list and reverse-sorted list, respectively; and the `_dryad_keys^` is the recursive function getting the set of keys stored. All these definitions along with their corresponding reach set definitions are defined in a separate head file `sll.dryad`.

As a preprocessing step, our transformer translates the DRYAD$_{sep}$ definitions and annotations into classical logical formulas in the form recognized by standard VCC, following the procedure described in Section 6.5. First, the transformer generates a head file `sll.h`, which mimics every recursive definition in `sll.dryad` using a set of ghost pure functions. For example, given the predicate `_dryad_srtl^` which is recursively defined as

```
define pred dryad_srtl^(x):
  (
    ((x l= nil) & emp) |
      ((x |-> loc next: nxt; int key: ky) *
           (dryad_srtl^(nxt) & (ky lt-set dryad_keys^(nxt)))
         )
  ) ;
```

the `sll.h` file declares both `_dryad_srtl` and its corresponding reach set `_dryad_N`, and defines how they should be unfolded:

```
_(ghost _:pure \bool _dryad_srtl(struct node * head)
    _(reads \universe())
;)

_(ghost _:pure \objset _dryad_N(struct node *head)
    _(reads \universe())
;)

_(pure \bool _dryad_unfold_srtl(struct node * x)
    _(reads \universe())
    _(ensures \result == (_dryad_srtl(x) == (
        (x == NULL && _dryad_N(x) == {}) ||
        ( x \in \universe() && _dryad_srtl(x->next) &&
          _dryad_N(x->next) \union _dryad_N(x->next)
              == _dryad_N(x->next) &&
          \intset_lt_set(x->key, _dryad_keys(x->next)) &&
          dryad_N(x->next) \union _dryad_N(x->next)
              == _dryad_N(x->next) &&
          {x} \union _dryad_N(x->next) \union _dryad_N(x->next)
              == _dryad_N(x) )
    )) )
;)

_(pure \bool _dryad_unfold_N(struct node * hd)
    _(reads \universe())
    _(ensures \result == (
        (hd == NULL && _dryad_N(hd) == {}) ||
        (hd != NULL &&
          _dryad_N(hd) == ({hd} \union _dryad_N(hd->next)))
    ));)
```

We omit the similar definitions for `_dryad_rsrtl`, `_dryad_unfold_rsrtl`, `_dryad_keys` and `_dryad_unfold_keys`.

Secondly, the transformer replaces each $\text{DRYAD}_{\text{sep}}$-style specification with a VCC-style specification. For instance, the loop invariant

```
_(invariant _dryad_srtl^(l) * _dryad_rsrtl^(r))
```

gets translated to the following ones:

```
_(invariant _dryad_srtl(l))
_(invariant _dryad_rsrtl(r))
_(invariant \disjoint(_dryad_N(l), _dryad_N(r)))
```

### 7.8.3 Natural Proofs for VCC

After the above translation, instead of the generating the VC and apply the natural proofs in steps as we set forth in chapter, our annotation synthesizer simply instrument assumptions at the beginning and the end of each basic block. These assumptions encode the known facts about the current program states: unfoldings and framings. These assumptions are essentially the same as the formula UNFOLDANDFRAME formulated in Section 7.4, but written in the VCC-specification format. To this end, we also define several macros in `sll.h` to make the instrumentation succinct and intuitive. In this particular example, these macros are defined as:

```
_(ghost _:pure \bool _dryad_unfoldAll(\object o)
   _(reads \universe())
   _(ensures _dryad_unfold_N(o))
   _(ensures _dryad_unfold_keys(o))
   _(ensures _dryad_unfold_srtl(o))
   _(ensures _dryad_unfold_rsrtl(o))
;)

_(logic \bool _dryad_recUnchanged(struct node * x,
   struct node * y, \state enter, \state exit) =
   ((! (x \in \at(enter, _dryad_N(y)))) ==>
     \at(enter, _dryad_N(y)) == \at(exit, _dryad_N(y)))
   && ((! (x \in \at(enter, _dryad_N(y)))) ==>
     \at(enter, _dryad_keys(y)) == \at(exit, _dryad_keys(y)))
   && ((!(x \in \at(enter, _dryad_N(y)))) ==>
```

```
      \at(enter, _dryad_srtl(y)) == \at(exit, _dryad_srtl(y)))
    && ((!(x \in \at(enter, _dryad_N(y)))) ==>
      \at(enter, _dryad_rsrtl(y)) == \at(exit, _dryad_rsrtl(y)))
    &&
    (\disjoint(\at(enter, _dryad_N(x)),
          \at(enter, _dryad_N(y))) ==>
        \at(enter, _dryad_N(x)) == \at(exit, _dryad_N(x)))
    && (\disjoint(\at(enter, _dryad_N(x)),
          \at(enter, _dryad_N(y))) ==>
        \at(enter, _dryad_keys(x)) == \at(exit, _dryad_keys(x)))
    && (\disjoint(\at(enter, _dryad_N(x)),
          \at(enter, _dryad_N(y))) ==>
        \at(enter, _dryad_srtl(x)) == \at(exit, _dryad_srtl(x)))
    && (\disjoint(\at(enter, _dryad_N(x)),
          \at(enter, _dryad_N(y))) ==>
        \at(enter, _dryad_rsrtl(x)) ==
          \at(exit, _dryad_rsrtl(x)))
;)
```

Let $c$ be the current program state, then intuitively, `_dryad_unfoldAll` unfolds each recursive definition (including reach sets) at $c$, `_dryad_recUnchanged` mimics the formula $\textsc{RecUnchanged}_c^{rec}$ for each recursive definition *rec*.

When `sll.h` is included, the instrumented program is generated as below:

```
#include "sll.h"

struct node* reverse_sorted(struct node* l)
  requires _dryad_srtl(l);
  ensures _dryad_rsrtl(result);
  ensures unchecked==(_dryad_keys(result),
                      old(@prestate, _dryad_keys(l)));
{
  // --- Dryad annotated function ---
  _math \objset _dryad_G0;
  _math \objset _dryad_G1;
```

```
  @spec(@=(_dryad_G0, _dryad_N(l)))
  @spec(@=(_dryad_G1, _dryad_G0))
  struct node* l;
  {
    assume _dryad_unfoldAll((checked \object)l);
    struct node* r;
    assume mutable_list(l);
    @=(r, (checked struct node*)(checked void*)0)
    @while(@loop_contract(assert _dryad_srtl(l);
, assert _dryad_rsrtl(r);
, assert \disjoint(_dryad_N(l), _dryad_N(r));
, assert @ite(unchecked!=((checked void*)l,
        (checked void*)0), \intset_ge(*((l->
        key)), _dryad_keys(r)), true);
, assert unchecked==(old(@prestate, _dryad_keys(l)),
        \intset_union(_dryad_keys(l), _dryad_keys(r)));
, assert mutable_list(l);
), unchecked!=((checked void*)l, (checked void*)0), {
assume _dryad_unfoldAll((checked \object)r);
assume _dryad_unfoldAll((checked \object)l);
struct node* t;
assume unfoldMutable(l);
{
assume _dryad_unfoldAll((checked \object)l);
@=(t, *((l->next)))
assume _dryad_unfoldAll((checked \object)l);
}
{
_math \state _dryad_S0;
@spec(@=(_dryad_S0, @_vcc_current_state))
@=(*((l->next)), (checked struct node*)r)
assume _dryad_unfoldAll((checked \object)l);
_math \state _dryad_S1;
@spec(@=(_dryad_S1, @_vcc_current_state))
assume _dryad_fieldUnchanged(l, t, _dryad_S0, _dryad_S1);
assume _dryad_fieldUnchanged(l, r, _dryad_S0, _dryad_S1);
```

```
}
{
@=(r, (checked struct node*)l)
}
{
@=(l, (checked struct node*)t)
}
}
)
    return r;
    skip;
  }

}
```

These automatically generated annotations help VCC find a natural proof. When the unfoldings and framings are explicitly stated, the built-in engine of VCC automatically finished the rest of the verification in a few seconds.

This example shows the salient feature of the annotation synthesis from natural proofs: reducing most extra burden on writing proof annotations. Besides the program itself, the programmer just needs to write the minimum specification which is unavoidable, including pre-/post-conditions and loop invariants. In many cases, the synthesized annotations are already enough to carry out an automatic proof. Otherwise, if the verifier does not finish the proof immediately, What left to the programmer is usually only a few hints or lemmas that are specific to the program, so that the programmer can focus on the most creative part of the verification. In either case, we believe our annotation synthesizer paves the way for tractable analysis of separation logic for C programs manipulating heaps using the idea of natural proofs, and makes the deductive verification more approachable to ordinary programmers.

# CHAPTER 8

# CONCLUSIONS

This chapter first presents the conclusions of this dissertation, followed by a look ahead of future research directions.

## 8.1 Conclusions

For several decades, automated reasoning for program verification has been an intense research topic. Most people today are aware of a fact: there is no one-size-fits-all solution to the problem of buggy programs. Computer systems is used in so many different contexts that each different technique could be potentially useful. few programmers are willing to write extra annotations for less-critical software. However, for software systems whose reliability is highly critical, it may be justifiable to do so if highly automated techniques and tools are available to help the user.

This dissertation focuses on heap-manipulating programs and spans several important aspects of heap analysis and verification: data-structures, decidability, theorem proving, separation, and recursion. A key theme in this work is to develop new program logics and methodologies that strike a nice balance between expressiveness and verifiability in the area of verifying heap-manipulating programs. We have defined two logical framework, one called STRAND (in Chapter 3 and 4), and the other one called Natural Proofs (in Chapter 5, 6 and 7). STRAND is so far one of the most powerful decidable logical frameworks for complex properties combining heap structures and data. The natural proof scheme is an efficient terminating proof methodology that can automatically verify the full correctness of a wide variety of challenging programs, including a large number of programs from the GTK library, the OpenBSD library and the Linux kernel.

The two approaches complement each other: Strand sticks to the decidability, and can be used in not only proving programs correct but also in software analysis and software testing; Natural proofs aim at expressiveness, and could alleviate the programmers burden, making the proof technology more feasible to ordinary programmers.

We believe that this work paves the way for deductive verification technology to be used by programmers who do not (and need not) understand the internals of the underlying logic solvers, significantly increasing their applicability in building reliable systems. By combining these different approaches, we envision the emerging of innovative techniques that can help the user to build reliable software in a natural and efficient way, and hold promise of hatching the next-generation automatic verification techniques.

## 8.2 A Look Ahead

To make software development easier, more reliable, and more productive, several promising research directions can be explored in the future.

One major direction for future research is software repair and debugging. An interesting observation is that, when the natural proof strategy succeeds, the proof is usually very similar to the program itself. We believe when the natural proof strategy discovers errors in a portion of a high-assurance program, if the bug is due to some particular statements in the program, there should be a method that automatically suggests some way to repair it. In particular, the counterexample from the failed proof may suggest some revisions of the program, which may carry a natural proof. We envision such a tool will reduce programmers burden drastically toward developing reliable software.

Another important direction ahead is to express and synthesize loop invariants. Real-world programmers tend to write programs with loops rather than recursion. Imperative programs with while-loops usually have very complex loop invariants that are challenging to even express. For example, the loop invariant for traversing a tree talks about a partial-tree structure that consists of nodes before the current node in preordering. To express such an invariant, one usually has to use some higher-order logics on graphs. Two tasks are involved in this direction: first, developing a logical framework that

is amenable to succinctly expressing invariants; then, developing automatic invariant synthesis techniques based on such a logical framework.

# REFERENCES


[1] *Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings* (2009), vol. 5643 of *LNCS*, Springer.

[2] Balaban, I., Pnueli, A., and Zuck, L. D. Shape analysis by predicate abstraction. In *VMCAI'05* (2005), vol. 3385 of *LNCS*, Springer, pp. 164–180.

[3] Balaban, I., Pnueli, A., and Zuck, L. D. Shape analysis of single-parent heaps. In *VMCAI'07* (2007), vol. 4349 of *LNCS*, Springer, pp. 91–105.

[4] Ball, T., Majumdar, R., Millstein, T., and Rajamani, S. K. Automatic predicate abstraction of C programs. In *PLDI'01* (2001), ACM, pp. 203–213.

[5] Barnett, M., Chang, B.-Y. E., DeLine, R., Jacobs, B., and Leino, K. R. M. Boogie: A modular reusable verifier for object-oriented programs. In *FMCO'05* (2005), vol. 4111 of *LNCS*, Springer, pp. 364–387.

[6] Barrett, C., Conway, C. L., Deters, M., Hadarean, L., Jovanovic, D., King, T., Reynolds, A., and Tinelli, C. CVC4. In *CAV* (2011), vol. 6806 of *LNCS*, Springer, pp. 171–177.

[7] Barrett, C., Deters, M., de Moura, L. M., Oliveras, A., and Stump, A. 6 years of SMT-COMP. *J. Autom. Reasoning 50*, 3 (2013), 243–277.

[8] Berdine, J., Calcagno, C., Cook, B., Distefano, D., O'Hearn, P. W., Wies, T., and Yang, H. Shape analysis for composite data structures. In *CAV'07* (2007), vol. 4590 of *LNCS*, Springer, pp. 178–192.

[9] Berdine, J., Calcagno, C., and O'Hearn, P. W. A decidable fragment of separation logic. In *FSTTCS'04* (2004), vol. 3328 of *LNCS*, Springer, pp. 97–109.

[10] Berdine, J., Calcagno, C., and O'Hearn, P. W. Smallfoot: Modular automatic assertion checking with separation logic. In *FMCO'05* (2005), vol. 4111 of *LNCS*, Springer, pp. 115–137.

[11] Berdine, J., Calcagno, C., and O'Hearn, P. W. Symbolic execution with separation logic. In *APLAS'05* (2005), vol. 3780 of *LNCS*, Springer, pp. 52–68.

[12] Bjørner, N., and Hendrix, J. Linear functional fixed-points. In *CAV'09* [1], pp. 124–139.

[13] Börger, E., Grädel, E., and Gurevich, Y. *The Classical Decision Problem*. Springer, 2001.

[14] Botinčan, M., Parkinson, M., and Schulte, W. Separation logic verification of C programs with an SMT solver. *ENTCS 254* (2009), 5 – 23.

[15] Bouajjani, A., Drăgoi, C., Enea, C., and Sighireanu, M. A logic-based framework for reasoning about composite data structures. In *CONCUR'09* (2009), vol. 5710 of *LNCS*, Springer, pp. 178–195.

[16] Bouajjani, A., Drăgoi, C., Enea, C., and Sighireanu, M. On inter-procedural analysis of programs with lists and data. In *PLDI'11* (2011), ACM, pp. 578–589.

[17] Bozga, M., Iosif, R., and Perarnau, S. Quantitative separation logic and programs with lists. In *IJCAR'08* (2008), vol. 5195 of *LNCS*, Springer, pp. 34–49.

[18] Bradley, A. R., and Manna, Z. *The Calculus of Computation*. Springer, 2007.

[19] Bradley, A. R., Manna, Z., and Sipma, H. B. What's decidable about arrays? In *VMCAI'06* (2006), vol. 3855 of *LNCS*, Springer, pp. 427–442.

[20] Buchi, J. R. Weak second-order arithmetic and finite automata. *Z. Math. Logik Grundl. Math. 6* (1960), 66–92.

[21] Calcagno, C., Yang, H., and O'Hearn, P. W. Computability and complexity results for a spatial assertion language for data structures. In *FSTTCS* (2001), vol. 2245 of *LNCS*, Springer, pp. 108–119.

[22] Chang, B.-Y. E., and Rival, X. Relational inductive shape analysis. In *POPL'08* (2008), ACM, pp. 247–260.

[23] Chin, W.-N., David, C., Nguyen, H. H., and Qin, S. Automated verification of shape, size and bag properties via user-defined predicates in separation logic. *Science of Computer Programming 77*, 9 (2012), 1006 – 1036.

[24] Chlipala, A. Mostly-automated verification of low-level programs in computational separation logic. In *PLDI'11* (2011), ACM, pp. 234–245.

[25] Cimatti, A., Griggio, A., Schaafsma, B. J., and Sebastiani, R. The MathSAT5 SMT solver. In *TACAS* (2013), vol. 7795 of *LNCS*, Springer, pp. 93–107.

[26] Cohen, E., Dahlweid, M., Hillebrand, M. A., Leinenbach, D., Moskal, M., Santen, T., Schulte, W., and Tobies, S. VCC: A practical system for verifying concurrent C. In *TPHOLs'09* (2009), vol. 5674 of *LNCS*, Springer, pp. 23–42.

[27] Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. *Introduction to Algorithms*, third ed. MIT Press, 2009.

[28] de Moura, L. M., and Bjørner, N. Z3: An efficient SMT solver. In *TACAS'08* (2008), vol. 4963 of *LNCS*, Springer, pp. 337–340.

[29] Detlefs, D., Nelson, G., and Saxe, J. B. Simplify: a theorem prover for program checking. *J. ACM 52*, 3 (2005), 365–473.

[30] Distefano, D., O'Hearn, P. W., and Yang, H. A local shape analysis based on separation logic. In *TACAS'06* (2006), vol. 3920 of *LNCS*, Springer, pp. 287–302.

[31] Doner, J. Tree acceptors and some of their applications. *Journal of Computer and System Sciences 4*, 5 (1970), 406 – 451.

[32] Dutertre, B., and de Moura, L. The Yices SMT solver. Tech. rep., SRI International, 2006.

[33] Elgot, C. C. Decision problems of finite automata design and related arithmetics. *Trans. AMS 98* (1961), 21–52.

[34] Engelfriet, J. Context-free graph grammars. In *Handbook of Formal Languages*, vol. 3. Springer, 1997, pp. 125–214.

[35] Flanagan, C., Leino, K. R. M., Lillibridge, M., Nelson, G., Saxe, J. B., and Stata, R. Extended static checking for Java. In *PLDI'02* (2002), ACM, pp. 234–245.

[36] Ge, Y., and de Moura, L. M. Complete instantiation for quantified formulas in satisfiabiliby modulo theories. In *CAV* [1], pp. 306–320.

[37] Godefroid, P., Klarlund, N., and Sen, K. DART: directed automated random testing. In *PLDI'05* (2005), ACM, pp. 213–223.

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

[39] Hubert, T., and Marché, C. A case study of C source code verification: the Schorr-Waite algorithm. In *SEFM'05* (2005), IEEE-CS, pp. 190–199.

[40] INRIA. *The Coq Proof Assistant Reference Manual*. Available at `http://coq.inria.fr/`.

[41] Jacobs, B., Smans, J., Philippaerts, P., Vogels, F., Penninckx, W., and Piessens, F. Verifast: A powerful, sound, predictable, fast verifier for C and Java. In *NFM'11* (2011), vol. 6617 of *LNCS*, Springer, pp. 41–55.

[42] Klarlund, N., and Møller, A. *MONA*. BRICS, Department of Computer Science, Aarhus University, January 2001. Available from `http://www.brics.dk/mona/`.

[43] Klarlund, N., and Schwartzbach, M. I. Graph types. In *POPL'93* (1993), ACM, pp. 196–205.

[44] Kuncak, V. *Modular Data Structure Verification*. PhD thesis, Massachusetts Institute of Technology, 2007.

[45] Kuncak, V., Piskac, R., and Suter, P. Ordered sets in the calculus of data structures. In *CSL'10* (2010), vol. 6247 of *LNCS*, Springer, pp. 34–48.

[46] Lahiri, S., and Qadeer, S. Back to the future: revisiting precise program verification using SMT solvers. In *POPL'08* (2008), ACM, pp. 171–182.

[47] Leino, K. R. M. Dafny: An automatic program verifier for functional correctness. In *LPAR-16* (2010), vol. 6355 of *LNCS*, Springer, pp. 348–370.

[48] Lev-Ami, T., and Sagiv, S. Tvla: A system for implementing static analyses. In *SAS'00* (2000), vol. 1824 of *LNCS*, Springer, pp. 280–301.

[49] Loginov, A., Reps, T. W., and Sagiv, M. Automated verification of the Deutsch-Schorr-Waite tree-traversal algorithm. In *SAS'06* (2006), vol. 4134 of *LNCS*, Springer, pp. 261–279.

[50] Madhusudan, P., Parlato, G., and Qiu, X. Decidable logics combining heap structures and data. In *POPL'11* (2011), ACM, pp. 611–622.

[51] Madhusudan, P., and Qiu, X. Efficient decision procedures for heaps using STRAND. In *SAS'11* (2011), vol. 6887 of *LNCS*, Springer, pp. 43–59.

[52] Madhusudan, P., Qiu, X., and Ştefănescu, A. Recursive proofs for inductive tree data-structures. In *POPL'12* (2012), ACM, pp. 123–136.

[53] Magill, S., Tsai, M.-H., Lee, P., and Tsay, Y.-K. THOR: A tool for reasoning about shape and arithmetic. In *CAV'08* (2008), vol. 5123 of *LNCS*, Springer, pp. 428–432.

[54] Mai, H., Pek, E., Xue, H., King, S. T., and Madhusudan, P. Verifying security invariants in ExpressOS. In *ASPLOS'13* (2013), ACM, pp. 293–304.

[55] McPeak, S., and Necula, G. C. Data structure specifications via local equality axioms. In *CAV'05* (2005), vol. 3576 of *LNCS*, Springer, pp. 476–490.

[56] Meyer, A. R. Weak monadic second order theory of succesor is not elementary-recursive. In *Logic Colloquium* (1975), vol. 453 of *Lecture Notes in Mathematics*, Springer, pp. 132–154.

[57] Minsky, M. L. *Computation: finite and infinite machines.* Prentice-Hall, Inc., 1967.

[58] Møller, A., and Schwartzbach, M. I. The pointer assertion logic engine. In *PLDI'01* (2001), ACM, pp. 221–231.

[59] Nelson, G. Verifying reachability invariants of linked structures. In *POPL'83* (1983), ACM, pp. 38–47.

[60] Nelson, G., and Oppen, D. C. Simplification by cooperating decision procedures. *ACM Trans. Program. Lang. Syst. 1*, 2 (1979), 245–257.

[61] Nieuwenhuis, R., Oliveras, A., and Tinelli, C. Solving SAT and SAT Modulo Theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). *J. ACM 53*, 6 (2006), 937–977.

[62] Nipkow, T., Paulson, L. C., and Wenzel, M. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, vol. 2283 of *LNCS*. Springer, 2002.

[63] O'Hearn, P. W., Reynolds, J. C., and Yang, H. Local reasoning about programs that alter data structures. In *CSL'01* (2001), vol. 2142 of *LNCS*, Springer, pp. 1–19.

[64] Qiu, X., Garg, P., Ştefănescu, A., and Madhusudan, P. Natural proofs for structure, data, and separation. In *PLDI'13* (2013), ACM, pp. 231–242.

[65] Rakamarić, Z., Bingham, J. D., and Hu, A. J. An inference-rule-based decision procedure for verification of heap-manipulating programs with mutable data and cyclic data structures. In *VMCAI'07* (2007), vol. 4349 of *LNCS*, Springer, pp. 106–121.

[66] Rakamarić, Z., Bruttomesso, R., Hu, A. J., and Cimatti, A. Verifying heap-manipulating programs in an SMT framework. In *ATVA'07* (2007), vol. 4762 of *LNCS*, Springer, pp. 237–252.

[67] Ranise, S., and Zarba, C. A theory of singly-linked lists and its extensible decision procedure. In *SEFM'06* (2006), IEEE-CS, pp. 206–215.

[68] Reynolds, J. Separation logic: a logic for shared mutable data structures. In *LICS'02* (2002), IEEE-CS, pp. 55–74.

[69] Rosu, G., Ellison, C., and Schulte, W. Matching logic: An alternative to Hoare/Floyd logic. In *AMAST'10* (2010), vol. 6486 of *LNCS*, Springer, pp. 142–162.

[70] Shostak, R. E. Deciding combinations of theories. *J. ACM 31*, 1 (1984), 1–12.

[71] Smans, J., Jacobs, B., and Piessens, F. Implicit dynamic frames. *ACM Trans. Program. Lang. Syst. 34*, 1 (2012), 2:1–2:58.

[72] Suter, P., Dotta, M., and Kuncak, V. Decision procedures for algebraic data types with abstractions. In *POPL'10* (2010), ACM, pp. 199–210.

[73] Suter, P., Köksal, A. S., and Kuncak, V. Satisfiability modulo recursive programs. In *SAS'11* (2011), vol. 6887 of *LNCS*, Springer, pp. 298–315.

[74] Thatcher, J. W., and Wright, J. B. Generalized finite automata theory with an application to a decision problem of second-order logic. *Mathematical Systems Theory 2*, 1 (1968), 57–81.

[75] Thomas, W. Languages, automata, and logic. In *Handbook of Formal Languages*. Springer, 1997, pp. 389–456.

[76] Yang, H. *Local Reasoning for Stateful Programs.* PhD thesis, University of Illinois at Urbana-Champaign, 2001.

[77] Yang, H., Lee, O., Berdine, J., Calcagno, C., Cook, B., Distefano, D., and O'Hearn, P. W. Scalable shape analysis for systems code. In *CAV'08* (2008), vol. 5123 of *LNCS*, Springer, pp. 385–398.

[78] Yang, J., and Hawblitzel, C. Safe to the last instruction: automated verification of a type-safe operating system. In *PLDI'10* (2010), ACM, pp. 99–110.

[79] Yorsh, G., Rabinovich, A. M., Sagiv, M., Meyer, A., and Bouajjani, A. A logic of reachable patterns in linked data-structures. In *FoSSaCS'06* (2006), vol. 3921 of *LNCS*, Springer, pp. 94–110.

[80] Yorsh, G., Reps, T. W., and Sagiv, M. Symbolically computing most-precise abstract operations for shape analysis. In *TACAS'04* (2004), vol. 2988 of *LNCS*, Springer, pp. 530–545.

[81] Zee, K., Kuncak, V., and Rinard, M. C. Full functional verification of linked data structures. In *PLDI'08* (2008), ACM, pp. 349–361.

[82] Zee, K., Kuncak, V., and Rinard, M. C. An integrated proof language for imperative programs. In *PLDI'09* (2009), ACM, pp. 338–351.