Static Checkers For Tree Structures and Heaps

Final Year Project Report

Matthew Hague <mh500@doc.ic.ac.uk>
Supervisors: Dr. Philippa Gardner and Dr. Cristiano Calcagno

Second Marker: Dr. Nobuko Yoshida

Department of Computing
Imperial College London

June 24, 2004

Abstract

Spatial Logics are used to reason about data structures and hierarchical network structures. Automated decision procedures for these logics allow us to formally verify the properties of a system, highlighting errors in the system before it is released. A specific instance of Spatial Logics is the Tree Logic, which describes the structural properties of semi-structured data, such as XML. Previous decision procedures for this logic suffered a complexity bound by a tower of exponentials, meaning that an implementation of the procedure did not run in viable times. Recently, Dal Zilio et al have proposed a new decision procedure for the Tree Logic, with a complexity that is doubly exponential. We provide the first implementation of this approach, and discover, through the use of several optimisations, that the decision procedure is viable. Furthermore, we take this decision procedure as an inspiration and produce two new decision procedures for the Separation Logic — a logic for reasoning about memory heap structures with a stack and pointers. The first decision procedure involves a translation from the Separation Logic to the Tree Logic, whilst the second procedure shows that the Separation Logic can be expressed in First-Order Logic with Equality.

Project Archive

This report and the full source code of the tool produced, is available at,

http://www.doc.ic.ac.uk/~mh500/finalyearproject/

Acknowledgements

I would like to thank my supervisors, Philippa Gardner and Cristiano Calcagno, for their support and direction throughout this project: both challenging me and rescuing me as the situation required.

I would also like to thank my peers at DoC for their help, the “friendly” competition, and their various talents, which I try to emulate to the best of my ability.

Contents

1 Introduction **11**

1.1 Heap-Like Structures 12

1.2 Tree-Like Structures 13

1.3 Contribution 13

1.4 Report Structure 14

2 Background: Tree Logics **16**

2.1 Trees 16

2.2 Tree Logic 18

2.3 A First Attempt 21

2.3.1 Complexity 22

2.4 A Second Attempt Using Sheaves 22

2.4.1 Sheaves 22

2.4.2 Sheaves Logic 23

2.4.3 Presburger Constraints 23

2.4.4 Bases 24

2.4.5 Encoding Tree Logic Using Sheaves Logic 25

2.4.6 A Method for Building a Common Basis from Heterogeneous Supports 27

2.4.7 Tree Automata . . . 29
2.4.8 Constructing Automata for the Sheaves Logic . . . 32
2.4.9 Complexity . . . 33
2.4.10 Recursive Sheaves Logic . . . 34
2.4.11 The Kleene Star . . . 35
2.4.12 Summary . . . 36

3 Background: Presburger Constraint Solvers 37

3.1 LASH . . . 37
3.2 The Omega Library . . . 38
3.3 CVC . . . 40

4 Implementation: Choosing the Approach 41

4.1 Validation and Satisfaction of Tree Logic Formulae . . . 41
4.1.1 Satisfaction/Validity . . . 41
4.1.2 Sheaves Logic/Recursive Sheaves Logic . . . 42
4.2 The Constraint Solver . . . 43
4.3 The Implementation Language . . . 43

5 Implementation: Design and Algorithms 44

5.1 Design Overview . . . 44
5.2 User Interface . . . 46
5.2.1 Type of Interface . . . 46
5.2.2 Grammar . . . 46
5.3 Translating Tree Logic to Sheaves Logic . . . 48
5.3.1 The Placement Modality . . . 48
5.3.2 Binary Connectives . . . 48
5.3.3 Finding a Common Basis . . . 50
5.3.4 Redefining Formulae Over a Basis . . . 50

5.4 Translating Sheaves Logic to Recursive Sheaves Logic 50
5.4.1 Complexity 51
5.5 Constructing the Automaton 51
5.6 Testing an Automaton for Emptiness 52
5.7 Translating Tree Logic to Recursive Sheaves Logic 52

6 Implementation: Optimisations 53

6.1 Extending the Sheaves Logic Syntax 53
6.2 Optimising Basis Construction 55
6.3 Reducing the Number of Quantified Variables 55
6.4 Unsatisfiable Presburger Formulae 56
6.5 Testing Automata for Emptiness 57

7 Implementation: Evaluation 58

7.1 Testing 58
7.2 Width and Depth 59
7.2.1 Width 60
7.2.2 Depth 61
7.3 Viability 61
7.4 Optimisations 63
7.4.1 Extended Sheaves Logic 63
7.4.2 Basis Optimisation 63
7.4.3 Quantified Variable Reduction 64
7.4.4 Unsatisfiable Presburger Constraints 64
7.4.5 The Test for Emptiness 64
7.4.6 Further Optimisations 65
7.5 Interface 66
7.6 Summary 66

8 Theory: Two New Decision Procedures for the Separation Logic 68

8.1 Separation Logic 69

8.1.1 Size 69

8.1.2 Finite States 71

8.2 Translating Separation Logic to Tree Logic 72

8.2.1 Translating a Heap into a Tree 73

8.2.2 Translating Separation Logic to Tree Logic 74

8.2.3 Complexity of the Translation 76

8.3 Translating Separation Logic to $FOL_=$ 77

8.3.1 Translating States to Sheaves 77

8.3.2 Translating Separation Logic to a Sheaves Logic 78

8.3.3 Translating Separation Sheaves Logic to $FOL_=$ 82

8.3.4 Complexity of the Translation 82

8.4 A Classical Fragment of the Separation Logic 83

8.4.1 CL 83

8.4.2 Intensional Equivalence 84

8.4.3 Characteristic Formulae 84

8.4.4 Expressing the Separation Logic in CL 85

8.5 Comparison with CL 85

8.6 Evaluation 86

9 Conclusions and Future Work 88

9.1 Conclusions 88

9.2 Future Work 89

A Implementation 93

A.1 Translations 93

A.1.1 Translation of the Placement Modality 93

A.1.2 Translating Sheaves Logic to Recursive Sheaves Logic . . 94
A.2 Optimisations 95
A.2.1 Extended Sheaves Logic Direct Translations 95
A.2.2 Translating Extended Sheaves Logic to Recursive Sheaves Logic 97
A.2.3 Basis Optimisation 98
A.3 Testing 99

B Theory **103**
B.1 Deciding the Separation Logic 103
B.1.1 Composition Adjunct ($-\!*$) 103
B.1.2 Reducing Heaps 104
B.1.3 A Finite Set of Heaps 104
B.1.4 A Finite Set Of Stacks 105
B.1.5 $H'_{\phi,n}$ 105
B.2 Translating Separation Logic to Tree Logic 105
B.3 Translating Separation Logic to $FOL_=$ 110
B.3.1 Translating Separation Logic to Presburger Arithmetic . . 110

C User Guide **114**
C.1 Trees 114
C.2 Tree Logic 114
C.3 Using the Tool 116
C.4 Examples 117
C.4.1 Types 117
C.4.2 Sub-types 119
C.4.3 Validity 119

List of Tables

2.1 The grammar for representing trees 16
2.2 Structural equivalence for trees 17
2.3 The Tree Logic syntax 18
2.4 Satisfaction of Tree Logic formulae 19
2.5 The Sheaves Logic syntax 23
2.6 The syntax of Presburger Arithmetic 24
2.7 Translation from Tree Logic to Sheaves Logic: base cases 25
2.8 Translation from Tree Logic to Sheaves Logic: positive operators 26
2.9 Translation from Tree Logic to Sheaves Logic: negative operators 26
2.10 The composition of Presburger formulae 27
2.11 The transition relation $\rightarrow$ 30
2.12 Testing an automaton for emptiness 32
2.13 The Recursive Sheaves Logic syntax 34

3.1 An example input file for LASH 38
3.2 The output produced by LASH when passed the file shown in table 3.1 38
3.3 An example session with the Omega Calculator 39
3.4 A sample run of CVC 40

5.1 The input grammar 47
5.2 Encoding placement in Sheaves Logic 48
5.3 Algorithm for translating the placement operator 49
5.4 Algorithm for translating binary connectives 49
5.5 Algorithm for constructing a common basis 50
5.6 A translation from Sheaves Logic to Recursive Sheaves Logic . . 51
5.7 Algorithm for constructing an automaton from an RSL formula . 52

6.1 A translation from the Extended Sheaves Logic to Recursive Sheaves Logic 54

7.1 Run times, additive 59
7.2 Run times, subtractive 59
7.3 The average speed up observed for each optimisation 60

8.1 The syntax of a sub-language of the Separation Logic 70
8.2 The semantics of a sub-language of the Separation Logic, given stack, s and heap, h 70
8.3 The size of a Separation Logic formula 70
8.4 A summary of the notations introduced in section 8.1.2 71
8.5 The function *heaptran* 73
8.6 Encoding a Separation Logic formula, ϕ, in Tree Logic 74
8.7 Definition of $vector_{\phi,p}(s, h)$ 79
8.8 Encoding a Separation Logic formula, ϕ, as a Presburger formula 80
8.9 A classical fragment of the Separation Logic 83
8.10 Classical connectives Separation Logic encoding 83

A.1 Translating Tree Logic to Extended Sheaves Logic 96
A.2 Simple test cases 100

A.3 Simple test cases continued 101
A.4 Moderate test cases 101
A.5 Difficult test cases 102

C.1 The Tree Logic syntax 116
C.2 Satisfaction of Tree Logic formulae 117
C.3 The input grammar 118

List of Figures

2.1 Simple example trees 17

2.2 Simple example Tree Logic formulae and the trees that satisfy them 20

3.1 The value of maxVars against run-time 39

5.1 An overview of the program structure 45

7.1 The width of a formula against the run-times of the tool 60

7.2 The depth of a formula against the run-times of the tool 61

C.1 Constructing trees 115

Chapter 1

Introduction

The ability to reason formally about computer systems is important in the production of both hardware and software. Formal reasoning allows us to prove desirable properties of the system, ensuring that a specification is met and that no unexpected errors will occur due to a bad design. Such formal methods have been embraced by industry because the cost of formal verification can be lower than the cost of repairing an unforeseen error. For example, Intel® now uses formal verification after a floating point error in their Pentium® Processor cost the company a reported $500 million [1, 2].

To be able to automatically verify software, we need to be able to describe the required properties of the software and the data that it manipulates. In this project, we concentrate on logics that can be used to reason about data structures, specifically, we focus on the area of Spatial Logics.

Spatial Logics are concerned with the properties of structured data and resources such as hierarchical network topologies, memory heaps and semi-structured data. Spatial Logics enrich Classical Logic with spatial connectives that describe the construction (and destruction) of data structures. The two most common paths of Spatial Logics are the Separation Logic and the Ambient Logic. Separation Logic [3] is used to reason about low-level data structures and processes that manipulate them; Ambient Logic analyzes the properties of hierarchical structures such as distributed systems. To reason about semi-structured data, such as XML, we use the Tree Logic [4] — the static fragment of the Ambient Logic [5].

Model checkers for these logics have many practical uses. For example, a verification tool that uses a logic for imperative programs based on the Separation Logic can detect memory leaks in imperative programs [6], whilst the Tree Logic has applications in verifying the security properties of firewalls, or in pattern-matching languages for manipulating tree-like data. However, model checking for these logics is a non-trivial task: there are many ways of decomposing a data structure, and some connectives are implicitly quantified over an infinite set of structures.

The first decision procedure for the Tree Logic was introduced by Calcagno et

al [4]. However, this decision procedure required the enumeration of potentially large sets of trees, and its complexity was very high. An implementation of this procedure was attempted as part of an MEng project by Zeeshan Alam [7] and it was found that the complexity was such that the approach could only run in reasonable times for very small examples.

The primary objective of this project is to produce the first implementation of a new method put forward by Dal Zilio et al in [8] for model checking using Tree Logic formulae. This approach has a much lower complexity than the previous method. We were able to further improve the approach through several optimisations and produce a tool that can evaluate moderately complex formulae in reasonable times. We provide an evaluation of the implementation and the viability of the approach in chapter 7. Further to the implementation, in chapter 8 we investigate whether the ideas in Dal Zilio et al's work can provide new decision procedures for the Separation Logic. Firstly we give a translation from the Separation Logic to the Tree Logic, which allows us to extend the tool that was implemented during this project to evaluate Separation Logic assertions. Then we apply some of the ideas behind Dal Zilio et al's procedure to the Separation Logic directly. A result of this work is a new decision procedure that translates the Separation Logic into First-Order Logic with Equality. Recent independent work by Etienne Lozes shows that the Separation Logic can be expressed by a classical fragment of the Separation Logic [9]. This, however, does not provide a decision procedure for the logic. We give a full comparison of these two results in section 8.5.

For the remainder of this chapter we discuss the Separation Logic and the Tree Logic in more depth. A further section details the contributions of this project.

1.1 Heap-Like Structures

Separation Logic is used to describe formally low-level data structures such as a heap model. The heap model describes program states using a heap and a stack. A heap is a flat memory structure that maps locations to values, and similarly, the stack maps variables to values. These values may be other memory locations or *nil*.

Calcagno, Yang and O'Hearn have shown that, in general, this logic is undecidable [10]. That is, it is not possible to decide in finite time whether a heap satisfies a given formula, or whether a formula is valid. However, they have shown decidability for a fragment of the logic, which includes the notion of pointers and equality, but not universal quantification or other operations on data, such as $\leq$ or $+$. This fragment can be used to reason about the structure of the heap but not about the contents of the data itself.

1.2 Tree-Like Structures

We use a tree model when reasoning about semi-structured data such as XML. Tree structures are constructed from labelled branches that are composed horizontally, or stacked vertically, much like an XML document, to form a tree. To analyze the properties of these tree structures we use the Tree Logic, the static fragment of the Ambient Logic.

A initial goal of this project was an implementation of a model-checking tool for the Tree Logic. This tool has several applications. For example, we may use logical formulae to represent a type system for tree structures. A pattern matching language that manipulates tree structures will need to decide which particular pattern a given tree conforms to. Also, at compile time it will be necessary to check whether two patterns overlap. Another application occurs when verifying security properties of network structures: networks are often constructed as a hierarchy, where entry into a sub-network is governed by a firewall. This structure can be represented in the Tree Logic, and so we can use it to specify properties of the network. For example, we may wish to enforce that a set of network nodes are behind a given firewall, that is, the nodes are branches on the firewall node.

In [4], Cristiano Calcagno et al prove the decidability of the Tree Logic and provide a decision procedure. However, their approach has a very high complexity — one that is bounded by a tower of exponentials. This procedure was implemented as part of an MEng project last year, and it was found that the approach can only decide the logic in reasonable time for very small examples.

A different approach was recently put forward by Silvano Dal Zilio et al in [8]. This new method involves translating a formula given in the Tree Logic into a new logic called the Sheaves Logic. Part of a Sheaves Logic formula is a numerical constraint that is a Presburger formula. A large amount of research has been conducted to provide efficient decision procedures for these constraints and an implementation of this approach can take advantage of this work. In chapter 3 we discuss several such constraint solvers. We use Tree Automata to decide the satisfiability of the Sheaves Logic formulae.

Abstracting over the cost of solving the Presburger constraints, the complexity of this approach is doubly exponential in the size of the Tree Logic formula — a significant improvement over a tower of exponentials, which bounded the first decision procedure. On the practical level, our implementation of an optimised version of Dal Zilio et al's approach ran in satisfactory times for formulae of a moderate size.

The Tree Model and its decision procedures are discussed in detail in chapter 2.

1.3 Contribution

The original goal of this project was a prototypical implementation of Dal Zilio et al's decision procedure. The design of the program and the algorithms it uses

are detailed in chapter 5.

This prototype revealed that a naive implementation of Dal Zilio et al's work was not a viable approach: its complexity meant that the size of the formulae that could be evaluated in reasonable times was quite small. Consequently, several optimisations that remove some of the redundancy from Dal Zilio et al's theoretical work were implemented, and significant increases in efficiency were observed. For example, for one set of test data the unoptimised algorithm was abandoned after 30 minutes; when optimisations were activated, the run-time was reduced to under four seconds. The tool produced performs model-checking for the Tree Logic and runs in satisfactory times for reasonably complex formulae. A discussion of the optimisations used by the tool is given in chapter 6. An evaluation of the tool and the optimisations it uses is given in chapter 7.

Further to this implementation, in chapter 8 we broaden the scope of the project to include the Separation Logic. This concerns the potential for Dal Zilio et al's decision procedure for the Tree Logic to provide decision procedures for the Separation Logic.

In section 8.2 we provide a translation of the Separation Logic into the Tree Logic. This means that the tool produced during this project may be extended to provide a tool for deciding whether a heap models an assertion, or whether an assertion is valid.

In section 8.3 we take inspiration from the approach put forward by Dal Zilio et al to produce a new decision procedure for the Separation Logic. The main result of this work is that the decidable fragment of the Separation Logic can be expressed using First-Order Logic with Equality[1]. Recent, independent work by Etienne Lozes shows a related result: that the Separation Logic can be expressed by a classical fragment of the logic that is still a Spatial Logic [9]. The atoms of this fragment are the size of a heap, the contents of an address and the equality of stack variables. In section 8.4 an overview of Lozes' work is provided, and in section 8.5 we give a comparison of the two results. The main points of difference are that our translation into First-Order Logic with Equality provides a decision procedure for the Separation Logic, whereas Lozes' work does not, and that the translation into $FOL_=$ removes the notion of a heap entirely. Finally, in section 8.6 we evaluate the work in chapter 8.

1.4 Report Structure

In chapter 2 we describe the decision procedure proposed by Dal Zilio et al. We discuss three available tools for evaluating Presburger Constraints in chapter 3. In chapters 4, 5 and 6 we describe the implementation and optimisation of the approach, concluding in chapter 7 with an evaluation of the tool produced.

The work on the Separation Logic is given chapter 8. The translation into the Tree Logic is presented in section 8.2 and the translation to $FOL_=$ is given in section 8.3. The work by Etienne Lozes in described in section 8.4. We finish

[1] $FOL_=$ is First Order Logic without relations and with equality. This is essentially the PSPACE-Complete problem, Quantified Boolean Formulae (QBF) [11].

the chapter by comparing our work with Lozes' (section 8.5) and evaluating our work into the Separation Logic (section 8.6).

The conclusions of this project are summarised in chapter 9, along with some suggestions for future extensions to the project.

Chapter 2

Background: Tree Logics

In this chapter we give the notion of a tree and define the Tree Logic for reasoning about tree structures. We present some important properties of the Tree Logic and give a brief overview of the previous decision procedure for the logic. Finally we cover in detail the approach to deciding satisfaction and validity for the logic that was introduced recently by Dal Zilio et al [8]. There are several different methods available in Dal Zilio et al's work, we summarise and compare these methods in chapter 4.

2.1 Trees

In [4], Calcagno, Cardelli and Gordon propose a simple syntax for representing edge-labelled finite trees that can be used to represent semi-structured data. Trees are characterised by their edges, rather than their nodes, and each edge is given a name. The syntax for this representation is shown in table 2.1. Structural equivalence is shown in table 2.2. Some simple examples of trees are shown in figure 2.1.

$d, d' ::=$	Tree
0	Empty tree
$d \mid d'$	Composition
$m[d]$	Edge labelled by name $m \in N$ atop tree d where N is an infinite set of names.

Table 2.1: The grammar for representing trees

$d \equiv d$	(Strut Refl)
$d \equiv d' \Rightarrow d' \equiv d$	(Struct Symm)
$d \equiv d', d' \equiv d'' \Rightarrow d \equiv d''$	(Struct Trans)
$d \equiv d' \Rightarrow d\|d'' \equiv d'\|d''$	(Struct Par)
$d \equiv d' \Rightarrow m[d] \equiv m[d']$	(Struct Amb)
$d\|d' \equiv d'\|d$	(Struct Par Comm)
$(d\|d')\|d'' \equiv d\|(d'\|d'')$	(Struct Par Assoc)
$d\|0 \equiv d$	(Struct Zero Par)

Table 2.2: Structural equivalence for trees

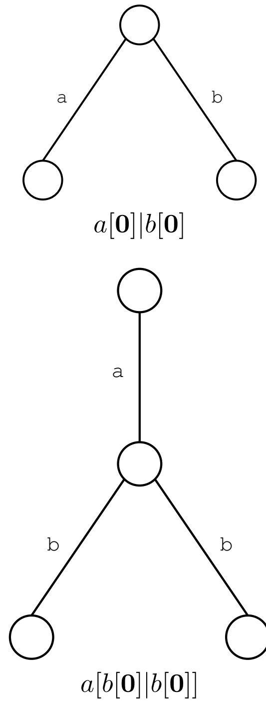


Figure 2.1: Simple example trees

$\mathcal{A}, \mathcal{B} ::=$	Formula
$\mathbf{F}$	False
$\mathcal{A} \wedge \mathcal{B}$	Conjunction
$\mathcal{A} \Rightarrow \mathcal{B}$	Implication
$\mathbf{0}$	Void
$\mathcal{A} \vert \mathcal{B}$	Composition
$\mathcal{A} \triangleright \mathcal{B}$	Guarantee
$n[\mathcal{A}]$	Location
$\mathcal{A}@n$	Placement

Table 2.3: The Tree Logic syntax

2.2 Tree Logic

To describe the properties of the trees described in section 2.1, we use the Tree Logic. This logic was introduced by Cardelli and Gordon in [5]. There are several applications of the Tree Logic, for example:

- Tree Logic formulae can be used in a pattern-matching language for manipulating tree-like structures. A formula can be used to represent the type of a tree. In this case, the validity of a formula can be seen as a compile time check — we may, for example, wish to check that one formula represents a sub-type of another, or that no two patterns can be satisfied by a single tree (leading to ambiguity in the program) — and a test for satisfaction can be seen as a run-time check, testing which pattern a tree adheres to, or whether it is of the correct type.

- The Tree Logic can be used to specify the security properties of a network. Networks are often protected by firewalls, where all traffic to and from a group of nodes must pass through (and be allowed by) the firewall. This forms a hierarchical structure than can be represented by the Tree Logic: sub-networks whose access is controlled by a firewall will appear as subtrees of the firewall branch. We can specify properties such as which nodes are behind which firewalls, and which nodes should not be in a given place in the tree.

The syntax of this logic is given in table 2.2; one can derive connectives such as $\mathbf{T}$, $\neg A$ and $A \vee B$ in the usual way.

We write, $d \models \mathcal{A}$ to denote that the tree d satisfies the formula $\mathcal{A}$, this is akin to saying that tree d is of type $\mathcal{A}$. The complete definition of satisfaction is given in table 2.4

Most of the satisfaction rules given in table 2.4 are straightforward, but some may benefit from an explanation.

$d \models \mathbf{F}$		Never
$d \models \mathcal{A} \wedge \mathcal{B}$	$\triangleq$	$d \models \mathcal{A} \wedge d \models \mathcal{B}$
$d \models \mathcal{A} \Rightarrow \mathcal{B}$	$\triangleq$	$d \models \mathcal{A} \Rightarrow d \models \mathcal{B}$
$d \models \mathbf{0}$	$\triangleq$	$d \equiv \mathbf{0}$
$d \models \mathcal{A}\vert\mathcal{B}$	$\triangleq$	$\exists d', d''.d \equiv d'\vert d'' \wedge d' \models \mathcal{A} \wedge d'' \models \mathcal{B}$
$d \models \mathcal{A} \triangleright \mathcal{B}$	$\triangleq$	$\forall d'.d' \models \mathcal{A} \Rightarrow d'\vert d \models \mathcal{B}$
$d \models n[\mathcal{A}]$	$\triangleq$	$\exists d'.d \equiv n[d'] \wedge d' \models \mathcal{A}$
$d \models \mathcal{A}@n$	$\triangleq$	$n[d] \models \mathcal{A}$

Table 2.4: Satisfaction of Tree Logic formulae

A composite formula ($\mathcal{A}|\mathcal{B}$) is satisfied if the tree can be split into any two parts, such that one part satisfies the formula $\mathcal{A}$ and the other satisfies the formula $\mathcal{B}$. The tree can be split in any way, for example $A|B|C|D$ may be split into the two trees, $A|C$ and $D|B$; it can also be split into the trees $A|B|C|D$ and $\mathbf{0}$.

A tree satisfies a formula of the form $\mathcal{A} \triangleright \mathcal{B}$ iff, when it is composed with any tree which satisfies the formula $\mathcal{A}$, the resulting tree will satisfy the formula $\mathcal{B}$.

Some simple example formulae and the kinds of tree that satisfy them are given in figure 2.2.

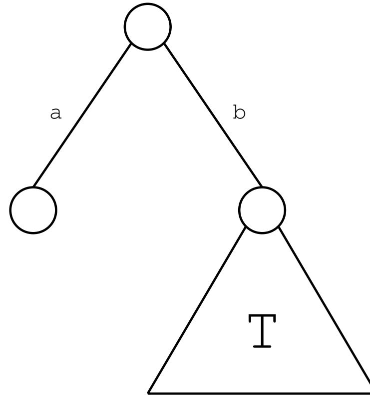
Validity

We say that a formula, $\mathcal{A}$, is valid iff it is true of all trees, that is $\forall d.d \models \mathcal{A}$. Validity can be thought of as a compile-time check. For example, we may want to check that type $\mathcal{A}$ is a subtype of type $\mathcal{B}$. Conversely, we can think of satisfaction as a run-time check that a tree is of a required type.

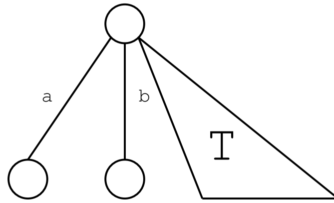
In the Tree Logic, the validity and satisfaction problems are mutually expressible; that is, satisfaction can be reduced to a validity problem and vice versa. The two problems can be reduced to each other as follows:

- **Validity to satisfaction** — A formula, $\mathcal{A}$, is valid iff $\mathbf{0} \models \mathbf{T} \triangleright \mathcal{A}$. This says that, any tree which satisfies truth (all trees), when composed with the empty tree, will satisfy $\mathcal{A}$. We know that any tree composed with the empty tree is structurally congruent to the tree on its own, therefore we are checking that every tree satisfies $\mathcal{A}$.

- **Satisfaction to validity** — Each tree can be directly translated into an equivalent formula. This can be seen by comparing the syntax of trees with the syntax of formulae; for example, the tree $a[\mathbf{0}]|b[\mathbf{0}]$ translates directly to the formula $a[\mathbf{0}]|b[\mathbf{0}]$. We denote the translation of tree d as $\underline{d}$, and we can check the whether tree d satisfies a formula $\mathcal{A}$ by testing the validity of the formula $\underline{d} \Rightarrow \mathcal{A}$.



$a[\mathbf{0}]|b[\mathbf{T}]$ — That is, any tree with two top branches, labelled a and b. The a branch must be atop the empty tree, whereas the b branch may be atop any tree. (All trees satisfy truth.)



$(a[\mathbf{0}]|\mathbf{T}) \wedge (b[\mathbf{0}]|\mathbf{T})$ — That is, any tree with an a branch, atop the empty tree, composed with any tree, and a b branch, atop the empty tree, composed with any tree. The tree has an a branch and a b branch, both atop the empty tree, composed with any tree.

Figure 2.2: Simple example Tree Logic formulae and the trees that satisfy them

Decidability

To test whether a tree satisfies a given formula, we can use a model checking approach. This approach, as we have shown, is also effective in deciding validity. However, to test satisfaction for the guarantee connective we need to evaluate an implicit "for all trees" quantifier, which ranges over the set of all trees, which is an infinite set[1]. Alternatively, we can reduce satisfaction to a validity problem. However, this approach will always require universal quantification (as we must check the property for all trees). The quantification problem occurs in both cases.

2.3 A First Attempt

In [4], Calcagno et al show that the Tree Logic[2] is decidable. This was shown using two observations: the number of label names to consider can be reduced to a finite set, and the number of trees to check when evaluating the guarantee connective or determining validity can be restricted to a finite set.

To restrict the set of names to a finite set Calcagno et al observed that Tree Logic formulae can express two things about branches: either a branch should have a given name, or it should not.

For example, the formula $a[\mathbf{0}]$ expresses a tree constructed from a single branch, named a. Branches can be divided into two sets: those labelled a, and those not labelled a. The name of the branches that are not labelled a are irrelevant in that we can change their name to another name that is not a without changing the satisfaction of the formula. This argument can be similarly applied to the formula, $\neg a[\mathbf{0}]$.

It is possible to extend this argument to formulae that mention any number of names. The result is that we can limit the set of names to all the identified names in a formula, plus one that is not mentioned. We can transform an arbitrary tree to use this restricted set of names by renaming any branches that use labels that are not mentioned in the formula to the additional unmentioned name, without changing the satisfaction result.

We still need to consider an infinite number of trees, even if we have a limited set of branch names. To address this problem, Calcagno et al proposed a notion of size; each tree can be given a depth and a multiplicity (the maximum number of times any branch label appears at any node) that constitute the size of the tree. Similarly, we can give formulae a size — to test the validity of a formula we need only check the trees whose size is less than or equal to the size of the formula.

For example, the formula, $a[\mathbf{0}]$ cannot distinguish between the tree, $a[b[\mathbf{0}]]$ and the tree, $a[b[b[\mathbf{0}]]]$. Similarly, the formula, $a[\mathbf{0}]$ cannot differentiate between the tree, $a[\mathbf{0}]|a[\mathbf{0}]$ and the tree, $a[\mathbf{0}]|a[\mathbf{0}]|a[\mathbf{0}]$. We can say that this formula has

[1] The quantification present in the semantics of the guarantee connective is separate from the quantification present in the Tree Logic with quantifiers (of names). The Tree Logic with quantifiers has been shown to be undecidable [12] and so it is not considered in this report.

[2] Without the quantification of names.

a depth of two and a multiplicity of two, where the depth refers to the maximum number of branches stacked on top of each other anywhere in the tree, and the multiplicity is the maximum number of occurrences of a particular branch label at any node in the tree.

We can use these observations to construct a finite set of trees that we need to check when testing whether a guarantee formula is satisfied. This approach uses similar sets to test the existential quantifiers in the semantics of composition and location, except that these sets are simpler to construct as they are built from decompositions of the tree being tested.

2.3.1 Complexity

Unfortunately, the approach detailed above has a complexity that is worse than a tower of exponentials. This is because the set of trees used to evaluate the guarantee operator is very large: for each node of a tree, we need to consider all combinations of the available names, up to the required multiplicity and the number of nodes depends on the number of names, the multiplicity, and the depth.

An implementation of this approach was attempted as part of a final year MEng project [7]. Unfortunately, because of the high complexity, the implementation could only reason about trees up to a depth of one in reasonable time.

2.4 A Second Attempt Using Sheaves

In [8], Dal Zilio, Lugiez, and Meyssonnier introduced a new approach to the tree logic problem. The fundamental difference in their approach is one of notation; this change in notation allows a new way of tackling the problem, which is only doubly exponential, rather than bounded by a tower of exponentials.

Dal Zilio et al introduce the Sheaves Logic, which is based on this new notation and provide a translation from the Tree Logic to the new logic. An automaton method for deciding whether a tree satisfies a formula specified in this sheaves logic is given as well as an algorithm for determining the emptiness of the automaton (satisfiability of the Tree logic formula); finally, a recursive version of the sheaves logic is introduced.

2.4.1 Sheaves

If, for example, we have a tree of the form $A|A|B|C|A|B$, the sheaf notation of this tree is the dot product of two vectors; the first denotes the multiplicity, and the second denotes the sub-trees. In this particular case, the sheaf is as follows: $(3, 2, 1) \cdot (A, B, C)$. That is, three As, two Bs and one C.

α ::=	Label expression
$a_1, \ldots, a_n$	Finite subset of N
$\alpha^\perp$	Complement of α
E ::=	Element formula
$\alpha[A]$	Element with label in α
A ::=	Counting formulae
$\top$	True
$\exists \mathbf{N}.\phi(\mathbf{N}) \cdot \mathbf{E}$	Sheaves composition (with $\lvert\mathbf{N}\rvert = \lvert\mathbf{E}\rvert$)

Table 2.5: The Sheaves Logic syntax

2.4.2 Sheaves Logic

The syntax for the sheaves logic is given in table 2.5. This logic assumes a set of branch labels, denoted N, and the set, $\mathcal{E}$, of elements (an element is of the form $a[d]$, where a is a label and d is a tree).

There are several differences between this and the tree logic presented in section 2.2. The first difference is the use of label expressions on elements, rather than single labels. A label expression is either a finite set $\alpha \subseteq N$ or a co-finite set $\alpha^\perp$, we often write α to denote a label expression of either form. The notation $\alpha^\perp$ describes any element that is not in the finite set α. A label expression α denotes any label a such that $a \in \alpha$. For example, the semi-formula $\{a, b\}[\mathbf{0}]$ matches either one of the two trees, $a[\mathbf{0}]$ or $b[\mathbf{0}]$. (Note: $\mathbf{0}$ is not valid sheaves logic syntax, rather an abbreviation of a more complex formula that is given in section 2.4.5.)

The set $\alpha \subseteq N$ must be finite to allow us to test, in finite time, whether a name, a is in the set α. In case $\alpha^\perp$, if α is finite, we can determine if a label, a, is not in α in finite time.

The second and main difference is the $\exists \mathbf{N}.\phi(\mathbf{N}) \cdot \mathbf{E}$ construct. This formula describes a sheaf of the form $\mathbf{N} \cdot \mathbf{E}$, where $\mathbf{N}$ is a vector, $(n_1, \ldots, n_p)$, satisfying the numerical constraint ϕ and $\mathbf{E}$ is a vector, $(\alpha_1[A_1], \ldots, \alpha_p[A_p])$, of element formulae. $\mathbf{E}$ is called the support vector. The sheaf then denotes a tree that, for each $i \in 1..p$, has n_i branches (at the top level) whose label is in the set α_i and whose sub-tree satisfies the formula A_i. A tree satisfies the formula $\exists \mathbf{N}.\phi(\mathbf{N}) \cdot \mathbf{E}$ if it can be described over the support vector $\mathbf{E}$ by any vector $\mathbf{N}$ such that $\mathbf{N}$ satisfies the constraint, ϕ.

2.4.3 Presburger Constraints

The formula $\exists \mathbf{N}.\phi(\mathbf{N}) \cdot \mathbf{E}$ contains a numerical constraint, ϕ, on the vector $\mathbf{N}$. This constraint is expressed as a Presburger formula. Presburger formulae act on the set of natural numbers (non-negative integers) and denote constraints upon them. They are a decidable subset of First-Order Logic, and a significant

$Exp ::=$	Integer expression
n	Non-negative integer constant
N	Non-negative integer variable
$Exp_1 + Exp_2$	Addition
$\phi, \psi, \ldots ::=$	Presburger arithmetic formulae
$(Exp_1 = Exp_2)$	Test for equality
$\neg\phi$	Negation
$\phi \vee \psi$	Disjunction
$\exists N.\phi$	Existential quantification

Table 2.6: The syntax of Presburger Arithmetic

amount of research has been conducted into producing efficient tools for solving them. The syntax is given in table 2.6.

We can define a large number of integer properties using these formulae, and we will often use abbreviations. For example, $\wedge$ may be defined using De Morgan and "M strictly greater than N" can be described using the formula, $\exists X.(M = N + X + 1)$.

We write $\phi(\mathbf{N})$ to denote a Presburger formula with its free variables in $\mathbf{N} = (n_1, \ldots, n_p)$. Often, when writing sheaves logic formulae, we omit the argument, $\mathbf{N}$, to the Presburger constraint and simply write $\exists\mathbf{N}.\phi \cdot \mathbf{E}$. The vector is implicitly passed as an argument.

When testing whether a tree satisfies $\exists\mathbf{N}.\phi(\mathbf{N}) \cdot \mathbf{E}$, we need to check that it can be expressed in the form $\mathbf{N} \cdot \mathbf{E}$, where $\mathbf{N}$ is a vector of integers, and $\phi(\mathbf{N})$ holds.

For example, if we want to express the tree logic formula, $a[\mathbf{0}]|b[\mathbf{0}]|b[\mathbf{0}]$ in sheaves logic, we could write, $\exists(n,m)(n = 1 \wedge m = 2) \cdot (\{a\}[\mathbf{0}], \{b\}[\mathbf{0}])$. However, if we wanted to express the tree logic formula $a[\mathbf{0}]|b[\mathbf{0}]|b[\neg\mathbf{0}]$, we would require three element formulae in the support vector, $\mathbf{E}$: $\{a\}[\mathbf{0}]$, $\{b\}[\mathbf{0}]$ and $\{b\}[\neg\mathbf{0}]$.

It is also worth noting that support vectors can be more specific than the formulae that they are representing. For example, the formula $a[\mathbf{T}]$ can be expressed as $\exists(n,m)(n + m = 1) \cdot (\{a\}[A], \{a\}[\neg A])$, where A is any formula. This formula represents $a[\mathbf{T}]$ because it describes a tree of the form $a[A]$ or $a[\neg A]$, which is equivalent to a tree of the form $a[\mathbf{T}]$.

2.4.4 Bases

We now give a definition of a basis, which is a useful property of a support vector $\mathbf{E}$. Intuitively, if a support vector is a basis, then it can be used to describe any tree. For example, the support vector $(a[\mathbf{0}])$ can only describe a tree formed from any number of $a[\mathbf{0}]$ elements. It is not possible, using this support, to denote a tree that has a branch that is not labelled a. We could address this

$$
\begin{array}{rcl}
AnyE & =_{def} & \emptyset^{\perp}[\top] \\
\\
\mathbf{0} & =_{def} & \exists(n).(n = 0) \cdot AnyE \\
\mathbf{T} & =_{def} & \exists(n).(n \geq 0) \cdot AnyE \\
\alpha[A] & =_{def} & \exists(n_1, n_2, n_3).(n_1 = 1 \wedge n_2 = 0 \wedge n_3 = 0) \cdot (\alpha[A], \alpha[\neg A], \alpha^{\perp}[\top])
\end{array}
$$

Table 2.7: Translation from Tree Logic to Sheaves Logic: base cases

problem by adding the element formula $\{a\}^{\perp}[\mathbf{T}]$ to the support. However, using this new support it is not possible to denote a branch a atop any tree except the empty tree. To complete the basis we need to add the element formula $a[\neg\mathbf{0}]$.

The basis property is required when constructing a Sheaves Logic formulae that is equivalent to a given Tree Logic formula. The translation, given in section 2.4.5, is defined inductively, so, for example, if we had the formula $\neg A$ we would first construct a Sheaves Logic formula that represents A. If the support vector generated by this translation only allowed us to denote trees that matched A, we would not be able to denote those trees that satisfy $\neg A$ using the same support vector, $\mathbf{E}$.

We write $[\![E]\!]$ to denote the set of all trees that satisfy the element formula, E. Element formulae have the form $\alpha[A]$, therefore, the trees in $[\![E]\!]$ will have a single branch (in α) from the root node, atop a tree that satisfies the formula A.

A vector $(E_1, \ldots, E_n)$ is a basis iff $i \neq j$ implies $[\![E_i]\!] \cap [\![E_j]\!] = \emptyset$ for all $i, j \in 1..n$ and $\bigcup_{i=1}^{n} [\![E_i]\!] = \mathcal{E}$, where $\mathcal{E}$ is the set of all elements. That is, the element formulae of the support vector define disjoint trees, and together, they cover all possible trees of the form $a[d]$, where d is a tree and a is a branch. A basis, $\mathbf{E}$, is proper iff every support vector appearing in a sub-formula of $\mathbf{E}$ is also a basis.

2.4.5 Encoding Tree Logic Using Sheaves Logic

Dal Zilio et al's translation from tree logic to sheaves logic is as follows. $AnyE$ is the element formula $\emptyset^{\perp}[\top]$ — that is, a branch with any label, atop any tree. The translation is inductive and the base cases are shown in table 2.7.

It is worth noting that a simpler translation of $\alpha[A]$ is $\exists(n).(n = 1) \cdot \alpha[A]$, however, $\alpha[A]$ does not define a basis. The basis property is desirable when translating a formula because it allows us to construct formulae such as $\neg A$ inductively, as described in the previous section.

When translating formulae, such as composition, with two sub-formulae, we require that the sub-formulae are defined over a common support vector. This is because the larger formulae will define trees that have some combination of the properties expressed by each of the sub-formulae. If the sub-formulae are

Assume $A = \exists \mathbf{N}.\phi_A.\mathbf{E}$ and $B = \exists \mathbf{N}.\phi_B.\mathbf{E}$

$$\begin{array}{rcl} A \vee B & =_{def} & \exists \mathbf{N}.(\phi_A \vee \phi_B)(\mathbf{N}) \cdot \mathbf{E} \\ A|B & =_{def} & \exists \mathbf{N}.(\phi_A + \phi_B)(\mathbf{N}) \cdot \mathbf{E} \end{array}$$

The connectives $\vee$ and $+$ for Presburger constraints are defined in table 2.10

Table 2.8: Translation from Tree Logic to Sheaves Logic: positive operators

Assume $A = \exists \mathbf{N}.\phi_A.\mathbf{E}$ and $B = \exists \mathbf{N}.\phi_B.\mathbf{E}$

$$\begin{array}{rcl} \neg A & =_{def} & \exists \mathbf{N}.(\neg \phi_A)(\mathbf{N}) \cdot \mathbf{E} \\ A \wedge B & =_{def} & \exists \mathbf{N}.(\phi_A \wedge \phi_B)(\mathbf{N}) \cdot \mathbf{E} \\ A \triangleright B & =_{def} & \exists \mathbf{N}.(\phi_A \triangleright \phi_B)(\mathbf{N}) \cdot \mathbf{E} \end{array}$$

The connectives $\wedge$ and $\triangleright$ for Presburger constraints are defined in table 2.10

Table 2.9: Translation from Tree Logic to Sheaves Logic: negative operators

$$
\begin{array}{rcl}
(\phi \wedge \psi)(\mathbf{N}) & =_{def} & \phi(\mathbf{N}) \wedge \psi(\mathbf{N}) \\
(\phi \vee \psi)(\mathbf{N}) & =_{def} & \phi(\mathbf{N}) \vee \psi(\mathbf{N}) \\
(\phi + \psi)(\mathbf{N}) & =_{def} & \exists \mathbf{N_1}, \mathbf{N_2}.((\mathbf{N} = \mathbf{N_1} + \mathbf{N_2}) \wedge \phi(\mathbf{N_1}) \wedge \psi(\mathbf{N_2})) \\
(\phi \triangleright \psi)(\mathbf{N}) & =_{def} & \forall \mathbf{M}.(\phi(\mathbf{M}) \Rightarrow \psi(\mathbf{N} + \mathbf{M}))
\end{array}
$$

Table 2.10: The composition of Presburger formulae

defined over the same support vector then that support vector will be adequate to express the combination of the two formulae. If these two sub-formulae are not defined over a common basis, but are each defined over a basis, we are able to redefine the formulae so that they share the same basis. This procedure is explained in section 2.4.6.

Additionally, negative operators, such as $\neg$ and $\triangleright$ require that the common support vector is also a basis. This is because both of these operators implicitly refer to the set of all trees. The semantics of $\triangleright$ contain universal quantification — for all trees — and so we must be able to express all trees using the support vector. Similarly, $\neg A$ is satisfied by any tree that does not satisfy A.

The positive composition operators are given in table 2.8, the translations for negation and composition adjunct ($\triangleright$) are given in table 2.9 and the composition of Presburger formulae is given in table 2.10. The methods for constructing a common basis and redefining a formula over a different basis are given in section 2.4.6.

To give an idea of the benefits of this approach, we run through the validity test $\mathbf{0} \models \mathbf{T} \triangleright \mathcal{A}$. This formula translates to a formula of the form $(\top \triangleright \phi_A)(\mathbf{0})$, which expands to $\forall \mathbf{M}.(\top(\mathbf{M}) \Rightarrow \phi_A(\mathbf{0} + \mathbf{M}))$, or $\forall \mathbf{M}.\phi_A(\mathbf{M})$. This approach has pushed the infinite quantification into the Presburger constraint. Constraint solvers, such as those discussed in chapter 3 are able to solve constraints with such quantification.

2.4.6 A Method for Building a Common Basis from Heterogeneous Supports

In this section we describe the method for ensuring that two formulae are defined over a common basis. The translation from Tree Logic to Sheaves Logic is inductive, and binary operators require that the translations of their sub-formulae are defined over a common basis. We begin by introducing the notion of refinement of bases. We then explain how a formula can be re-expressed over a refining basis. Finally, we detail the method for constructing a (common) basis that refines two bases.

Refinements

We say that a support vector $\mathbf{F}$ refines the support vector $\mathbf{E}$ if $\mathbf{F}$ represents a more precise decomposition of the trees expressible over $\mathbf{E}$. For example, the support $(a[A], a[\neg A])$ is a refinement of the support $(a[\top])$ since they both express all trees that are a branch a atop any sub-tree, but the first support allows us to distinguish between those sub-trees that satisfy a given formula and those that don't.

Formally, a support $\mathbf{F} = (F_1, \ldots, F_q)$ is a refinement of the support $\mathbf{E} = (E_1, \ldots, E_p)$ if there is a relation $\mathcal{R}$ of $1..p \times 1..q$ such that for all $i \in 1..p$ we have $[\![E_i]\!] = \bigcup_{(i,j)\in\mathcal{R}} [\![F_j]\!]$.

Redefining a Formula Over a Refining Support

If we have a Presburger constraint, $\phi(\mathbf{N})$, defined over the support vector $\mathbf{E} = (E_1, \ldots, E_p)$, we construct the constraint $\phi_{\mathcal{R}}(\mathbf{M})$ defined over the refining support vector $\mathbf{F} = (F_1, \ldots, F_q)$ such that $[\![\exists \mathbf{N}.\phi \cdot \mathbf{E}]\!] = [\![\exists \mathbf{M}.\phi_{\mathcal{R}} \cdot \mathbf{F}]\!]$ where $\mathbf{N} = (N_1, \ldots, N_p)$, $\mathbf{M} = (M_1, \ldots, M_q)$, and $\phi_{\mathcal{R}}(\mathbf{M})$ is the constraint:

$$\exists (N_i)_{i\in 1..p}, (X^i_j)_{(i,j)\in\mathcal{R}}. \left(\begin{array}{l} \bigwedge_{j\in 1..q}(M_j = \sum_{(i,j)\in\mathcal{R}} X^i_j) \\ \wedge \bigwedge_{i\in 1..p}(N_i = \sum_{(i,j)\in\mathcal{R}} X^i_j) \\ \wedge\ \phi(\mathbf{N}) \end{array} \right)$$

To understand why this formula, expressed over $\mathbf{F}$, is equivalent to ϕ expressed over $\mathbf{E}$ we can think of the variables X^i_j as mappings from the element formulae of $\mathbf{F}$ to the element formulae of $\mathbf{E}$. The element formula, F_j, may describe elements from several element formulae, E_i, and vice versa. The variables, X^i_j, denote the decomposition (and recomposition) of the elements described by the basis $\mathbf{F}$ into the basis $\mathbf{E}$. If there is a decomposition such that ϕ holds, then the formula is satisfied.

Building a Common Basis

Given the proper bases $\mathbf{E} = (E_1, \ldots, E_p)$ and $\mathbf{F} = (F_1, \ldots, F_q)$, we can construct a common proper basis, denoted $\mathbf{E} \times \mathbf{F}$ by building the support vector $\mathbf{G}$ containing the element formulae (G_{ij}) for all $(i, j) \in 1..p \times 1..q$ where the set of trees accepted by G_{ij} is the intersection of the set of trees accepted by E_i and F_j.

It can be seen that such a basis will refine both $\mathbf{E}$ and $\mathbf{F}$. This is because both $\mathbf{E}$ and $\mathbf{F}$ are proper bases, and so define the complete set of elements, $\mathcal{E}$. For example, $[\![E_i]\!] = \bigcup_{j\in 1..q} [\![G_{ij}]\!] = \bigcup_{j\in 1..q} [\![E_i]\!] \cap [\![F_j]\!]$ because $\mathbf{F}$ is a basis, and so $\bigcup_{j\in 1..q} [\![F_j]\!] = \mathcal{E}$.

When constructing the common basis we need to construct the element formula G_{ij} that accepts the same set of trees as the intersection of the element formulae E_i and F_j. Let $E_i = \alpha_i[A_i]$ and $F_j = \beta_j[B_j]$. There are two cases to consider:

- Case $A_i = \top$. In this case $G_{ij} = (\alpha_i \cap \beta_j)[B_j]$.
- Otherwise $G_{ij} = (\alpha_i \cap \beta_j)[A_i \wedge B_j]$. To be able to construct this element formula we have to build the formula $A_i \wedge B_j$. This requires that A_i and B_i are defined over a common basis. We ensure this property by recursively redefining A_i and B_j so that they are defined over a common basis. It is because of the recursive nature of the algorithm that the original bases must be proper — that is, every support vector appearing in a sub-formula of the basis is also a basis.

2.4.7 Tree Automata

To evaluate a Sheaves Logic formula we translate the formula into an equivalent automaton. We can determine satisfiability and satisfaction using the resulting automaton.

A tree automaton has a set of states, a set of final states and a set of rules. We denote an automaton, $\mathcal{A} = \langle \mathcal{Q}, \mathcal{Q}_{fin}, \mathcal{R} \rangle$, where $\mathcal{Q}$ is the set of states, $\mathcal{Q}_{fin}$ is the set of final states included in $\mathcal{Q}$, and $\mathcal{R}$ is the set of rules. The automaton takes a tree as input reduces its sub-trees to states, starting from the leaf nodes. If the automaton can use its rules to reduce the tree to a single state that is a final state, then the automaton is said to accept the tree.

The transition relation of an automaton is the transitive closure of the relation defined by the rules in table 2.11. There are two types of rule:

- Type 1 rules operate vertically on the tree and are analogous to an element formula (of the form $\alpha[A]$). They are written $\alpha[q'] \to q$. The state q' is the state that a sub-tree that satisfies the sub-formula A can be reduced to, and α is the same α in the element formula. When this rule is applied, we reduce a branch atop a state to a single state, which indicates that an element formula has been matched.
- Type 2 rules operate horizontally on the tree and are written $\phi \to q$, where ϕ is a Presburger constraint that takes as its arguments the number of occurrences of each state at the current level of the tree. They are analogous to the Presburger constraints in the Sheaves Logic since they count the number of states at a node of the tree, which represent the element formula that have been matched (by type 1 rules). Type 2 rules count, using a Presburger constraint, the element formulae that have been matched, and reduce them to a single state if the conditions of the formula have been met.

There a several notations used in table 2.11 and in general when referring to these automaton. $e_1|\dots|e_n$ is a tree built from the composition of several element formulae. $q_1|\dots|q_n$ is the result of reducing each of the element formulae to a state. #q is the number of occurrences of state q in a term $q_1|\dots|q_n$, and $\#\mathcal{Q}(q_{j_1}|\dots|q_{j_n})$ denotes the multiplicities of each of the states in $q_{j_1}|\dots|q_{j_n}$. We write $\#\mathcal{Q}(q_{j_1}|\dots|q_{j_n}) \in [\![\phi]\!]$ to denote that ϕ holds when the multiplicities of the states in the term $q_{j_1}|\dots|q_{j_n}$ are passed as arguments.

(type 1)

$$\frac{d \to q' \quad \alpha[q'] \to q \in \mathcal{R} \quad a \in \alpha}{a[d] \to q}$$

(type 2)

$$\frac{e_1 \to q_{j_1} \quad \ldots \quad e_n \to q_{j_n} \quad (n \neq 1) \qquad \phi \to q \in \mathcal{R} \quad \#\mathcal{Q}(q_{j_1}|\ldots|q_{j_n}) \in [\![\phi]\!]}{e_1|\ldots|e_n \to q}$$

Table 2.11: The transition relation $\to$

An example automaton which accepts all trees with no more as than bs in parallel at any sub-tree, is as follows: $\mathcal{Q} = \{q_a, q_b, q_s\}$, $\mathcal{Q}_{fin} = \{q_s\}$ and $\mathcal{R}$ contains three rules: $a[q_s] \to q_a$, $b[q_s] \to q_b$ and $(\#q_a \leq \#q_b) \wedge (\#q_s \geq 0) \to q_s$.

Since the constraint $(\#q_a \leq \#q_b) \wedge (\#q_s \geq 0) \to q_s$ is satisfied by $(0,0,0)$, we have that $\mathbf{0} \to q_s$. A possible accepting run of the automaton is as follows:

$$\begin{array}{l} a[\mathbf{0}]|b[a[\mathbf{0}]|b[\mathbf{0}]|b[\mathbf{0}]] \to a[q_s]|b[a[\mathbf{0}]|b[\mathbf{0}]|b[\mathbf{0}]] \to a[q_s]|b[a[q_s]|b[\mathbf{0}]|b[\mathbf{0}]] \to \\ a[q_s]|b[a[q_s]|b[q_s]|b[\mathbf{0}]] \to a[q_s]|b[a[q_s]|b[q_s]|b[q_s]] \to a[q_s]|b[a[q_s]|b[q_s]|q_b] \to \\ a[q_s]|b[a[q_s]|q_b|q_b] \to a[q_s]|b[q_a|q_b|q_b] \to q_a|b[q_a|q_b|q_b] \leadsto \\ q_a|b[q_s] \to q_a|q_b \leadsto q_s \end{array}$$

For the transitions marked $\leadsto$, we use the type 2 rule of the example. In the first case the multiset, $q_a|q_b|q_b$, is accepted ($\phi(1,2,0)$ holds) and in the second case, the multiset, $q_a|q_b$, satisfies ϕ. That is, $\phi(1,1,0)$ holds.

An automaton is deterministic iff for every pair of distinct type 1 rules, $\alpha[q] \to q_1$ and $\beta[q] \to q_2$, we have $[\![\alpha]\!] \cap [\![\beta]\!] = \emptyset$; and, for every distinct pair of type 2 rules $\phi \to q_1$ and $\psi \to q_2$, we have $[\![\phi]\!] \cap [\![\psi]\!] = \emptyset$. An automaton accepts a tree, d, if there is a final state $q \in Q_{fin}$ such that $d \to q$. The language $\mathcal{L}(\mathcal{A})$ is the set of trees accepted by $\mathcal{A}$.

When checking whether a particular tree is accepted by an automaton, determinism is a desirable property. This is because a non-deterministic automaton will require arbitrary decisions to be made throughout the algorithm. To be able to conclude that a tree is not accepted by the automaton we must evaluate every possible run — increasing the complexity exponentially.

For an automaton, $\mathcal{A} = \langle \mathcal{Q}, \mathcal{Q}_{fin}, \mathcal{R} \rangle$, it is possible to check whether a tree, d, is in the language accepted by $\mathcal{A}$ in time $\mathrm{O}(|d|.|\mathcal{R}|.Cost(|\mathcal{Q}|, |d|))$, where $Cost(|\mathcal{Q}|, |d|)$ is a function that returns the time in which all constraints, ϕ, can be evaluated, $|d|$ is the size of the tree, and $|\mathcal{R}|$ is the number of rules in $\mathcal{A}$. Intuitively, we can derive this result by considering the worst case execution where, at every stage in the evaluation of the tree we test each rule in $\mathcal{R}$. In the worst case, testing the rule $\mathcal{R}$ takes $Cost(|\mathcal{Q}|, |d|)$ time.

However, this result requires that the automaton, $\mathcal{A}$, is deterministic. For every non-deterministic automaton, $\mathcal{A}$, we can construct a deterministic automaton, $\det(\mathcal{A})$, such that $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\det(\mathcal{A}))$. However, the states of $\det(\mathcal{A})$ is the power-set of the states of $\mathcal{A}$ and the number of rules in $\det(\mathcal{A})$ is expo-

nential in the size of $\mathcal{A}$.

Product Automata

Given two automata, we can construct the product automaton. We can define the final states of this automaton to test several properties. For example, we can construct the automaton that accepts the intersection of the original automata. The ability to build such automata is important when translating Sheaves Logic formulae to automata (described in section 2.4.8).

The product of two automata $\mathcal{A} = \langle \mathcal{Q}, \mathcal{Q}_{fin}, \mathcal{R} \rangle$ and $\mathcal{A}' = \langle \mathcal{Q}', \mathcal{Q}'_{fin}, \mathcal{R}' \rangle$ is defined as follows: $\mathcal{A} \times \mathcal{A}' = \mathcal{A}^{\times} = \langle \mathcal{Q}^{\times}, \emptyset, \mathcal{R}^{\times} \rangle$, where $\mathcal{Q}^{\times} = \mathcal{Q} \times \mathcal{Q}' = \{(q_1, q'_1), \ldots, (q_p, q'_r)\}$, and:

- For every type 1 rule, $\alpha[q] \rightarrow s \in \mathcal{R}$ and $\alpha[q'] \rightarrow s' \in \mathcal{R}'$, if $\alpha \cap \beta \neq \emptyset$ then the rule $(\alpha \cap \beta)[(q, q')] \rightarrow (s, s')$ is in $\mathcal{R}^{\times}$.
- For every type 2 rule $\phi \rightarrow q \in \mathcal{R}$ and $\phi' \rightarrow q' \in \mathcal{R}'$, the rule $\phi^{\times} \rightarrow (q, q')$ is in $\mathcal{R}^{\times}$, where $\phi^{\times}$ is the product of the formulae ϕ and ϕ', obtained as follows: let $\#(q, q')$ be the variable associated with the numbers of occurrences of the state (q, q'), then $\phi^{\times}$ is the formula:

$$\phi \left(\sum_{q' \in \mathcal{Q}'} \#(q_1, q'), \ldots, \sum_{q' \in \mathcal{Q}'} \#(q_p, q') \right) \wedge \phi' \left(\sum_{q \in \mathcal{Q}} \#(q, q'_1), \ldots, \sum_{q \in \mathcal{Q}} \#(q_p, q'_r) \right)$$

Intuitively, the product automaton combines the automata, $\mathcal{A}$ and $\mathcal{A}'$, producing an automaton that is equivalent to running $\mathcal{A}$ and $\mathcal{A}'$ 'in parallel'. The states of $\mathcal{A}^{\times}$ represent all possible combinations of the states of $\mathcal{A}$ and $\mathcal{A}'$ and the rules, derived from the rules of $\mathcal{A}$ and $\mathcal{A}'$, are adjusted to account for the existence of the second automaton.

Given two automata, $\mathcal{A}$ and $\mathcal{A}'$, to produce an automaton that accepts the union or the intersection of the two automata we take the set of final states of the automaton $\mathcal{A} \times \mathcal{A}'$ to be $\{(q, q') | q \in \mathcal{Q}_{fin} \vee q' \in \mathcal{Q}'_{fin}\}$ and $\{(q, q') | q \in \mathcal{Q}_{fin} \wedge q' \in \mathcal{Q}'_{fin}\}$ respectively. That is, either both automata accept the tree, or either of the automata accept the tree.

Dal Zilio et al state that the test $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{A}')$ (a sub-typing test) can be evaluated using the automaton $\mathcal{A} \times \mathcal{A}'$ with the final states $\mathcal{Q}_{fin} \times (\mathcal{Q}' \backslash \mathcal{Q}'_{fin})$, provided that $\mathcal{A}'$ is deterministic. One can show, using a simple counter example, that this is not the case. For the property to hold, we must also enforce that the automata are complete.

Test for Emptiness

The algorithm that computes whether $\mathcal{L}(\mathcal{A}) = \emptyset$ in time $O(|\mathcal{Q}|.|\mathcal{R}|.Cost_A)$ is given in table 2.12. $Cost_A$ is the maximal time required to decide the satisfiability of the type 2 constraints, $|\mathcal{Q}|$ is the number of states of $\mathcal{A}$ and $|\mathcal{R}|$ is the number of rules in $\mathcal{A}$. This result can be seen by observing that the main

$\mathcal{Q}_{[]} = \emptyset$
$\mathcal{Q}_M = \{q | \phi \to q \in \mathcal{R} \wedge \models \phi(\mathbf{0})\}$

repeat
  if $\alpha[q'] \to q \in \mathcal{R}$ and $q' \in \mathcal{Q}_M$ and $\alpha \neq \emptyset$
  then $\mathcal{Q}_M ::= \mathcal{Q}_M \cup \{q\}$ and $\mathcal{Q}_{[]} ::= \mathcal{Q}_{[]} \cup \{q\}$
  if $\phi \to q \in \mathcal{R}$ and $\phi \backslash \mathcal{Q}_{[]}$ is satisfiable
  then $\mathcal{Q}_M ::= \mathcal{Q}_M \cup \{q\}$
until no new state can be added to $\mathcal{Q}_M$

if $\mathcal{Q}_M$ contains a final state
then return not empty else return empty

Table 2.12: Testing an automaton for emptiness

loop of the algorithm has to add at least one state to $\mathcal{Q}_M$ during each iteration, and therefore the maximum number of iterations is the number of states in $\mathcal{Q}$. During each iteration we check each rule for satisfiability — hence we check $|\mathcal{R}|$ rules per iterations. In the worst case, checking a rule requires the evaluation of a Presburger constraint, therefore, in the worst case, it takes $Cost_A$ time to check a rule.

The algorithm works by assessing which states of the automaton are reachable. We maintain two sets of states: $\mathcal{Q}_M$ contains all states that are reachable, and $\mathcal{Q}_{[]}$ contains all sets that are reachable by the application of a type 1 rule; that is, the reachable states that correspond to an element formula. We write $\phi \backslash \mathcal{Q}_{[]}$ is satisfiable iff the formula $\phi(\#q_1, \ldots, \#q_p) \wedge \bigwedge_{q \notin \mathcal{Q}_{[]}} \#q = 0$ holds. This construct is used to ensure that we take account of which states are reachable before we utilize a type 2 rule.

Initially, all the states that are immediately reachable are added to the set of reachable states. These states are those type 2 rules that accept the empty tree. The empty tree represents the leaf nodes of any tree. The algorithm then iteratively calculates which type 1 rules can be applied — determining the set of all possible tree elements at the current stage of execution — and then calculates all possible states reachable by the application of a type 2 rule, given the set of element formulae that are available. The algorithm terminates when no more states will be added to the set of reachable states.

2.4.8 Constructing Automata for the Sheaves Logic

Given any Sheaves Logic formula A, we can build an automaton that accepts the same trees as A. The procedure is recursive, there are two cases to consider:

- Case $A = \top$

We can simply choose an automaton that accepts all trees. For example, the automaton with unique final state, q, and with rules $\emptyset^{\perp}[q] \rightarrow q$, and $(\#q \geq 0) \rightarrow q$.

- Case is $A = \exists \mathbf{N}.\phi(\mathbf{N}) \cdot (\alpha_1[A_1], \ldots, \alpha_p[A_p])$

  In this case we can create, recursively, automata $\mathcal{A}_i$ for each formula A_i defined in A. From each automaton $\mathcal{A}_i$ we can construct the automaton $\mathcal{A}_i^{\alpha_i}$ from $\mathcal{A}_i$ that accepts all trees that are formed by a branch whose label is in α_i atop a subtree accepted by A_i. We can achieve this by adding a new state, q_s, that is the only final state of $\mathcal{A}_i^{\alpha_i}$ and then adding the type 1 rule $\alpha[q] \rightarrow q_s$ for each final state q of $\mathcal{A}_i$. For a tree to be accepted by $\mathcal{A}_i^{\alpha_i}$ the new type 1 rule must be applied, leaving the single state q_s. For this to be the case the root of the tree must have only one branch, whose label is in α_i and whose sub-tree is accepted by $\mathcal{A}_i$ — that is, whose sub-tree is a model of A_i.

  After constructing $\mathcal{A}_i^{\alpha_i}$ for all $i \in 1..p$, we construct $\mathcal{A}$ by calculating the product automaton of the $\mathcal{A}_i^{\alpha_i}$s constructed in the previous step. Let the set of states of $\mathcal{A}$ be $\{\mathcal{Q}_1, \ldots, \mathcal{Q}_m\}$. Each state, $\mathcal{Q}$, of $\mathcal{A}$ will be of the form $(q_1, \ldots, q_p)$ where each q_i is a state of $\mathcal{A}_i^{\alpha_i}$. We write $\mathcal{Q} \in \text{fin}(i)$ to denote that q_i is a final state of $\mathcal{A}_i^{\alpha_i}$ and M_i to denote the number of occurrences of the state $\mathcal{Q}_i$ in a term $\mathcal{Q}_{j_1}|...|\mathcal{Q}_{j_n}$.

  We complete the automaton by adding a new state q_F that will be the only final state of $\mathcal{A}$. We then add the type 2 rule $\phi^{\exists}(M_1, \ldots, M_m) \rightarrow q_s$. We define $\phi^{\exists}$ such that it accepts only those configurations $\mathcal{Q}_{j_1}|...|\mathcal{Q}_{j_n}$ where each $\mathcal{Q}_{j_k}$ contains a final state for any $\mathcal{A}_i^{\alpha_i}$. That is, the configuration is built from sub-trees that are accepted by any of the element formulae of A. $\phi^{\exists}$ is defined as follows:

$$\exists (X_i^j)_{\substack{i\in 1..m \\ j \in 1..p}} . \left(\begin{array}{l} \bigwedge_{i\in 1..m} \left(M_i = \sum_{\substack{j\in 1..p \\ \mathcal{Q}_i \in \text{fin}(j)}} X_i^j \right) \\ \wedge \phi \left(\sum_{\substack{i\in 1..m \\ \mathcal{Q}_i \in \text{fin}(1)}} X_1^i, \ldots, \sum_{\substack{i\in 1..m \\ \mathcal{Q}_i \in \text{fin}(p)}} X_p^i \right) \end{array} \right)$$

  Because each state $\mathcal{Q}_i$ may represent an accepting run of several element formulae, we use the variables X_j^i to break M_i down into the different element formulae it may represent. We then test ϕ to see if it holds for that particular distribution of the possible element formulae. The rule is applied if such a decomposition exists.

2.4.9 Complexity

A Sheaves Logic formula, A, has a tree-like structure, therefore we can define the height, $h(A)$, of A, and the degree, $d(A)$, of A. Building the automaton that accepts A requires the construction of the product of at most $d(A)$ automata for each sub-formula in A. It is easy to see that the size of the automaton, $\mathcal{B} \times \mathcal{B}'$, is the size of $\mathcal{B}$ multiplied by the size of $\mathcal{B}'$, therefore, the size of the

$X, Y, \ldots$	Recursive variables
$E ::=$	Element formula
$\quad \alpha[X]$	$\quad$ Element with label in α
$D ::=$	Recursive definition
$\quad X \leftarrow \exists \mathbf{N}.\phi(\mathbf{N}) \cdot \mathbf{E}$	Sheaves composition
$A ::=$	RSL formula
$\quad \langle D_1, \ldots, D_n; X \rangle$	

Table 2.13: The Recursive Sheaves Logic syntax

automaton $\mathcal{A}$ is at most the size of $\mathcal{A}'$ to the power $d(A)$, where $\mathcal{A}'$ is the largest sub-automaton of $\mathcal{A}$. At each level in the tree, a maximum of $d(A)$ automata must be combined. The base case is at the leaves of the tree, where $A' = \top$. Consequently, defining T as the size of the automaton accepting $\top$, the size of the automaton accepting A is of size $\mathbf{O}(T^{d(A)^{h(A)}})$.

When translating a Tree Logic formula, A, to a Sheaves Logic formula, A_s, we find that $h(A_s)$ is bounded by $|A|$ and that $d(A_s)$ is at most $3^{|A|}$. $3^{|A|}$ is derived from the size of the smallest basis (the branch operator) and the size of a common basis, which is equal to the sizes of the original bases multiplied together. Hence, the size of the largest basis in A_s is $3^{|A|}$. Therefore, the size of the automaton accepting the Tree Logic formula, A, is $\mathbf{O}(T^{(3^{|A|})^{|A|}})$, or $\mathbf{O}(T^{3^{|A|^2}})$.

We can test the satisfiability of an automaton, $\mathcal{A} = \langle \mathcal{Q}, \mathcal{Q}_{fin}, \mathcal{R} \rangle$ in $O(|\mathcal{Q}|.|\mathcal{R}|.CostA)$ time. For a formula A, $|\mathcal{Q}|$ and $|\mathcal{R}|$ are bounded by $T^{3^{|A|^2}}$, and hence, the satisfiability of A can be determined in $O(T^{3^{|A|^2}}.T^{3^{|A|^2}}.CostA)$ time. That is, $O(T^{2.(3^{|A|^2})})$ when abstracting over the cost of the Presburger constraints. Therefore, the satisfiability of A is doubly exponential. Similarly, we can test $d \models A$ in $O(|d|.T^{3^{|A|^2}})$ time.

2.4.10 Recursive Sheaves Logic

Table 2.13 gives the syntax for a recursive variant of sheaves logic. The main advantages of this logic are increased expressivity and a much neater automaton construction algorithm. The Recursive Sheaves Logic is more expressive than the Sheaves Logic because of the use of recursive variables: formulae can refer to themselves, or a set of formulae may be mutually recursive. This allows us to reason about paths of an infinite length that match a repeating pattern.

We write $D \vdash d : X$ to denote that tree d matches a formula $\langle D; X \rangle$. A tree d matches a formula iff there is a definition in D of the form $X \leftarrow \exists \mathbf{N}.\phi(\mathbf{N}) \cdot \mathbf{E}$ and d satisfies $\exists \mathbf{N}.\phi(\mathbf{N}) \cdot \mathbf{E}$. The semantics of $\exists \mathbf{N}.\phi(\mathbf{N}) \cdot \mathbf{E}$ are similar to the Sheaves Logic, except that an element formula, $\alpha[Y]$, is satisfied by a tree of

the form, a[d], where $a \in \alpha$ and $A \vdash d : Y$.

The following example is not strictly an RSL formula, as the right hand side of the recursive definitions are not in the correct format, but it should show how a recursive formula is evaluated:

$\langle X \leftarrow (a[Y]|\top) \vee \mathbf{0}, Y \leftarrow (b[X]|\top); X\rangle$ is a formula that matches any tree with a path $(a.b)^*$ — that is, a tree with a path of alternating as and bs. Firstly, X must be satisfied, requiring a branch labeled a, atop a tree satisfying Y, which requires a branch labeled b atop a tree satisfying X, etc.

Constructing Automaton

The construction of an automaton that accepts the same trees as a recursive formula, A, is simpler than the method for a non-recursive formula. We set up the state $Q_{\alpha[Y]}$ for every element formula $\alpha[Y]$ of A, and the state q_Z for every recursive variable, Z in A. If we let $A = \langle D; X\rangle$, the automaton has the single final state q_X.

We add a type 1 rule, $\alpha[q_Y] \rightarrow q_{\alpha[Y]}$, to the automaton for every element formula, $\alpha[Y]$. This rule is fired whenever we have a single tree that satisfies Y below a branch whose label is in α — that is, it is fired whenever we have a tree that satisfies the element formula $\alpha[Y]$.

Finally, we add a type 2 rule, $\phi(\#q_{\alpha_1[Y_1]}, \ldots, \#q_{\alpha_p[Y_p]}) \rightarrow q_Y$, for each definition, $Y \leftarrow \exists \mathbf{N}.\phi(\mathbf{N}) \cdot \mathbf{E}$ in D, with $E = (\alpha_1[Y_1], \ldots, \alpha_p[Y_p])$. Intuitively, this formula accepts all trees composed from sub-trees that satisfy the element formulae in E whose multiplicities meet the restrictions specified in ϕ.

2.4.11 The Kleene Star

The Kleene Star is a Tree Logic operator, written $\mathcal{A}^*$, that means that sub-trees satisfying $\mathcal{A}$ may appear zero or more times in parallel. For example, $a[\mathbf{0}]^*$ can mean $\mathbf{0}$, or $a[\mathbf{0}]$, or $a[\mathbf{0}]|a[\mathbf{0}]$, or $a[\mathbf{0}]|a[\mathbf{0}]|a[\mathbf{0}]$, etc.

An advantage of Dal Zilio et al's approach is that it is possible to decide formulae that use the Kleene Star — this was not possible using Calcagno et al's method, described in section 2.3. Intuitively, Calcagno et al's approach was unable to prove decidability for the Kleene Star because it specified a potentially infinite number of sub-trees that could not be restricted using the notion of size used by Calcagno et al. Dal Zilio et al's approach uses Presburger formulae to count the multiplicity of element formulae, this count is not restricted (in the positive direction). In the simplest case the Tree Logic formula, $a[\mathbf{0}]^*$ can be represented using the Sheaves Logic formula, $\exists(n).n \geq 0 \cdot (a[\mathbf{0}])$.

In the general case, the translation of this operator is a complex and potentially expensive operation. For example, it is not obvious how a Sheave Logic formula that describes $(a[\mathbf{0}]|b[\mathbf{0}]^*)^*$ can be constructed. Due to time restrictions, the Kleene Star is beyond the scope of this project. Instead it is left as a possible avenue of future work.

2.4.12 Summary

The primary advantage of the approach put forward by Dal Zilio et al is the significant improvement in complexity over the previous approach. However, this complexity is still quite high: doubly exponential when abstracting over the cost of the Presburger constraints, which, in the worst case, are at least doubly exponential themselves. Fortunately current tools for evaluating Presburger formulae are quite efficient in many cases (three such tools are discussed in chapter 3). Furthermore, we are able to apply several optimisations when implementing Dal Zilio et al's method to help improve the run time.

Another benefit of the approach is that the Kleene Star has been proved decidable. Although this operator has not been included in this project, it represents a possible extension to it.

A drawback of this method is that the element formulae are defined down the whole depth of the tree. This means that the reasoning methods are bottom-up, and so the whole tree must be loaded when testing whether it satisfies a given formula. In the case of large trees, a top down approach would allow the relevant sections of the tree to be loaded as they are required, reducing memory requirements and the cost of loading the tree. Also, it means that we require separate element formulae to describe sub-trees that differ by a small amount at any point in the tree.

Chapter 3

Background: Presburger Constraint Solvers

The methods described in chapter 2 involve the evaluation of Presburger Constraints. This is not a trivial task and a lot of research has been conducted into the problem. An advantage of the approach described is that this body of work can be utilised in solving the problem of satisfaction and validity for the Tree Logic.

In this chapter an overview will be given of three software tools for the evaluation of Presburger Constraints. These three tools are LASH, The Omega Library and CVC.

The tool chosen for the implementation of this project was The Omega Library. We chose this tool because it allows a large amount of quantified variables to occur in the Presburger Formula and because its interface is the most suitable for this project. We discuss the reasoning behind the decision in more detail in chapter 4.

3.1 LASH

LASH [13] is an acronym for The Liège Automata-based Symbolic Handler. It is currently being maintained at Institut Montefiore, Université de Liège and it is described as "a tool-set for representing infinite sets and exploring infinite state spaces."

The LASH tool-set is primarily a set of C libraries that provides functions and datatypes for finite state automata and both finite and infinite sets of values. One of the features provided by LASH is the ability to represent NDDs, or Number Decision Diagrams. NDDs are more expressive than Presburger Arithmetic and can be used to solve Presburger constraints.

LASH also provides a front end for solving Presburger problems. This front end is a stand alone program which reads its input from a file specified on the command line and outputs the results to the standard output. For example,

```
EXISTS(x:  x>3 AND x<4)
```

Table 3.1: An example input file for LASH

```
Number of solutions  :  0.
Number of NDD states :  0.
Runtime statistics:
  residual memory :  0 byte(s).
  max memory      :  9426 byte(s).
```

Table 3.2: The output produced by LASH when passed the file shown in table 3.1

when passed the file shown in table 3.1, LASH produces the output shown in table 3.2.

Empirical tests have shown that LASH is capable of processing roughly 10 to 15 existentially quantified variables.

3.2 The Omega Library

The Omega Library [14] is part of the High Performance Software Systems Laboratory at the Computer Science Department of the University of Maryland, College Park. It is described as, "a system for manipulating sets of affine constraints over integer variables," and has been extended to solve Presburger problems.

The Omega Library has a front end called the Omega Calculator. This is an interactive command line program that reads its input from the standard input and produces output on the standard output. An example session with the Omega Calculator is given in table 3.3 (commands are delimited by a semi colon, and the answer is output on the next line).

The Omega Calculator has a compile time constant, `maxVars`, which can be set to any number. This number restricts the maximum number of existentially quantified variables allowed in a single formula. Increasing the number allows larger expressions to be evaluated at the expense of run time. A graph plotting the run times of the Omega Calculator for different values of `maxVars` is given in figure 3.1. The data used were the complete set of test cases discussed in chapter 7.

```
# Omega Calculator v1.2 (based on Omega Library 1.2, August, 2000):
{[i]: forall(x: x > 3)};
{[i] : FALSE }
{[i]: forall(x, y: exists(z: x <= z and y >= z))};
{[i] : FALSE }
{[i]: forall(x: exists(z: x <= z))};
{[i]}
```

Table 3.3: An example session with the Omega Calculator

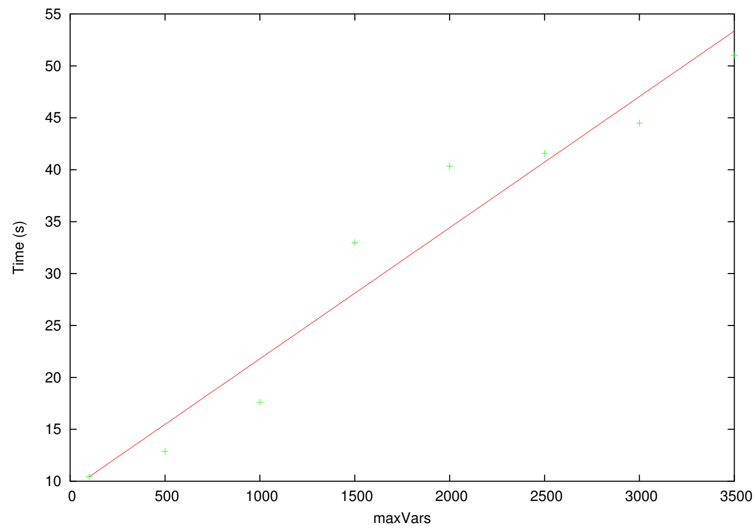


Figure 3.1: The value of maxVars against run-time

```
x:  REAL; QUERY NOT(x > 3);
Invalid.
x, y:  REAL; QUERY (x >= y) AND (y >= x) => x = y;
Valid.
a, b:  REAL; QUERY (a > 0 AND b > 0 AND a + b = 2) => (a = 1 AND b = 1);
Invalid.
```

Table 3.4: A sample run of CVC

3.3 CVC

CVC [15] is a "Co-operating Validity Checker" that tests the validity of mathematical formulae with logical connectives. CVC does not allow quantified variables, but it does allow free variables, which are universally quantified implicitly.

CVC is a command line program that reads its input from the standard input and writes to the standard output. From the command line it is possible to specify whether CVC produces full proofs and whether it should check for validity or satisfaction.

However, CVC only allows free variables to be reals and not integers. Since Presburger formulae range over integer values, CVC will not be suitable for this project. For example, we may want to consider the constraint $3 < x < 4$; this constraint specifies no integer values, but it does specify infinitely many reals.

A sample run of CVC is given in table 3.4. The third query demonstrates a situation where variables of type integer are required.

Chapter 4

Implementation: Choosing the Approach

In this chapter a summary of the different methods for calculating the validity or satisfaction for formulae written in the Tree Logic is given. A choice is made as to which of the approaches will provide the best implementation. Additionally, an evaluation of the constraint solvers introduced in chapter 3 is presented. Finally, the language for the implementation is chosen.

4.1 Validation and Satisfaction of Tree Logic Formulae

In chapter 2 there are several different methods for deciding the validity of a Tree Logic formula. There are two main choices that have to be made: whether a model-checking (satisfaction) approach or a validity test is used, and whether the Sheaves Logic or the Recursive Sheaves Logic is used.

4.1.1 Satisfaction/Validity

In section 2.2 it is shown that satisfaction and validity of a Tree Logic formula are mutually expressible. This means that we can use either the model checking approach described in section 2.4.7 or the test for emptiness described in section 2.4.7.

Given a Sheaves Logic formula or a Recursive Sheaves Logic formula, A, we can construct an automaton using an algorithm derived from the proofs in section 2.4.8 or section 2.4.10 respectively. In general, the automata produced by these methods are non-deterministic. For example, the tree logic formula $a[\mathbf{T}]|b[\mathbf{T}]$ has two occurrences of truth. The automata constructed for both logics will have multiple representations of truth, and so each automaton will

have type 2 rules, $\phi \rightarrow q$ and $\psi \rightarrow q'$ such that $[\![\phi]\!] \cap [\![\psi]\!] \neq \emptyset$, violating the conditions for determinism presented in section 2.4.7.

Given a non-deterministic automaton, $\mathcal{A}$, we can construct a deterministic automaton, $\mathcal{A}'$ such that $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}')$. However, the size of $\det(\mathcal{A})$ is exponentially related to the size of $\mathcal{A}$.

For an automaton $\mathcal{A} = \langle \mathcal{Q}, \mathcal{Q}_{fin}, \mathcal{R} \rangle$, we can check if a tree d is in the language accepted by $\mathcal{A}$ in time $O(|d|.|\mathcal{R}|.Cost(|\mathcal{Q}|, |d|))$, where $Cost(p, n)$ is a function that returns the time in which all constraints, ϕ can be evaluated. This result requires that the automaton, $\mathcal{A}$ is deterministic.

Alternatively, the test for emptiness is decidable in time $O(|\mathcal{Q}|.|\mathcal{R}|.Cost_{\mathcal{A}})$, where $Cost_{\mathcal{A}}$ is the maximum time required to decide the satisfiability of the type 2 constraints in $\mathcal{A}$. The test for emptiness does not require that the automaton is deterministic.

Because model checking (satisfaction) requires that the automaton is deterministic, its complexity will be greater than the test for emptiness (validation). Consequently the validation approach was chosen.

However, when testing whether $d \models A$ for some tree d and some formula A, the validation approach requires that we check the validity of the formula $\underline{d} \Rightarrow A$. This increases the size of the formula and consequently the size of the automaton. It does not, however, increase the complexity of the method. A possible solution to this problem is discussed in chapter 7.

4.1.2 Sheaves Logic/Recursive Sheaves Logic

The two types of Sheaves Logic are presented in section 2.4.2 and section 2.4.10. The main difference between the two logics is the introduction of recursive variables in the recursive variant. These recursive variable flatten the structure of the formulae, which has several benefits.

Firstly, to construct an automaton from the Recursive Sheaves Logic is more straight-forward than the construction of automata for Sheaves Logic. The methods for automata construction for the Sheaves Logic and the Recursive Sheaves Logic are given in section 2.4.8 and section 2.4.10 respectively. The generation of Sheaves Logic automaton requires the recursive generation of sub-automata that must be manipulated for the construction of the larger automaton. Then a complex type 2 rule must be added to "bind" the sub-automata together. Comparatively, the automata for the Recursive Sheaves Logic can be constructed directly from given formula.

Additionally, the introduction of recursive variables increases the scope for optimisation. For example, if a formula contains repeated sub-formulae, then each instance of the repeated sub-formulae can be replaced by a single formula, and the recursive variables can be replaced with the recursive variable for the new formula.

Because of the advantages described above, the recursive variant of the Sheaves Logic was used for the implementation of this project.

4.2 The Constraint Solver

Three alternative constraint solvers were described in chapter 3. These were, LASH, The Omega Library and CVC.

Because CVC does not support integer variables and has limited support for universal quantification, it is not ideal for the implementation of this project.

The main advantage of LASH is that, unlike the Omega Library, it provides a stand-alone executable that solves Presburger constraints specifically. This means that the resulting implementation will be smaller in size, as it does not include facilities for solving a wider class of numerical constraints.

However, LASH has two primary drawbacks:

- LASH is restricted to only a few quantified variables: about 15 or so. During the implementation of this project it quickly became clear that more variables were required — even when the use of variables is optimised.
- LASH's Presburger solver reads its input from a file that is specified on the command line. For the purposes of this project, this is less efficient than reading from the standard input. This is because, to communicate with the solver, a file containing the constraint must be created. As a result, communication suffers from a greater overhead than when input can be sent directly to the solver via a Unix pipe.

The Omega Library overcomes both of these problems: constraints can be written to the Omega Calculator's standard input, and the result is returned via its standard output. Further to reducing the cost of creating a constraint file, this system also allows us to keep a single instance of the Omega Calculator running throughout the evaluation of a formula. This removes the overhead present when using LASH's Presburger solver, which is caused by the need to start a new instance of LASH for each constraint that needs to be evaluated. [1]

4.3 The Implementation Language

The language chosen for the implementation is OCaml [16, 17]. OCaml is a variant of the functional language ML.

A functional language is suited to this project because of the recursive nature of the formulae that the implementation will evaluate. The use of pattern matching means that functional languages have an advantage over imperative languages when the data is defined recursively. This feature means that the resultant code is more succinct and more natural than the code that would be required if an imperative language were used.

The functional language OCaml also allows the use of "imperative features" and provides a good Unix library. This allows the use of system calls — which are required for communicating with the constraint solver — in a more comfortable imperative manner.

[1] A buffering strategy will reduce this overhead, but a similar buffering strategy could also be used when interacting with the Omega Calculator.

Chapter 5

Implementation: Design and Algorithms

In this chapter a high-level overview of the program structure is given, followed by a discussion of the algorithms required for a naive implementation without optimisations. Optimisations to these algorithms are discussed in chapter 6.

5.1 Design Overview

Figure 5.1 gives an overview of the structure of the implementation. The structure is:

- The input (a string) is translated into a problem record — which holds the kind of problem to be solved and the trees and/or Tree Logic formulae that are parameters to the problem.
- A single formula representing the problem is constructed.
- The formula is translated into a Sheaves Logic formula.
- The Sheaves Logic formula is translated into a Recursive Sheaves Logic formula.
- An automaton is constructed from the resultant formula.
- The automaton is tested for emptiness. The test for emptiness may involve the evaluation of several Presburger formulae, therefore, we interact with the Presburger Constraint solver during this stage
- The result is processed to form the output of the program.

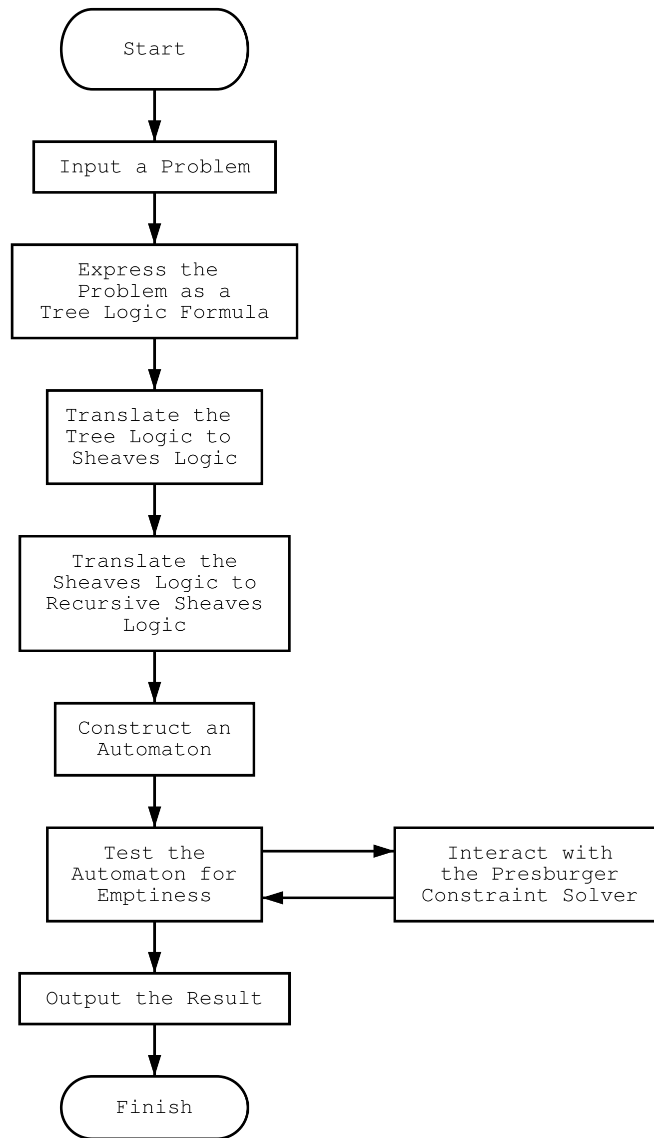


Figure 5.1: An overview of the program structure

5.2 User Interface

5.2.1 Type of Interface

The implementation provides a simple command line interface that reads from the standard input and writes to the standard output. This type of interface has an advantage over the alternatives because the standard input and output channels are very flexible. By using these channels the program is able to accept input from a file, a terminal or from another process via a pipe. Similarly, the output can be sent to a file, a terminal or to another process.

Alternatively, a graphical user interface could have been provided. A GUI would not have been as flexible as the command line approach, but it would have been more usable for a human. For example, the GUI could have provided tools for visually constructing trees, or it could have displayed graphically a schema of the kinds of trees a given formula accepts. These features, however, are not trivial to implement and are beyond the scope of this project. A cut down interface, however, would provide very little over the command line approach.

Another alternative is to read and write to files, rather than the standard I/O channels. However, this feature can be provided by the standard I/O channels, but features such as direct user interaction cannot be provided if the program handles files directly.

5.2.2 Grammar

The grammar accepted by the implementation is given in table 5.1. The lexer and parser generators provided by OCaml [17] were used to construct the interface as automatically as possible. The parser accepts the standard input and returns "problem" records to the main program, which interprets and solves them — sending the result to the standard output.

The grammar for trees and tree logic formulae was derived directly from the syntax in tables 2.1 and 2.2. The additional constructs are:

- Commands — commands describe a problem and are terminated by a semi-colon. If the command begins with the keyword `EXPTRUE` or `EXPFALSE` then the program will expect the result of the problem to be true or false respectively. These keywords are especially useful in cases such as testing, where we are interested in whether the result is as expected, rather than in the specific result.
- `VALID` — a validity problem takes a tree logic formula as its input and outputs 'true' if the formula is valid, 'false' otherwise.
- `SATISFIABLE` — a satisfaction problem takes a tree logic formula as its input and outputs 'true' if the formula is satisfiable, 'false' otherwise.
- `SUBTYPE` — a sub-typing problem takes two tree logic formulae, $\mathcal{A}$ and $\mathcal{B}$ as its input and returns 'true' if the formula $\mathcal{A}$ represents a sub-type of $\mathcal{B}$. This is equivalent to testing the validity of $\mathcal{A} \Rightarrow \mathcal{B}$.

commands:	
problem;	A command terminated with a semi-colon.
`EXPTRUE` *problem*;	A command whose result is expected to be 'True'.
`EXPFALSE` *problem*;	A command whose result is expected to be 'False'.
problem:	
`VALID` *tl*	A validity problem.
`SATISFIABLE` *tl*	A satisfiability problem.
tl `SUBTYPE` *tl*	A sub-typing problem
tree `OFTYPE` *tl*	A satisfaction problem
`EXIT`	Quit the program.
tree:	
`0`	Void
a[*tree*]	Branch
tree`|`*tree*	Composition
`(`*tree*`)`	Parenthesis
tl:	
`0`	Void
`T`	Truth
`F`	Falsity
a[*tl*]	Branch
tl`@`*a*	Placement
tl`|`*tl*	Composition
tl `->` *tl*	Implication
tl `AND` *tl*	Conjunction
tl `OR` *tl*	Disjunction
tl `|>` *tl*	Guarantee
`NOT` *tl*	Negation
`(`*tl*`)`	Parenthesis

Table 5.1: The input grammar

Let $A = \exists(n_1, \ldots, n_p).\phi_A \cdot (\alpha_1[B_1], \ldots, \alpha_p[B_p])$

Assume that $B_i = \exists \mathbf{N_B}.\phi_{B_i} \cdot \mathbf{E}$ for all $i \in 1..p$
where $\mathbf{N_B}$ and $\mathbf{E}$ are common to all B_i.

Then $A@a =_{\text{def}}$
$$\exists \mathbf{N_B}. \left[(\exists(n_1, \ldots, n_p). \left(\phi_A \wedge \bigvee_{\substack{i \in 1..p \\ a \in \alpha_i}} \left(n_i = 1 \wedge \bigwedge_{\substack{j \in 1..p \\ i \neq j}} (n_j = 0) \wedge \phi_{B_i} \right) \right) \right] \cdot \mathbf{E}$$

Table 5.2: Encoding placement in Sheaves Logic

- `OFTYPE` — a typing problem takes a tree, d, and a formula, $\mathcal{A}$ as its input, and returns 'true' if $d \models \mathcal{A}$, and 'false' otherwise.

5.3 Translating Tree Logic to Sheaves Logic

Because satisfaction, satisfiability and validity are mutually expressible in Tree Logic, each of the input problems can be described as a Tree Logic formula. The next step is to translate the formula into a Sheaves Logic formula.

In section 2.4.5 Dal Zilio et al's translations from Tree Logic to Sheaves Logic are given. These translations are inductive and for operators whose arguments contain zero or one tree logic formulae, the translation algorithm is simply a direct encoding. The algorithm for more complex operators is given in section 5.3.2.

5.3.1 The Placement Modality

A notable omission from Dal Zilio et al's translations is that of the placement modality (@). An encoding of this modality is given in table 5.2 and the proof of the translation is given in appendix A.1.1. The algorithm for translating placement is given in table 5.3.

5.3.2 Binary Connectives

For operators that take two tree logic formulae (such as conjunction and guarantee) the translations in section 2.4.5 cannot be applied directly. This is because a pre-condition of the translations is that both formulae are defined over a common basis. Therefore, before we can apply the conversion, we must ensure that both formulae are defined over the same basis. The algorithm is given in table 5.4.

```
translate (A@a)
  Case A = ⊤:  Return ⊤
  Case A = ∃N.φ_A · (α_1[B_1], ..., α_p[B_p]):
    Construct a common basis, E, from E_{B_i} where i ∈ 1..p
    Translate φ_{B_i} to use the basis E for all i ∈ 1..p
    Construct and return the formula in table 5.2
```

Table 5.3: Algorithm for translating the placement operator

```
translate (A op B), op ∈ {∧, ∨, |, ▷}
  Case A = ⊤ and B = ⊤:  Return ⊤
  Case A = ⊤ xor B = ⊤:
    Replace ⊤ with the formula ∃N.(N ≥ 0) · E,
    where E is the required basis.
    Complete the translation as defined in section 2.4.5
  Case A = ∃N_A.φ_A · E_A and B = ∃N_B.φ_B · E_B:
    Construct a common basis, E, from E_A and E_B
    Translate φ_A and φ_B to use the basis E
    Complete the translation as defined in section 2.4.5
```

Table 5.4: Algorithm for translating binary connectives

```
common basis ((α_1[A_1], ..., α_p[A_p]), (β_1[B_1], ..., β_q[B_q]))
  Start with an empty basis, E and
  empty relations, R_A and R_B
  For all (i, j) ∈ 1..p × 1..q
    Convert A_i and B_j so that they share a common basis
    For some new k such that E_k is not already defined
    add the element E_k = (α_i ∩ β_j)[A_i ∧ B_i] to E
    Add (i, k) to R_A
    Add (j, k) to R_B
  Return (E, R_A, R_B)
```

Table 5.5: Algorithm for constructing a common basis

5.3.3 Finding a Common Basis

When combining two sub-formulae, the translations require that the two formulae are defined over the same basis. Therefore we must be able to find a common basis given two heterogeneous bases. The algorithm (given in table 5.5) has been lifted from Dal Zilio et al's proof (given in section 2.4.6) that, given any two bases, it is always possible to define a basis that refines each of the two original bases. It is worth noting that the construction of a common basis requires the construction of further common bases recursively. The base case occurs when either $A_i = \top$ or $B_i = \top$ — in this case a common basis can be defined trivially.

5.3.4 Redefining Formulae Over a Basis

Once a common basis for two formulae, $A = \exists \mathbf{N_A}.\phi_A \cdot \mathbf{E_A}$ and $B = \exists \mathbf{N_B}.\phi_B \cdot \mathbf{E_B}$ has been constructed, we need to construct formulae ϕ'_A and ϕ'_B such that $A \iff \exists \mathbf{N}.\phi'_A \cdot \mathbf{E}$ and $B \iff \exists \mathbf{N}.\phi'_B \cdot \mathbf{E}$, where $\mathbf{E}$ is the common basis. The ϕ'_A and ϕ'_B required are $\phi_{\mathcal{R}_\mathbf{A}}$ and $\phi_{\mathcal{R}_\mathbf{B}}$ respectively — this formula is defined in section 2.4.6.

5.4 Translating Sheaves Logic to Recursive Sheaves Logic

Dal Zilio et al did not provide a translation from Sheaves Logic to Recursive Sheaves Logic. A translation is given in table 5.6, the proof of the translation is given in appendix A.1.2. These translations can be applied directly to a Sheaves Logic formula.

The translation flattens the tree structure of the Sheaves Logic formulae by creating a definition of the form, $Y \leftarrow \exists \mathbf{N}.\phi(N).\mathbf{E}$, for each sub-formula in the

$$\begin{array}{rcl}\top & =_{\text{def}} & < X \leftarrow \exists N.(N \geq 0).\emptyset^{\perp}[X]; X > \\ \exists \mathbf{N}.\phi(\mathbf{N}) \cdot \mathbf{E} & =_{\text{def}} & < Y \leftarrow \exists \mathbf{N}.\phi(N).(\alpha_1[X_1], \ldots, \alpha[X_n]), D_1, \ldots, D_n; Y > \\ & & \text{where } \mathbf{E} = (\alpha_1[A_1], \ldots, \alpha[A_n]) \\ & & \text{inductively we enforce } A_i =< D_i; X_i > \text{ for all } i \in 1..n \\ & & i \neq j \Rightarrow rvn(A_i) \cap rvn(A_j) = \emptyset \text{ for all } i, j \in 1..n \\ & & \text{and } Y \notin \bigcup_{i \in 1..n} rvn(A_i)\end{array}$$

We write $rvn(A)$ to denote the set of all recursive variable names used in the formula A.

Table 5.6: A translation from Sheaves Logic to Recursive Sheaves Logic

Sheaves Logic formula.

5.4.1 Complexity

Let A_r be a recursive formula, it is easy to see that the method for building an automaton for A_r (described in section 2.4.10) produces an automaton, $\mathcal{A}$, that is linear in the size of A_r. This is because we add a state and a type 2 rule for each recursive definition in A_r, and a state and a type 1 rule for each element formula in A_r. Therefore the size of the automaton is $\mathbf{O}(|A_r|)$. Since the size of the automaton corresponding to a Sheaves Logic formula, A_s is $\mathbf{O}(T^{d(A)^{h(A)}})$, the recursive approach appears to be an improvement over the Sheaves Logic.

However, when translating from the Sheaves Logic to the Recursive Sheaves Logic, we add a recursive definition for each Sheaves Logic sub-formula of the Sheaves Logic formula. Sheaves Logic formulae can be thought of as a tree structure, where each sub-formula is a node, and the element formulae of the sub-formulae can be thought of as branches in the tree. Given that, for the Sheaves Logic formula, A_s, and the Tree Logic formula, A, the height, $h(A_s)$, of A_s is $\mathbf{O}(|A|)$ and the degree, $d(A_s)$, is $\mathbf{O}(3^{|A|})$, the number of branches in A_s is $\mathbf{O}((3^{|A|})^{|A|})$, or, $\mathbf{O}(3^{|A|^2})$. And hence, the automaton, $\mathcal{A}$ is of size $\mathbf{O}(T^{3^{|A|^2}})$ — the same as for the Sheaves Logic.

5.5 Constructing the Automaton

The next stage of the program — after constructing the Recursive Sheaves Logic formula — is to build the corresponding automaton. The algorithm can be lifted directly from Dal Zilio el al's proof, given in section 2.4.10, that an automaton can be built for every formula. The algorithm itself is presented in table 5.7.

```
build automaton (A = ⟨D; X⟩)
  Construct the set of states, 𝒬 (initially empty)
    For each element formula α[Y] in A, add q_{α[Y]} to 𝒬
    For each recursive variable Z in A, add q_Z to 𝒬
  Construct the set of rules, ℛ (initially empty)
    For each element formula α[Y] in A,
      add the rule α[q_Y] → q_{α[Y]} to ℛ
    For each definition Y ← ∃N.φ · (α_1[Y_1], ..., α_p[Y_p]) in D
      add the rule φ(#q_{α_1[Y_1]}, ..., #q_{α_p[Y_p]}) → q_Y to ℛ
  𝒬_fin ::= {q_X}
  Return ⟨𝒬, 𝒬_fin, ℛ⟩
```

Table 5.7: Algorithm for constructing an automaton from an RSL formula

5.6 Testing an Automaton for Emptiness

The algorithm defined by Dal Zilio et al in [8] tests whether the language accepted by an automaton is empty. This algorithm is described in table 2.12.

5.7 Translating Tree Logic to Recursive Sheaves Logic

An alternative approach to translating from Tree Logic to Recursive Sheaves Logic is to translate directly, rather than by translating the Tree Logic to Sheaves Logic and then the Sheaves Logic to Recursive Sheaves Logic. Potentially, this approach could be more efficient that the method presented here — for example, we may take advantage of recursive variables to reduce repeated Tree Logic sub-formulae to a single definition in D.

Translating from Sheaves Logic to its recursive variant is a fairly small step. Translating directly from Tree Logic to Recursive Sheaves Logic is more involved, and consequently, a direct translation is offered in section 7.4.6 as a possible extension to this project.

Chapter 6

Implementation: Optimisations

In chapter 5 the algorithms for a naive implementation of Dal Zilio et al's decision procedure were discussed. Because the complexity of the approach is doubly exponential, we need to optimise the implementation so that satisfactory tun-times may be achieved. We discuss several redundancies in the naive approach and discuss how these redundancies may be reduced. Each of the optimisations presented in this chapter has been implemented, and is evaluated in chapter 7.

6.1 Extending the Sheaves Logic Syntax

During the translation from Sheaves Logic to Recursive Sheaves Logic we create a new recursive variable definition, $X \leftarrow \exists(n).(n \geq 0).\emptyset^{\perp}[X]$ for each occurrence of $\top$ in the Sheaves Logic formula. This leads to redundancy as only one definition for truth is required. This redundancy means that the automaton constructed from the recursive formula has more rules than is strictly necessary, which causes inefficiency during the test for emptiness.

For example, consider the case where we are translating the Tree Logic formula $a[\mathbf{T}]$. The Sheaves Logic translation has the basis $(\{a\}[\top], \{a\}[\neg\top], \{a\}^{\perp}[\top])$. The recursive translation of this formula will then have two translations of $\top$ amongst its rules and as a result the constructed automaton will also have two rules representing truth. Both of these rules will be satisfied immediately during the test for emptiness, and so the states associated with the rules will be available throughout. If we were to replace these two rules with a single rule for truth that mapped to a state q, and replaced all occurrences of the previous "truth" states with this new state, then the automaton would accept exactly the same trees as the previous automaton, but it would also have fewer rules.

Therefore, we can reduce the size of the constructed automaton by producing a single rule for truth, instead of several rules that are applied in the same

$$
\begin{array}{rcl}
\top & =_{\text{def}} & \langle \emptyset; X_\top \rangle \\
\bot & =_{\text{def}} & \langle \emptyset; X_\bot \rangle \\
\mathbf{0} & =_{\text{def}} & \langle \emptyset; X_\mathbf{0} \rangle \\
\exists \mathbf{N}.\phi(\mathbf{N}) \cdot \mathbf{E} & =_{\text{def}} & \langle Y \leftarrow \exists \mathbf{N}.\phi(N) \cdot \mathbf{E}, D_1, \ldots, D_n; Y \rangle \\
 & & \text{where } \mathbf{E} = (\alpha_1[A_1], \ldots, \alpha[A_n]) \\
 & & \text{inductively we enforce } A_i = \langle D_i; X_i \rangle \text{ for all } i \in 1..n \\
 & & i \neq j \Rightarrow rvn(A_i) \cap rvn(A_j) \subseteq \{X_\top, X_\bot, X_\mathbf{0}\} \\
 & & \qquad \text{for all } i, j \in 1..n \\
 & & \text{and } Y \notin \bigcup_{i \in 1..n} rvn(A_i) \cup \{X_\top, X_\bot, X_\mathbf{0}\}
\end{array}
$$

After the formula has been translated, we add the following definitions:

$X_\top \leftarrow \exists N.(N \geq 0).\emptyset^\perp[X_\top]$
$X_\bot \leftarrow \exists N.(\mathbf{F}).\emptyset^\perp[X_\bot]$
$X_\mathbf{0} \leftarrow \exists N.(N = 0).\emptyset^\perp[X_\mathbf{0}]$

Table 6.1: A translation from the Extended Sheaves Logic to Recursive Sheaves Logic

situation. If we consider a Sheaves Logic formula to be a tree structure, $\top$ represents a leaf in the tree. In section 2.4.9 we state that the size of the Sheaves Logic translation, A_s, of the Tree Logic formula A, has $h(A_s)$ bounded by $|A|$, and $d(A_s)$ bounded by $3^{|A|}$. Therefore, the number of leaves in the tree is bounded by $3^{|A|^2}$. Therefore, there are a potentially exponential number of occurrences of $\top$, and so we can reduce the number of rules significantly.

We can extend this argument to the other singleton formulae in the Tree Logic, **F** and **0**. However, the Sheaves Logic does not have a simple representation for these two formulae, which means that it is not a trivial task to detect when we are creating a rule that represents **F** or **0**.

We can solve this problem by extending the sheaves logic to include simple representations for **F** and **0** — $\bot$ and **0** respectively. A further benefit of this approach is that we can optimise the construction of the sheaves logic formulae. For example, the the formula $\mathbf{F} \wedge \mathbf{0}$ can be translated directly to $\bot$, rather than by calculating the conjunction of the two Sheaves Logic formulae, $\exists(n).(n = 0) \cdot AnyE$ and $\exists(n).(\mathbf{F}) \cdot AnyE$. A full listing of these optimised translations are given with proofs in appendix A.2.1.

Changing the Sheaves Logic syntax requires a change in the translation to Recursive Sheaves Logic. This translation is given in table 6.1. The proof of the translation is giving in appendix A.2.2.

6.2 Optimising Basis Construction

Throughout the translation from Tree Logic to Sheaves Logic a common basis between two bases is constructed. The size of the common basis is equal to the sizes of the two original bases multiplied together. This has a cumulative effect over large formulae. Reducing the size of the common basis means that there will be fewer quantified variables in the associated Presburger formula, the formula itself will be smaller, and that there will be fewer rules in the automaton. In the worst case, Presburger arithmetic is at least doubly exponential in the size of the formula [18], therefore, it is beneficial for the generated formulae to be as small as possible. Furthermore, the size of the automaton constructed from a Sheaves Logic formula, A, is $d(A)^{h(A)}$; reducing $d(A)$ by optimising the size of the bases in A can have a significant effect on the efficiency.

The first and most obvious method of reducing the size of a common basis is to check whether the two bases are the same; in this case we do not need to explicitly construct a common basis because the basis is already common. For example, consider the basis $(\emptyset^{\perp}[A], \emptyset^{\perp}[B])$. The naive algorithm for constructing a common basis produces $(\emptyset^{\perp}[A \wedge A], \emptyset^{\perp}[A \wedge B], \emptyset^{\perp}[B \wedge A], \emptyset^{\perp}[B \wedge B])$. For the purposes of evaluation, we do not consider checking for a common basis before generating one to be an optimisation.

A more sophisticated method of optimising the size of the basis generated is to only add a new element, $\alpha[A]$, to the basis that is being constructed if $\alpha \neq \emptyset$ and A is satisfiable. A proof that this is sound is provided in appendix A.2.3. For example, the naive common basis of the bases $(\{a\}[\top], \{a\}[\neg\top], \{a\}^{\perp}[\top])$ and $(\{a\}[A], \{a\}[\neg A], \{a\}^{\perp}[\top])$ is $(\{a\}[\top \wedge A], \{a\}[\neg\top \wedge A], \emptyset[\top \wedge \top], \{a\}[\top \wedge \neg A], \{a\}[\neg\top \wedge \neg A], \emptyset[\top \wedge \top], \emptyset[\top \wedge A], \emptyset[\top \wedge \neg A], \{a\}^{\perp}[\top \wedge \top])$. By removing those elements where the set of branch labels is empty, or the sub-formula of the branch is not satisfiable, we construct the following basis: $(\{a\}[\top \wedge A], \{a\}[\top \wedge \neg A], \{a\}^{\perp}[\top \wedge \top])$. Using the optimisations presented in section 6.1, the basis can be simplified to $(\{a\}[A], \{a\}[\neg A], \{a\}^{\perp}[\top])$

The two bases used in the example are the two bases that would need to be combined if we were translating the tree logic formula $a[\mathbf{T}]|a[A]$.

However, to test whether a formula is unsatisfiable, we must evaluate its Presburger constraint. This is a potentially expensive operation and so we must ensure that the cost of testing whether each constraint is satisfiable does not outweigh the benefits of the reduced bases. Generally, in practice, this optimisation has improved performance.

6.3 Reducing the Number of Quantified Variables

After a common basis between two formulae has been found it is often the case that each of the formulae needs to be adjusted to this new basis. Dal Zilio et al show in [8] that given a formula ϕ defined over the basis $\mathbf{E}$, and the (common)

basis $\mathbf{F}$ and relation $\mathcal{R}$ that refines $\mathbf{E}$, we have $[\![\exists \mathbf{N}.\phi \cdot \mathbf{E}]\!] = [\![\exists \mathbf{M}.\phi_{\mathcal{R}} \cdot \mathbf{F}]\!]$, where $\mathbf{M} = (M_1, \ldots, M_q)$, and $\phi_{\mathcal{R}}(\mathbf{M})$ is the constraint:

$$\exists (N_i)_{i \in 1..p}, (X^i_j)_{(i,j) \in \mathcal{R}}. \left(\begin{array}{l} \bigwedge_{j \in 1..q} (M_j = \sum_{(i,j) \in \mathcal{R}} X^i_j) \\ \wedge \bigwedge_{i \in 1..p} (N_i = \sum_{(i,j) \in \mathcal{R}} X^i_j) \\ \wedge \phi \end{array} \right)$$

There are three cases where this formula can be optimised:

- If there is some $(i, j) \in 1..q \times 1..p$ such that $(i, j) \notin \mathcal{R}$, the existentially quantified variable X^i_j will not be used in $\phi_{\mathcal{R}}(\mathbf{M})$. Therefore, we can remove the variable, reducing the number of quantified variables in $\phi_{\mathcal{R}}(\mathbf{M})$.
- If, for some $i \in 1..p$ there exists only one $j \in 1..q$ such that $(i, j) \in \mathcal{R}$, then it is easy to see that $\phi_{\mathcal{R}}(\mathbf{M})$ will contain the clause $M_j = X^i_j$. Because $M_j = X^i_j$ it is safe to substitute the variable M_j for each occurrence of the variable X^i_j. Because X^i_j will be unused in this new formula, we can remove it from the formula entirely, reducing the number of existentially quantified variables.
- Similarly, if, for some $j \in 1..q$ there exists only one $i \in 1..p$ such that $(i, j) \in \mathcal{R}$, then it is easy to see that $\phi_{\mathcal{R}}(\mathbf{M})$ will contain the clause $N_i = X^i_j$. Because $N_i = X^i_j$ it is safe to substitute the variable N_i for each occurrence of the variable X^i_j. Because X^i_j will be unused in this new formula, we can remove it from the formula entirely, reducing the number of existentially quantified variables.

6.4 Unsatisfiable Presburger Formulae

During the construction of Sheaves Logic formulae it is often the case that two Presburger constraints are combined in some way. For example, when translating the Tree Logic formula $A \wedge B$ the constraint $\phi_A \wedge \phi_B$ is created. Because the translation to Sheaves Logic is inductive this constraint may be combined with other constraints in a cumulative process. However, if $\phi_A \wedge \phi_B$ is unsatisfiable it is possible to safely replace the generated Sheaves Logic formula with $\perp$. In addition to preventing the construction of large, unsatisfiable constraints, this substitution will also allow the optimisations introduced in section 6.1 to be used, further reducing the size of the generated formulae.

To implement this optimisation a check has to be made each time two constraints are combined in a way which may result in an unsatisfiable formula. For example, the conjunction of two satisfiable formulae may be unsatisfiable, but the disjunction will always be satisfiable. Therefore, a trade-off exists between the cost of checking the satisfiability of formulae during construction, and the additional cost of evaluating larger constraints with unsatisfiable sub-formulae.

Because, in the worst case, the complexity of Presburger constraints is doubly exponential in the size of the formula, the cost of evaluating two constraints

is significantly lower than the cost of evaluating a single large constraint constructed from the two smaller constraints. Of course, not every formula can be simplified using this technique, so we require that the decrease in run time caused by simplifying the constraints outweighs the increase in time caused by checking each sub-formula for satisfiability.

6.5 Testing Automata for Emptiness

There are two simple optimisations that can be applied to the algorithm for testing whether the language accepted by an automaton is empty. This algorithm is given in section 2.4.7.

- The algorithm can terminate immediately whenever a final state is added to $\mathcal{Q}_{\mathbf{M}}$. This is because the algorithm never removes any elements from the set of reachable states ($\mathcal{Q}_{\mathbf{M}}$). Therefore, if at some point during execution, a final state is added, it will remain in the set until the algorithm terminates. A result of "true" is returned if a final state is reachable, so, as soon as we find a final state that is reachable, there is no need to continue execution.

- When a state is added to either of the sets of reachable states, they remain in that set until the algorithm has terminated. As a result, once a rule has been applied and its result state has been added to the relevant sets, applying the rule a second time will not change the contents of the sets, since its result state will already be contained in them. Therefore, we do not need to re-evaluate the rule, and so we may remove it from the set of rules. This optimisation reduces the number of rules that need to be evaluated during each iteration of the algorithm's main loop. Additionally, we could also remove any rules whose result state is already in the relevant sets, further reducing the number of potentially applicable rules. However, due to the method of automaton construction, this additional case is rare, and consequently it has not been implemented.

Chapter 7

Implementation: Evaluation

In this chapter both a quantitative and qualitative evaluation of the implementation is presented. We begin by evaluating the run-times (and correctness) of several sets of test cases using various degrees of optimisation, followed by an investigation into the effect of the width and depth of a Tree Logic formula on the run-times of the tool. We then evaluate the success of the optimisations and the viability of the approach as a whole. A further section discusses the interface.

To collect the run-times of the tool we used the Linux command, 'time'. The tool was compiled using the optimising compiler, 'ocamlopt'. In section 7.1 the machine used was a 2.6Ghz Intel® Pentium® IV, with 1Gb RAM. In section 7.2 the machine used was a 666Mhz Intel® Pentium® III, with 256Mb RAM.

7.1 Testing

The implementation was tested using three sets of test data, and several levels of optimisation. The first set of test data are simple cases designed to test the different operators of the logic, ensuring that they behave as the semantics prescribe. The second set of test data represent "moderate" cases, that are slightly more complex than the simple cases. The final set of test data are difficult cases where formulae typically have a depth or width of four or more. The moderate and difficult test cases allow us to assess the tool under different situations: a pattern matching tool is likely to use more moderate formulae, whereas a security verifier is likely to use more complicated formulae. The test data are given in full in appendix A.3.

There are two sets of optimisation levels, in addition to no optimisation and all optimisations. The first set — the additive set — measures the performance of each optimisation by disabling all others. For this round of testing optimisation of bases was enabled because, without it, runs either failed due to memory errors, or ran for untenable periods of time. The second set — the subtractive set — of optimisation levels measures the performance of the system with each

Enabled Optimisation	Simple	Moderate	Hard
All optimisations disabled	3.28s	Error [a]	> 30mins [b]
Basis optimisation	0.84s	1.87s	41.74
Quantified variables	0.43s	0.94s	24.29
Extended Sheaves Logic	0.38s	0.90s	17.36s
Unsatisfiable constraints	0.71s	1.46s	29.23
Test for emptiness	0.52s	1.20s	20.48s

[a]The Omega Calculator suffers a stack overflow error when parsing a constraint generated during execution.

[b]After 30 minutes, the test was abandoned

Table 7.1: Run times, additive

Disabled Optimisation	Simple	Moderate	Hard
All optimisations enabled	0.15s	0.33s	3.84s
Quantified variables	0.29s	0.59s	8.05s
Extended Sheaves Logic	0.24s	0.54s	8.82s
Unsatisfiable constraints	0.13s	0.32s	4.00s
Test for emptiness	0.21s	0.44s	6.5s

Table 7.2: Run times, subtractive

optimisation (except basis optimisation) individually disabled. The first set is designed to test how effectively the optimisation reduces run-times, whereas the second set aims to test the optimisation's performance when interacting with the others. The results for each set are given in tables 7.1 and 7.2 respectively. The mean speed-ups observed in each case are given in table 7.3.

Except in those cases where execution terminated abnormally, for each test case and level of optimisation, the results returned by the program were as expected: all tests were passed.

7.2 Width and Depth

In this section we investigate the run-times of the tool when either the width or the depth of the Tree Logic formula is increased. Using this data we can judge where the complexity in the approach lies.

Optimisation	Additive	Subtractive
Quantified variables	1.89	1.93
Extended Sheaves Logic	2.23	1.84
Unsatisfiable constraints	1.30	0.96
Test for emptiness	1.73	1.48

Table 7.3: The average speed up observed for each optimisation

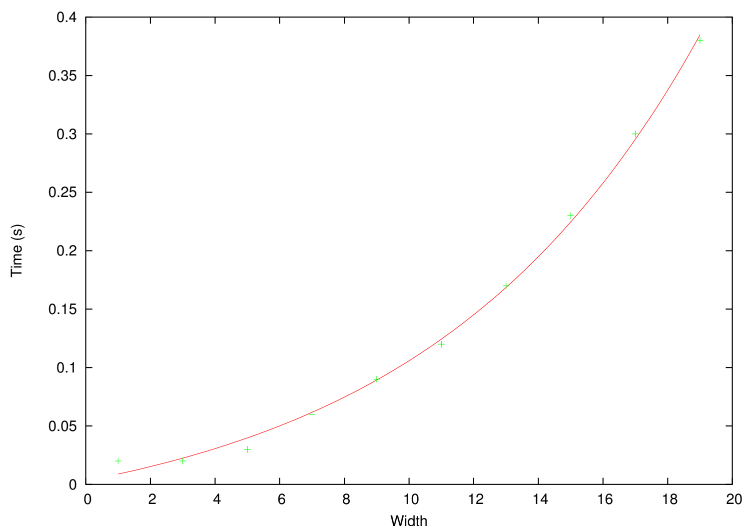


Figure 7.1: The width of a formula against the run-times of the tool

7.2.1 Width

We measure the width of a Tree Logic formula by the number of occurrences of a branch label at a single node in the tree. We tested the effect of the width on the run-time of the tool by checking the validity of the formula $(a[\mathbf{0}])^n$, where $n = 1, 3, 5 \ldots$. When the width of the formula reached 21, the Omega Calculator suffered a segmentation fault. The results of the tests are shown in figure 7.1.

As the graph shows, there is an exponential increase in run-time as the width of the formula increases. This growth, however, is quite gentle, especially when compared with the growth associated with the depth of the formula. This reduced growth rate may occur due to the optimisations that the implementation uses: optimisations such as the basis optimisation occur when two bases are being combined — this is an action that is performed when the composition operator is translated. Consequently, the width of the formula benefits from

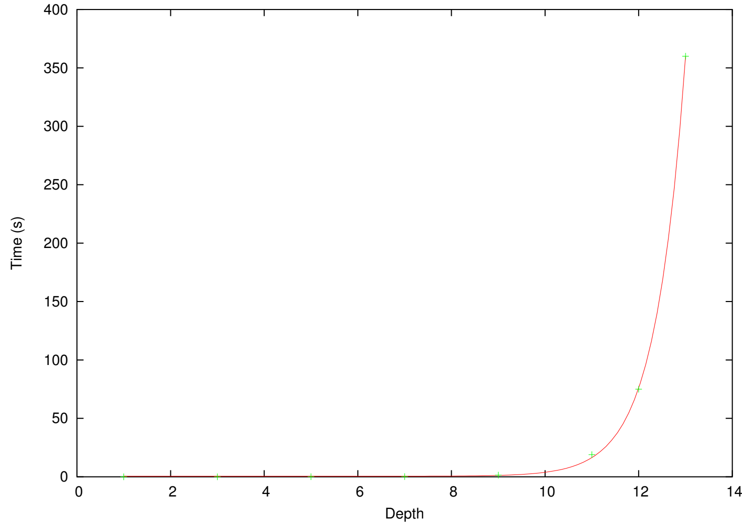


Figure 7.2: The depth of a formula against the run-times of the tool

these optimisations, and so the run-time does not increase rapidly.

7.2.2 Depth

Figure 7.2 plots the run-time of the tool against the depth of a test formula. The formula used was of the form $a[a[\ldots a[a[\mathbf{0}]]]]$, that is, a stack of n branches labeled a, where $n = 1, 3, 5 \ldots$. The tests finish at $n = 13$: at $n = 15$ the run-time became untenable.

The growth rate of the runs-times increases rapidly as n becomes large. The reason for this exponential growth rate is that, at each level in the tree, the element formula $\{a\}[A]$ uses the basis $(\{a\}[A], \{a\}[\neg A], \{a\}^{\perp}[\top])$. This basis has two copies of the sub-formula, A. This occurs at each level in the tree, and so the size of the Sheaves Logic formula increases exponentially. No optimisations that address this problem have been implemented. A possible solution to this problem is discussed in section 7.4.6.

7.3 Viability

Before we can evaluate the viability of the implementation, and the approach in general, we must consider where this tool may be used, and what criteria we require in these situations.

In section 2.2 we identified two applications of a model checking tool similar to the tool implemented for this project. These are, for pattern matching in tree manipulation languages, and for verifying security properties of a network.

Pattern matching will occur during execution of a program, and hence a desirable feature is speed. As testing has shown, with optimisations, simple cases can be evaluated fairly quickly, whereas larger cases have shown an (expected) exponential slow down. However, it is unlikely that the formulae used for pattern matching will be significantly more complex than the moderate test cases. This is because pattern matching usually occurs on a local level with only a few cases. This will mean that validity or satisfaction of these patterns can be checked quite quickly; but, when testing whether a tree, d, matches a type, A, we evaluate the formula, $\underline{d} \Rightarrow A$. It is highly likely that the size of the tree data is quite large, and so the resulting formula will be large and the performance of the tool will not be satisfactory. The performance may be improved significantly if we introduce a further optimisation: using the observation — exploited by Calcagno et al when proving decidability of the Tree Logic [4] — that formulae can only differentiate a limited number of trees, we should be able to use a reduced representation of the original tree as the antecedent in the implication. This may reduce the size of the formulae, in the average case, to an acceptable level.

Also, the Tree Logic as a pattern matching language is inherently non-deterministic. For example, the tree $a[\mathbf{0}]|b[\mathbf{0}]|b[c[\mathbf{0}]]$ can be divided by the formula $a[\mathbf{T}]|b[\mathbf{T}]|b[\mathbf{T}]$ in two different ways. This may or may not be a desirable feature, and a deterministic variant of the Tree Logic may be preferred by the programmer. The deterministic variant may additionally benefit from a lower complexity.

A factor that must also be considered is the availability of alternatives. The Tree Logic is very expressive and may be more expressive than is required for most cases. In this cases a less expressive pattern language — such as the pattern matching available in functional languages like Haskell and ML — may be preferable due to a lower complexity. It is possible that the two alternatives may be combined such that the Tree Logic is only used when its increased power is required.

When verifying security properties, speed remains an important factor, but the conditions under which the tool is used will differ. Unlike pattern matching, verification is a once-only task, and whilst pattern matching requires an immediate response to ensure the efficiency of the programs that use it, verification does not require such expediency. However, the formulae that will be evaluated during verification are likely to be significantly more complex than those evaluated during pattern matching, and the verification of an entire system will involve possibly thousands of formulae or more. Therefore, efficiency is required.

As the test cases have shown, more complex formulae run in significantly longer times. Depending on the size and number of the formulae that need to be checked, these times could quite easily become unacceptable. However, the relatively simple optimisations used for this implementation have shown very encouraging results. It is possible that more serious optimisations that address the underlying approach could produce the increased performance required. Several ideas for further optimisation are discussed in section 7.4.6.

7.4 Optimisations

The optimisations used in the implementation of this project are fairly simple, and yet the performance of the system has benefitted greatly from them. Perhaps most convincingly, the difficult cases that ran for more than 30 minutes when unoptimised, were solved in under four seconds when the optimisations were enabled.

In the following sections we discuss each of the optimisations individually.

7.4.1 Extended Sheaves Logic

The extended Sheaves Logic showed reasonable improvements in run time for both the additive and subtractive tests. In the additive case, the contribution was greater than the subtractive case, which shows that its performance is affected by the other optimisations. Because it is one of the first optimisations to be applied in the flow of the program, it is likely that its absence means a greater scope for efficiency for other optimisations. In the additive tests, this optimisation performed the best (modulo basis optimisation), whereas in the subtractive tests, its performance was one of the best. Therefore, we conclude that this optimisation is beneficial to the implementation.

7.4.2 Basis Optimisation

The results of testing demonstrate than the optimisation of basis construction produces significant gains in performance, and is often essential. For the difficult test cases, the tool without optimisation ran for 30 minutes before the test was abandoned. With the basis optimisation enabled, the difficult test cases were solved within 42 seconds.

The naive algorithm for the construction of a common basis always produces a basis whose number of elements is equal to the number of elements in the original bases multiplied together. As a result, bases can grow very large very quickly.

However, many of the elements in the common bases are unsatisfiable: they represent no trees and are therefore redundant in the basis. These redundant elements can occur for two reasons: an empty label expressions or an unsatisfiable sub-formula. Empty label expressions occur commonly because the translation of Tree Logic formulae predominantly produces label expressions describing a single label. This is easy to see by inspecting the translation of a branch expression, $a[A]$. For this formula the basis contains three elements, two of which have the label expression, $\{a\}$. When there are several different labels and predominantly singleton label expressions, there is a high probability that the intersection of two label expressions is empty. Similarly, in the test cases the majority of the sub-formulae describe disjoint sets of trees; because of this, the conduction of two sub-formulae is likely to be unsatisfiable.

Hence, the size of the common basis produced can often be reduced considerably. This has a cumulative effect as common bases are produced inductively.

Smaller bases mean that their associated Presburger constraint has fewer quantified variables and is smaller in size. This means that they can be evaluated more quickly.

To test whether a the sub-formula of an element formula is satisfiable we must make a call to the Omega Calculator. However, since occurrences of unsatisfiable sub-formulae are quite common, and the performance gains caused by their removal is quite high, the cost of these extra tests is justified.

7.4.3 Quantified Variable Reduction

Although, in the additive cases, quantified variable reduction was did not give the best performance, its contribution remains significant — an average speed up of $\approx$1.9. This optimisation appears to work independently of the other optimisations and the difficulty of the test cases.

7.4.4 Unsatisfiable Presburger Constraints

Although speed ups were observed in the additive cases, in the subtractive cases the program performed better when the unsatisfiable Presburger constraints optimisation was disabled, except when difficult test cases were used. The program's run time was slightly increased when this optimisation was disabled.

The benefits of this optimisation are small, and are likely to be smaller in practical cases. This is because the optimisation relies on unsatisfiable sub-formulae. When the formulae that are being checked are written for a specific purpose, other than testing, the frequency of unsatisfiable sub-formulae is likely to be small. During the construction of common bases, it is quite probable that sub-formulae will be unsatisfiable. However, this case is already checked when optimising the construction of bases.

To fully determine whether this optimisation is justifiable, tests on practical examples must be undertaken.

7.4.5 The Test for Emptiness

In the additive cases, this 'test for emptiness' optimisations did not perform as well as other optimisations in the easy cases, but outperformed many in the difficult cases. This may be due to the exponential growth in the size of the automaton: the improved efficiency caused by the optimisation will be more apparent with larger automata. In the subtractive cases, the increase in efficiency for the harder cases does not appear to occur. This may be because the test for emptiness is the last algorithm in the flow of the program, and hence the optimisations that have already been applied reduce the benefits of this optimisation.

Optimising the test for emptiness, however, is a relatively low cost operation, and provides more than satisfactory results: good speed ups were observed in all cases. There is a small amount of cost associated with the algorithm, and that occurs when checking if a newly reachable state is in the set of finals states.

Because of the method of automaton construction, the set of final states is always a singleton set, and so the cost of this test will be negligible.

7.4.6 Further Optimisations

Whilst effective, the optimisations presented here are quite simple. In this section we discuss several ideas for additional optimisations that have not been implemented as part of this project.

- Rather than translating the Tree Logic to the Sheaves Logic and then to the Recursive Sheaves Logic, a more efficient method may be to translate the Tree Logic directly to the Recursive Sheaves Logic. A direct translation may allow repeated sub-formulae to be represented by a single set of rules, rather than several sets of equivalent rules. Because the set of recursive definitions for a given formula is exponential in size, a single set of rules rather than several may reduce the size of the resulting recursive formula significantly. Another benefit is that a direct translation is likely to reduce the time taken to translate the Tree Logic formula, although this is unlikely to constitute a significant reduction in overall run-time.

- There is an exponential increase in the size the Sheaves Logic formula generated from a Tree Logic formula. Part of the reason for this expansion is that, at each branch of an element formula, $\alpha[A]$, its sub-formula is translated twice in the basis constructed for it. For example, the basis constructed for the element formula, $\alpha[A]$ is $(\alpha[A], \alpha[\neg A], \alpha^{\perp}[\top])$. The basis contains a translation for both A and $\neg A$. Similarly, during the creation of a common basis, we need to construct several different formulae to represent formulae such as $A \wedge B$, $A \wedge \neg B$. The number of these formulae is equal to the product of the number of element formulae in each basis.

  Element formulae in the Recursive Sheaves Logic are of the form $\alpha[X]$, where X is a recursive variable. X is then defined in the set of recursive definitions. If we were translating a Tree Logic formula directly to Recursive Sheaves Logic we may be able to reduce the number of generated definitions significantly by allowing element formula to be of the form $\alpha[A]$, where A is a simple logical formula constructed from the recursive variables and the connectives $\wedge$ and $\neg$.

  With this richer syntax, the basis for the element formula, $\alpha[A]$, will only require one definition of the formula A, whereas, with the original logic, we needed a definition for both A and $\neg A$. Similarly, when constructing a common basis, we only need definitions for A and B, not $A \wedge B$, $A \wedge \neg B$, etc.

  This, however, is a fairly significant change to the theory, and will require the methodology to be updated. Also, the Recursive Sheaves Logic will become more complicated, and the increase in efficiency gained by the optimisation may suffer from a more difficult decision procedure.

- When evaluating an typing problem, that is, is tree P of type A, we test the validity of the formula, $\underline{P} \Rightarrow A$. It is likely that the tree P is larger than the size of the formula A. In this case, it should be possible to reduce P to a smaller tree, such that the validity of $\underline{P} \Rightarrow A$ is preserved. This may reduce the size of the Tree Logic formula that needs to be evaluated. Since the complexity of Dal Zilio et al's decision procedure is doubly exponential in the size of the Tree Logic formula, reducing the size of the formula could yield large gains in efficiency.

Additionally, the method used for communication with the Omega Calculator is a very basic one. A more sophisticated approach involves the use of OCaml's ability to interface with C programs to utilise the Omega Library — a set of library functions for the manipulation of numerical constraints. It is likely that this approach would provide a more efficient method of solving Presburger constraints since they could be stored in the format required by the Omega Library, rather than being stored as a string that must be parsed by the Omega Calculator. This may have prevented the parse errors that occur when the moderate test cases were attempted with no optimisation. However, because our implementation is an experimental tool, the less efficient, but simpler method of communication was favoured.

7.5 Interface

The interface provided by the tool is a very basic, although adequate for experimental purposes. This tool may also be utilised by other programs using a similar method to the one used by the implementation to communicate with the Omega Calculator. Alternatively, an OCaml program may use the solver module to solve Tree Logic formulae directly.

However, complex formulae can quickly become difficult to read. A Graphical User Interface would provide the potential for several features that could make the tool easy to use for a human. For example, the tree structure of the formulae could be represented to aid in the reading of formulae.

7.6 Summary

The aim of implementing this tool was to test the viability of the approach put forward by Dal Zilio et al. We have shown that this approach can reason about trees much more efficiently than previous approaches, and often runs in satisfactory times. The approach is also very receptive to optimisation, and so there is potential for the method to be applied in many situations.

However, the exponential complexity means that the size of the formulae that can be evaluated in reasonable time is restricted. However, we are still able to reason about formulae of a moderate size in satisfactory time and the large potential for further optimisation indicates that the results can be significantly improved.

Due to time constraints many avenues of further work remained unexplored. These are detailed in chapter 9. Despite this, the results of this work are encouraging and set the stage for further investigation.

Chapter 8

Theory: Two New Decision Procedures for the Separation Logic

We have seen that the decision procedure for the Tree Logic presented by Dal Zilio et al in [8] has shown a significant improvement in run-times over the previous decision procedure for the Tree Logic. In this chapter we investigate whether this work can be used to provide similar improvements for heap-like structures.

We begin by introducing the Separation Logic. In section 8.2 we provide a translation of the Separation Logic into the Tree Logic. This means that the tool implemented during this project can be used to provide a decision procedure for the Separation Logic.

In section 8.3 we apply the ideas in Dal Zilio et al's work to the Separation Logic. This leads to a translation from the Separation Logic to First-Order Logic with Equality. Recent independent work by Etienne Lozes has shown that the Separation Logic can be expressed using a fragment of the Separation Logic without multiplicative connectives, that still relies on the notion of a heap. Furthermore, their result does not provide a decision procedure for the Separation Logic, because the construction of the fragment's equivalent formula for a Separation Logic formula assumes an external decision procedure for the Separation Logic. By contrast, the translations into First-Order Logic with Equality presented here removes the notion of a heap from the logic, and provides a decision procedure. These two translation into Classical Logic are compared in more detail in section 8.5. An evaluation of this chapter is given in section 8.6.

8.1 Separation Logic

The Separation Logic [3] allows us to reason about a data structure defined by a stack and a heap. Decidability results have been shown for a sub-language of this logic, which includes the notion of pointers and equality, but not expressions for describing data or universal quantification [10]. It has been shown that the inclusion of universal quantification causes the logic to become undecidable.

The sub-language of [10] is given in table 8.1 and its semantics are given in table 8.2. Separation Logic formulae are interpreted in the following model:

$$\begin{array}{rcl} Val & \triangleq & Loc \cup \{nil\} \\ Stack & \triangleq & Var \rightarrow Val \\ Heap & \triangleq & Loc \rightharpoonup_{fin} Val \times Val \\ State & \triangleq & Stack \times Heap \end{array}$$

where *Loc* is a memory location and $X \rightharpoonup_{fin} Y$ denotes a finite map from X to Y. Additionally, we write $dom(f)$ for the domain of f and $f\#g$ to indicate that f and g have disjoint domains.

Informally, the heap maps expressions (E) to expressions. The value of each expression is ascertained by evaluating the expression with respect to the given stack. Note that, for an assertion, $E \mapsto E_1, E_2$, the semantics require that $dom(h) = \{[\![E]\!]_s\}$. A heap maps locations to values and therefore, E cannot evaluate to nil.

For example, if we have a heap, h, of the form, $(2 \mapsto 1, 3) * (3 \mapsto 1, nil)$, and a stack, s, such that $s(x) = 1, s(y) = 2, s(z) = 3$ and $s(n) = nil$, then (s, h) would satisfy the following formulae, $(z \mapsto x, n) * (y \mapsto x, z)$, $(z \mapsto x, n) * \top$, $(y \mapsto x, z) * (z \mapsto x, n)$, etc. However, if $s(y) = nil$ then $(y \mapsto x, z) * \top$ would not be satisfied since the heap would have to be of the form $\ldots * nil \mapsto 1, 3 * \ldots$. This is not a valid heap as nil is not a location.

The composition adjunct connective, $-\!*$, is analogous to the guarantee connective, $\triangleright$, in the Tree Logic. Intuitively, the formula $\phi -\!* \psi$ is satisfied by a state (s, h) if all heaps h' that satisfy ϕ (given the stack s) and whose domain is disjoint from the domain of h, can be composed with h such that $(s, h * h')$ satisfies ψ.

8.1.1 Size

We denote the size of a Separation Logic formula, ϕ, as $|\phi|$. The complete definition is given in table 8.3. The size of a formula gives a limit on the number of cells that can be in a heap before the formula cannot differentiate between a larger heap and a heap that has a number of cells less than the size of the formula. This property is useful when limiting the number of heaps that we need to consider when determining properties for all heaps — a property that is important for the results in the section 8.1.2.

$E ::=$	Expressions
$x, y, \ldots$	Variables
nil	Nil
$\phi, \psi ::=$	Assertions
$E \mapsto E_1, E_2$	A cell, E, that points to binary heap cell, E_1, E_2
$E = E$	Equality
false	Falsity
$\phi \Rightarrow \psi$	Implication
emp	The empty heap
$\psi * \psi$	Composition
$\psi \mathrel{-\!\!*} \psi$	Composition adjunct

Table 8.1: The syntax of a sub-language of the Separation Logic

$$\llbracket x \rrbracket_s \triangleq s(x)$$
$$\llbracket nil \rrbracket_s \triangleq nil$$

$s, h \models (E \mapsto E_1, E_2)$	iff $dom(h) = \{\llbracket E \rrbracket_s\}$ and $h(\llbracket E \rrbracket_s) = (\llbracket E_1 \rrbracket_s, \llbracket E_2 \rrbracket_s)$
$s, h \models E_1 = E_2$	iff $\llbracket E_1 \rrbracket_s = \llbracket E_2 \rrbracket_s$
$s, h \models$ false	never
$s, h \models \phi_1 \Rightarrow \phi_2$	iff $s, h \models \phi_1$ then $s, h \models \phi_2$
$s, h \models$ emp	iff $dom(h) = \emptyset$
$s, h \models \phi_1 * \phi_2$	iff there exists h_1 and h_2 such that $h_1 \# h_2$; $h_1 * h_2 = h$; $s, h_1 \models \phi_1$ and $s, h_2 \models \phi_2$
$s, h \models \phi_1 \mathrel{-\!\!*} \phi_2$	iff for all h_1 such that $h \# h_1$ and $(s, h_1 \models \phi_1$, $(s, h * h_1) \models \phi_2$

Table 8.2: The semantics of a sub-language of the Separation Logic, given stack, s and heap, h

$\|(E \mapsto E_1, E_2\|$	$=$	1	$\|E_1 = E_2\|$	$=$	0
$\|\text{false}\|$	$=$	0	$\phi \Rightarrow \psi$	$=$	$max(\|\phi\|, \|\psi\|)$
$\|\phi * \psi\|$	$=$	$\|\phi\| + \|\psi\|$	$\|\phi \mathrel{-\!\!*} \psi\|$	$=$	$\|\psi\|$
$\|\text{emp}\|$	$=$	1			

Table 8.3: The size of a Separation Logic formula

$S_{\phi,s}$	The set of values, $s(FV(\phi))$.
v	A value from the set $Loc - S_{\phi,s}$.
$L_{\lvert\phi\rvert}$	A set of $\lvert\phi\rvert$ locations from the set $Loc - S_{\phi,s} - \{v\}$.
$H_{\phi,s}$	The finite set of heaps that need to be enumerated when evaluating an assertion, ϕ, for all heaps.
$H_{\phi,s,L}$	The finite set of heaps whose domain is a subset of $L \cup S_{\phi,s}$ and whose values are restricted set, determined by ϕ and s.
$\mathcal{R}_{\phi,s}(h)$	The reduction of a heap to an equivalent heap in the set $H_{\phi,s}$.
$S_{\lvert\phi\rvert}$	A set of $\lvert FV(\phi)\rvert$ locations.
$\mathcal{S}^{\phi}$	The set of stacks that must be evaluated when determining the validity of an assertion, ϕ.
$H'_{\phi,n}$	The set of heaps where $\lvert dom(h)\rvert \leq n + \lvert S_{\lvert\phi\rvert}\rvert$.
$\mathcal{R}_{\phi}(s,h)$	The reduction of the state, (s,h), to an equivalent state in the set $\mathcal{S}^{\phi} \times H'_{\phi,\lvert\phi\rvert}$.
v	A value from the set $Loc - S_{\phi,s} - L_{\lvert\phi\rvert}$

Table 8.4: A summary of the notations introduced in section 8.1.2

8.1.2 Finite States

The properties discussed in this section are used in both the translation from Separation Logic to Tree Logic, and the translation from Separation Logic to Classical Logic. In both of these translations we define formulae that "enumerate" the states that we need to consider. These formulae are required to ensure that the trees that we are using are representations of valid heaps.

The results discussed in the section are corollaries of the results presented by Calcagno, Yang and O'Hearn in [10]. The results are derived in full in appendix B.1.

In this section we introduce several notations. These notations are summarised in table 8.4. A detailed discussion of these definitions follows.

Satisfaction

When determining if $(s,h) \models \phi$, we can define a finite set of heaps, denoted $H_{\phi,s}$. Any given heap, h, can be reduced to a heap, $h' \in H_{\phi,s}$, such that $(s,h) \models \phi$ iff $(s,h') \models \phi$. We write $\mathcal{R}_{\phi,s}(h)$ to denote the translation of the heap, h, to the equivalent heap, $h' \in H_{\phi,s}$.

We write $S_{\phi,s}$ to denote the set $s(FV(\phi))$, define $L_{\lvert\phi\rvert}$ to be a set of $\lvert\phi\rvert$ locations in $Loc - S_{\phi,s}$, and choose a value, v from the set $Loc - S_{\phi,s} - L_{\lvert\phi\rvert}$. The heaps in the set $H_{\phi,s}$ all have a domain that is a subset of $L_{\lvert\phi\rvert} \cup S_{\phi,s}$ and whose values are in the set $S_{\phi,s} \cup \{nil, v\}$.

For any set of locations, L, we write $H_{\phi,s,L}$ to denote the set of heaps whose

domain is a subset of $L \cup S_{\phi,s}$, and whose values are in the set $S_{\phi,s} \cup \{nil, v\}$. We notice that $H_{\phi,s} = H_{\phi,s,L_{|\phi|}}$.

A consequence of these properties is that, given a stack, s, and assertion, ϕ, we can determine if $s, h \models \phi$ for all heaps by checking that the assertion holds for all heaps in the set $H_{\phi,s,L_{|\phi|}}$.

Additionally, when evaluating an assertion of the form $s, h \models \phi_1 \mathrel{-\!\!*} \phi_2$, where $h \in H_{\phi,s,L}$, the universal quantification can be checked by enumerating another finite set of heaps, $h_1 \in H_{\phi,s,L'}$, where L' is a set of $max(|\phi_1|, |\phi_2|)$ locations from the set $Loc - L - S_{\phi,s} - \{v\}$. The heap, $h * h'$, is then an element of the set $H_{\phi,s,L \cup L'}$.

Validity

We saw in section 8.1.2 that we can determine whether a property holds for all heaps, given a stack, by checking the property for each of a finite set of heaps. Additionally, any given heap could be translated into a heap in this finite set. We now introduce a similar property for stacks.

Given an assertion, ϕ, the sets of stacks that we need to consider to determine the validity of ϕ are those that map all free variables of ϕ to an element of a set $S_{|\phi|} \cup \{nil\}$ and all other variables to nil. The set $S_{|\phi|}$ contains $|FV(\phi)|$ locations from Loc. Only a finite number of stacks, $\mathcal{S}$ have these properties. We can reduce any stack, s, into a stack satisfying these properties.

To determine the validity of an assertion, ϕ, (that is, does ϕ hold for all heaps and stacks), it is enough to check that, for all stacks, $s \in \mathcal{S}$, $s, h \models \phi$ for all $h \in H_{\phi,s}$.

The encoding of Separation Logic as Classical Logic presented in section 8.3 does not require us to restrict the values used in the heaps or stacks, it simply requires that the domain of the heap is of a limited size, and that the stack maps all variables not in the set $FV(\phi)$ to nil. We introduce the notation $\mathcal{S}^{\phi}$ for the set of stacks that map all variables not in $FV(\phi)$ to nil, and $H'_{\phi,n}$ to denote those heaps, h, where $|dom(h)| \leq n + |S_{|\phi|}|$.

We show in appendix B.1.5 that an assertion, ϕ, will holds for all heaps and stacks iff ϕ holds for all states in the set $\mathcal{S}^{\phi} \times H'_{\phi,|\phi|}$.

For an assertion, ϕ, we can reduce any state, (s, h) into, (s', h') that preserves the truth of ϕ, such that $s' \in \mathcal{S}$ (and therefore $s \in \mathcal{S}^{\phi}$), and $h \in H_{\phi,s,L_{|\phi|}}$ (and therefore $h \in H'_{\phi,|\phi|}$). We write $(s', h') = \mathcal{R}_{\phi}(s, h)$ to denote this reduction.

8.2 Translating Separation Logic to Tree Logic

In this section we provide a method of encoding the Separation Logic using the Tree Logic. This means that the tool implemented during this project can be used to provide a new decision procedure for the Separation Logic. It also highlights some of the important differences between the two logics. For example, a location may only appear once in a heap, whereas there is no such uniqueness constraints for trees.

We begin by defining a translation from heaps to trees, and then we define the translation from the Separation Logic to the Tree Logic. It is worth noting at this point that our translation requires a fixed stack. This is because there is no apparent method for representing stacks succinctly in the Tree Logic. Validity can be determined by enumerating all stacks as described in section 8.1.2.

Our translation has the following properties:

- given a formula, ϕ, and a stack, s,

$$\forall h \in H_{\phi,s} \, [(s, h \models \phi \iff heaptran(h) \models tran(\phi, s))]$$

- given a stack, s, and an assertion, ϕ,

$$[\forall h(s, h \models \phi)] \iff [\forall d(d \models tran(\phi, s))]$$

These properties mean that we are able to reduce both satisfaction and validity for the Separation Logic to satisfaction and validity for the Tree Logic. These problems can be solved using the tool that was implemented as part of this project.

It can be noted that the first property only holds for a finite set of heaps. This finite set of heaps is the set of heaps that need to be considered to determine the validity of an assertion. An arbitrary heap can be transformed (using $\mathcal{R}_{\phi,s}$) into an equivalent heap that is within the finite set, and so we can transform any satisfaction problem in the Separation Logic to an equivalent problem in the Tree Logic.

8.2.1 Translating a Heap into a Tree

The translation given in table 8.5 defines the function $heaptran(h)$ that defines a tree representation of a given heap. The function that translates a tree into a heap follows naturally from this definition. If the tree does not represent a valid heap, no translation is defined.

This translation exploits the similarities between the horizontal spatial operators $*$ and $|$, and uses tree branches to model the heap's $\mapsto$ relation. Quite simply, $h_1 * h_2$ is the composition ($|$) of two trees, and a heap of the form, $\ell \mapsto v_1, v_2$ is encoded as a tree, $\ell[v_1[v_2[\mathbf{0}]]]$, of depth three.

$$heaptran(h) \triangleq \begin{cases} \mathbf{0} & \text{if } h = [] \\ \ell[v_1[v_2[\mathbf{0}]]] & \text{if } h = (\ell \mapsto v_1, v_2) \\ heaptran(h_1) | heaptran(h_2) & \text{if } h = h_1 * h_2 \end{cases}$$

Table 8.5: The function *heaptran*

s is the given stack.
$S_{\phi,s}$ is the set $s(FV(\phi))$.
v is a value not in $S_{\phi,s} \cup \{nil\}$.
$L_{|\phi|}$ denotes a set of $|\phi|$ locations in $Loc - S_{\phi,s} - \{v\}$.
$\{v_1, \ldots, v_p\}[D]$ is the Tree Logic formula, $v_1[D] \vee \ldots \vee v_p[D]$

$$
\begin{array}{rcl}
tran(\phi, s) & \triangleq & heap(L, s, \phi) \Rightarrow tran'(\phi, s, \phi, L) \\
 & & \text{where } L = L_{|\phi|} \\
 & & \\
tran'((E \mapsto E_1, E_2), s, \phi, L) & \triangleq & [\![E]\!]_s \left[[\![E_1]\!]_s \left[[\![E_2]\!]_s \left[\mathbf{0} \right] \right] \right] \\
tran'((E_1 = E_2), s, \phi, L) & \triangleq & \begin{cases} \mathbf{T} & \text{if } [\![E_1]\!]_s = [\![E_2]\!]_s \\ \mathbf{F} & \text{otherwise} \end{cases} \\
tran'(\text{false}, s, \phi, L) & \triangleq & \mathbf{F} \\
 & & \\
tran'((\phi_1 \Rightarrow \phi_2), s, \phi, L) & \triangleq & tran'(\phi_1, s, \phi, L) \Rightarrow tran'(\phi_2, s, \phi, L) \\
tran'((\phi_1 * \phi_2), s, \phi, L) & \triangleq & (tran'(\phi_1, s, \phi, L) | tran'(\phi_2, s, \phi, L)) \\
tran'((\phi_1 -\!\!* \phi_2), s, \phi, L) & \triangleq & (tran'(\phi_1, s, \phi, L') \wedge heap(L', s, \phi)) \\
 & & \qquad \triangleright (heap(L \cup L', s, \phi) \Rightarrow tran'(\phi_2, s, \phi, L \cup L')) \\
 & & \text{where } L' \text{ denotes } max(|\phi_1|, |\phi_2|) \text{ locations} \\
 & & \text{from the set } Loc - L - S_{\phi,s} - \{v\} \\
 & & \\
heap(L, s, \phi) & \triangleq & (A_1 \vee \mathbf{0}) | \ldots | (A_p \vee \mathbf{0}) \\
 & & \text{where, for all } i \in 1..p,\ A_i = l_i[vs[vs[\mathbf{0}]]] \\
 & & L \cup S_{\phi,s} = \{l_1, \ldots, l_p\} \\
 & & vs = S_{\phi,s} \cup \{nil, v\}
\end{array}
$$

Table 8.6: Encoding a Separation Logic formula, ϕ, in Tree Logic

For example, the heap, $(1 \mapsto 2, 3) * (2 \mapsto nil, nil)$, is represented by the tree, $1[2[3[\mathbf{0}]]]|2[nil[nil[\mathbf{0}]]]$.

8.2.2 Translating Separation Logic to Tree Logic

In table 8.6 we define an encoding of the Separation Logic in the Tree Logic.

The translation begins by restricting the set of trees with the formula $heap(L, s, \phi)$. $heap(L, s, \phi)$ is designed to accept only those trees that correspond to a heap in the set $H_{\phi,s,L}$. $heap(L, s, \phi)$ is required to ensure that the tree represents a valid heap, preventing duplicate locations and branch structures that do not represent the relation, $\mapsto$.

$heap(L, s, \phi)$ models all trees in $H_{\phi,s,L}$ by composing formulae of the form $(A_i \vee \mathbf{0})$, where the sub-tree, A_i represents a mapping of a unique domain element to the set of all possible values, (vs, vs). All possible subsets of $L \cup S_{\phi,s}$, and

therefore all possible domains, are represented by this formula since each domain element may be in the heap (satisfying A_i), or not (satisfying $\vee\mathbf{0}$). We write $vs[D]$, where $vs = \{l_1, \ldots, l_n\}$ to denote the Tree logic formula, $l_1[D]\vee\ldots\vee l_n[D]$. This notation has parallels with the element formulae used by the Sheaves Logic in section 2.4.2, where element formula are of the form $\alpha[A]$, where α is a set of labels. If we were to translate the Tree Logic formula into Sheaves Logic, we may wish to exploit these similarities to remove the large disjunction represented by $vs[D]$ from the translation.

When testing whether a formula holds for all heaps, we want to ensure that trees that are not heaps do not affect the test, hence the implication in the definition of $tran(\phi, s)$ whose antecedent is $heap(L_{|\phi|}, s, \phi)$. When testing for satisfaction, it is important that the heap satisfies the properties required by $heap(L, s, \phi)$, since failure may result in false positives. Any heap, h, can be reduced to $h' = \mathcal{R}_{\phi,s}(h)$, where $h' \in H_{\phi,s,L}$. h' will satisfy $heap(L, s, \phi)$. It is worth noting that the implication will mean that the formula will be satisfied by any tree that does not represent a valid heap. Consequently, it is not the case that the translation is satisfiable iff the original formula is satisfiable. We may solve this problem by reducing satisfiability to validity, or by replacing the implication with a conjunction.

The translation then passes the set, L, as a parameter. The set of locations that may appear in the domain of the current heap is constructed using this set. Initially this set is $L_{|\phi|}$ because we are initially only interesting in heaps that are in the set $H_{\phi,s,L_{|\phi|}}$. For most operators this set will remain unchanged. However, composition adjunct, $-\!\!*$, requires the enumeration of new heaps whose domain may differ. The right-hand side of this operator requires that the result of composing the current heap and the new heap satisfy the given formula. This means that the domain of the current heap when evaluating this formula may be expanded, and hence, the argument, L, needs to change to reflect this expansion. Composition adjunct will be discussed in detail after we have discussed each operator individually.

The translation provided is inductive, the base cases occur when $\phi = E \mapsto E_1, E_2$, $\phi = (E_1 = E_2)$ and $\phi =$ false. In the case when $\phi =$ false we simply translate the formula to $\mathbf{F}$ — false in the Tree Logic. Because the Tree Logic does not have a notion of equality, and because the value of the formula $E_1 = E_2$ is determined by the stack, regardless of the heap, we evaluate $E_1 = E_2$ during translation, resulting in either $\mathbf{T}$ or $\mathbf{F}$. In case $\phi = E \mapsto E_1, E_2$ we evaluate E, E_1 and E_2 using the given stack, and produce the tree representation of the heap that would satisfy the formula.

There are three inductive cases: $\phi = \phi_1 \Rightarrow \phi_2$, $\phi = \phi_1 * \phi_2$ and $\phi_1 -\!\!* \phi_2$. The translation of $\phi = \phi_1 \Rightarrow \phi_2$ is quite straight forward, but the translations for the remaining cases require some explanation.

The translation of the Separation Logic formula, $\phi = \phi_1 * \phi_2$, is a straightforward translation using the $|$ connective. Although $|$ does not enforce the uniqueness constraints required by $*$, we know that the two heaps and their composition will represent valid heaps because $tran(\phi, s)$ checks that the given heap is valid (and hence its decomposition will be valid), and $tran'(\psi, s, \phi, L)$

ensures that any heaps constructed when translating composition adjunct are valid heaps.

The translation of the formula, $\phi = \phi_1 \mathrel{-\!\!*} \phi_2$ exploits the similarities between the guarantee operator and the magic wand. The formula, $heap(L', s, \phi)$ ensures that the trees that satisfy ϕ_1 are also valid heaps from the set of heaps, $H_{\phi,s,L'}$, that we need to evaluate. The right-hand side of the translation enforces the requirement that the two heaps being composed have disjoint domains by using an implication whose antecedent is $heap(L \cup L', s, \phi)$. If the two heaps do not have disjoint domains, then their composition does not have to satisfy ϕ_2, consequently, we accept the composition trivially (as $heap(L \cup L', s, \phi)$ fails). If the two heaps do have disjoint domains, then $heap(L \cup L', s, \phi)$ will hold and so we must check that the translation of ϕ_2 is satisfied to check whether the assertion holds.

This translation has the following properties:

- given a formula, ϕ, and a stack, s,

$$\forall h \in H_{\phi,s} \left[(s, h \models \phi \iff heaptran(h) \models tran(\phi, s))\right]$$

- given a stack, s, and an assertion, ϕ,

$$[\forall h(s, h \models \phi)] \iff [\forall d(d \models tran(\phi, s))]$$

The proof of these properties is given in appendix B.2. Notice that the first property only ranges over those heaps in the set $H_{\phi,s}$. It is always possible to reduce a heap to a heap in the set $H_{\phi,s}$, using $\mathcal{R}_{\phi,s}$, and so we can check satisfaction for all trees.

These properties mean that, for any heap problem, we can construct an equivalent tree problem that can be solved using the tool implemented during this project.

8.2.3 Complexity of the Translation

The translations given by $tran'(\psi, s, \phi, L)$ are all linear, except for the case when $\psi = \phi_1 \mathrel{-\!\!*} \phi_2$. In this case we require the formula given by $heap(L, s, \psi)$. For each $\mathrel{-\!\!*}$ connective, the size of the set L is increased by $\mathbf{O}(n)$ elements (where n is the length of the formula ϕ). In the worst case, there are n occurrences of composition adjunct, and so the size of L is $\mathbf{O}(n^2)$. $heap(L, s, \psi)$ contains $\mathbf{O}(|L|)$ elements, each of length $\mathbf{O}(n)$, if we allow the Sheaves Logic label sets, rather than singular branch labels. Therefore, the length of $heap(L, s, \psi)$ is $\mathbf{O}(n^3)$. In the worst case, when there are n occurrences of composition adjunct, the size of the translation will be $\mathbf{O}(n^4)$. $tran(\phi, s)$ also requires $heap(L, s, \phi)$, but this does not dominate the complexity.

8.3 Translating Separation Logic to $FOL_=$

The translation from the Separation Logic to the Tree Logic provided a new decision procedure for the Separation Logic. However, this procedure requires the stack to be fixed, and inherits the complexity of the Tree Logic. Unlike trees, heaps are flat structures and so it is reasonable to expect a lower complexity. In this section we take inspiration from Dal Zilio et al's work to provide another decision procedure for the Separation Logic that translates the logic into First-Order Logic with equality[1]. $FOL_=$ is less expressive than Presburger Constraints, and so the complexity of the procedure is inherently lower than the complexity of the translation into the Tree Logic, whose complexity abstracts over the complexity of the Presburger constraints. Additionally, because $FOL_=$ contains quantifiers, we do not need to fix the stack. The resulting $FOL_=$ formula can be evaluated using existing tool for reasoning about $FOL_=$.

In section 8.3.1 we present a new notion of Sheaves that represent states. In section 8.3.2 we use these Sheaves to translate a given assertion into a Presburger constraint, that can be translated into $FOL_=$ using the method described in section 8.3.3.

The translation given in this section has analogous satisfaction and validity properties to to the translation from Separation Logic to Sheaves Logic. These properties are given in full at the end of section 8.3.2.

8.3.1 Translating States to Sheaves

The notion of a Sheaf, $\mathbf{N} \cdot \mathbf{E}$ is described in section 2.4.1. For the Tree Logic, the vector, $\mathbf{N}$, represented the counts of the element formulae in the support vector, $\mathbf{E}$. To represent states, we adapt the notion of a Sheaf to contain three components: $(\mathbf{N}, \mathbf{B}, \mathbf{S})$. The vectors $\mathbf{N}$ and $\mathbf{B}$ are similar to the vectors in an ordinary Sheaf: $\mathbf{N} = (n_1, \ldots, n_n)$ denotes the counts of the cells in $\mathbf{B}$, and $\mathbf{B} = (b_1, b'_1, b''_1, \ldots, b_n, b'_n, b''_n)$ denotes the cells, $(b_i \mapsto b'_i, b''_i)$. The vector, $\mathbf{S} = (v_x, v_y, v_z, \ldots)$, is used to describe the stack. Will represent the locations, l, using the natural numbers, with $nil = 0$. This is a safe assumption since, if the location, 0, is allocated, we can always relocate the cell to an unallocated location, changing any references as required.

For example, the state, (s, h) — where $h = (1 \mapsto 2, 3) * (3 \mapsto nil, 2)$, and $s(x) = 1, s(y) = 3, s(z) = 2$ — may be described using the vectors, $\mathbf{N} = (1, 1)$, $\mathbf{B} = (1, 2, 3, 3, 0, 2)$ and $\mathbf{S} = (1, 3, 2)$. Notice that all counts in $\mathbf{N}$ are one. Because h is a heap, these counts can never be greater than one — else h would not be a function. The counts, however, can be zero. An alternative representation of our example is, $\mathbf{N} = (1, 0, 1)$, $\mathbf{B} = (1, 2, 3, 3, 2, 1, 3, 0, 2)$ and $\mathbf{S} = (1, 3, 2)$. In this case, the value of the center three numbers in $\mathbf{B} = (\ldots, 3, 2, 1, \ldots)$, does not matter, since they do not form a part of the relevant state.

In section 8.1.2, we conclude that, for a given assertion, ϕ, we can transform any state, (s, h) to (s', h') where $|dom(h')| \leq |\phi| + |FV(\phi)|$ and that s' maps the

[1] $FOL_=$ is First Order Logic without relations and with equality. This is essentially the PSPACE-Complete problem, Quantified Boolean Formulae (QBF) [11].

free variables of ϕ to a set of at most $|FV(\phi)|$ values. The following property will also hold: $(s,h) \models \phi$ iff $(s',h') \models \phi$. This means that we can represent such states with the vector $(\mathbf{N}, \mathbf{B}, \mathbf{S})$, where $|\mathbf{N}| = |\phi| + |FV(\phi)|$, $|\mathbf{B}| = 3(|\phi| + |FV(\phi)|)$, and $|\mathbf{S}| = |FV(\phi)|$.

However, the vectors, $\mathbf{N}$, $\mathbf{B}$, are not specific enough to provide the basis of a translation of the Separation Logic to Presburger formulae. In the case of composition adjunct, we can evaluate $s,h \models \phi_1 \mathrel{-\!\!*} \phi_2$ by enumerating all heaps in the set $H'_{\phi,|L'|}$, for some L'. The domain of these heaps is less than $|L' \cup S_{|\phi|}|$ in size. Similarly, because we can use the translation, $\mathcal{R}_\phi$, the domain of h is less than $|L \cup S_{|\phi|}|$ in size, for some L. Recall that the vectors, L, L' and $S_{|\phi|}$ are disjoint. The domain of the heap constructed by the composition of the two heaps yields a heap whose domain is less than $|L \cup L' \cup S_{\phi,s}|$ in size. This requires that the vectors, $\mathbf{N}$ and $\mathbf{B}$ are extended to incorporate the whole of this domain. It is for this reason that we split the sets into two parts: $\mathbf{N_L} \oplus \mathbf{N_S}$ (where $\oplus$ denotes the concatenation of two vectors) and $\mathbf{B_L} \oplus \mathbf{B_S}$ respectively, where $|\mathbf{N_L}| = |L|$, $|\mathbf{N_S}| = |S_{|\phi|}|$, $|\mathbf{B_L}| = 3(|L|)$ and $|\mathbf{B_S}| = 3(|S_{|\phi|}|)$.

Using this notation, heaps in the set $H'_{\phi,|L|}$ will require $\mathbf{B} = \mathbf{B_L} \oplus \mathbf{B_S}$, heaps in the set $H'_{\phi,|L'|}$ will require $\mathbf{B} = \mathbf{B_{L'}} \oplus \mathbf{B_S}$ and the composition of heaps from these sets will require $\mathbf{B} = \mathbf{B_L} \oplus \mathbf{B_{L'}} \oplus \mathbf{B_S}$. The vector $\mathbf{N}$ requires similar construction, except that the two vectors, $\mathbf{N_S}$, must be added together, since they refer to the same set of heap cells.

Table 8.7 gives the definition of $vector_{\phi,p}(s,h)$, that relates states to Sheaves representing that state, for a given assertion, ϕ. A relation is used because, as we have seen, there are often several different ways to represent a particular state as a vector. The subscript, p, denotes the size of the vector, L, and is analogous to the notation $H'_{\phi,|L|}$.

The relation, $vector_{\phi,p}(s,h)$ says that a vector represents a heap if all of the cells included in the vector (that is, $n_j = 1$) are cells that correspond to a particular cell of the heap, and that all cells in the heap have a unique corresponding cell in the vector. Additionally, the relation makes the distinction between cells that will be unique to that heap, and cells that may overlap with cells in one of the heaps considered during the evaluation of a composition adjunct.

8.3.2 Translating Separation Logic to a Sheaves Logic

In table 8.8 we provide a translation from the Separation Logic to a formula of Presburger Arithmetic.

The translation begins by with an implication that ignores all vectors that do not represent a valid state. This is analogous to the translation into the Tree Logic where we used an implication to ignore all trees that do not represent a valid heap. The antecedent of the implication is the formula *bounded*$(\mathbf{N}) \wedge$ *heap*$(\mathbf{N}, \mathbf{B})$. *heap*$(\mathbf{N}, \mathbf{B})$ ensures that locations (b_i) are unique and non-*nil* (not zero) if they are present in the heap. However, this is not enough to ensure a valid heap since the counts in $\mathbf{N}$ may specify the repeated use of a location. The

$(\mathbf{N_L} \oplus \mathbf{N_S}, \mathbf{B_L} \oplus \mathbf{B_S}, \mathbf{S}) \in \mathit{vector}_{\phi,p}(s,h)$
where:
$\mathbf{N_L} = (n_{L_1}, \ldots, n_{L_p})$
$\mathbf{B_L} = (b_{L_1}, b'_{L_1}, b''_{L_1}, \ldots, b_{L_p}, b'_{L_p}, b''_{L_p})$
$\mathbf{N_S} = (n_{S_1}, \ldots, n_{S_{|FV(\phi|)}})$
$\mathbf{B_S} = (b_{S_1}, b'_{S_1}, b''_{S_1}, \ldots, b_{S_{|FV(\phi)|}}, b'_{S_{|FV(\phi)|}}, b''_{S_{|FV(\phi)|}})$
$\mathbf{S} = (v_x, v_y, v_z, \ldots)$ and $FV(\phi) = \{x, y, z, \ldots\}$

iff

There is a subset, J_L of $1..p$, such that:
There is a bijection, $R_L : (dom(h) - s(FV(\phi))) \times J_L$
where $(l, j) \in R_L$ iff:
$n_{L_j} = 1$
$b_{L_j} = l$
$(b'_{L_j}, b''_{L_j}) = h(l)$
and, for all $j \in (1..|\phi| - J_L)$
$n_{L_j} = 0$

There is a subset, J_S of $1..|FV(\phi)|$, such that:
There is a bijection, $R_S : (dom(h) \cap s(FV(\phi))) \times J_S$
where $(l, j) \in R_S$ iff:
$n_{L_j} = 1$
$b_{L_j} = l$
$(b'_{L_j}, b''_{L_j}) = h(l)$
and, for all $j \in (1..|FV(\phi)| - J_S)$
$n_{Lj} = 0$

For each $x \in FV(\phi)$:
$v_x = s(x)$

Table 8.7: Definition of $\mathit{vector}_{\phi,p}(s,h)$

$$tran(\phi)(\mathbf{N},\mathbf{B},\mathbf{S}) \triangleq \left(\begin{array}{l} bounded(\mathbf{N}) \wedge heap(\mathbf{N},\mathbf{B}) \\ \qquad \Rightarrow\ tran'(\phi)(\mathbf{N},\mathbf{B},\mathbf{S}) \end{array}\right)$$

where

$$\begin{array}{l} \mathbf{N} = \mathbf{N_L} \oplus \mathbf{N_S} = (n_1, \ldots, n_p) \\ |\mathbf{N_L}| = |\phi|,\ |\mathbf{N_S}| = |FV(\phi)| \\ \mathbf{B} = \mathbf{B_L} \oplus \mathbf{B_S} = (b_1, b'_1, b''_1, \ldots, b_p, b'_p, b''_p) \\ |\mathbf{B_L}| = 3|\phi|,\ |\mathbf{B_S}| = 3|FV(\phi)| \\ \mathbf{S} = (v_x, v_y, v_z, \ldots) \text{ and } \{x, y, z, \ldots\} = FV(\phi) \\ p = |\phi| + |FV(\phi)| \end{array}$$

$$tran'(E \mapsto E_1, E_2)(\mathbf{N},\mathbf{B},\mathbf{S}) \triangleq \bigvee_{i \in 1..|\mathbf{N}|} \left[\begin{array}{l} n_i = 1 \wedge \bigwedge_{\substack{j \in 1..|\mathbf{N}| \\ i \neq j}} [n_j = 0] \\ \wedge\ b_i = var(E) \\ \wedge\ b'_i = var(E_1) \wedge b''_i = var(E_2) \end{array}\right]$$

$$\begin{array}{rcl} tran'(E_1 = E_2)(\mathbf{N},\mathbf{B},\mathbf{S}) & \triangleq & var(E_1) = var(E_2) \\ tran'(\phi_1 \Rightarrow \phi_2)(\mathbf{N},\mathbf{B},\mathbf{S}) & \triangleq & tran'(\phi_1)(\mathbf{N},\mathbf{B},\mathbf{S}) \Rightarrow tran'(\phi_2)(\mathbf{N},\mathbf{B},\mathbf{S}) \\ tran'(\text{emp})(\mathbf{N},\mathbf{B},\mathbf{S}) & \triangleq & \bigwedge_{i \in 1..|\mathbf{N}|} n_i = 0 \\ tran'(\text{false})(\mathbf{N},\mathbf{B},\mathbf{S}) & \triangleq & \mathbf{F} \end{array}$$

$$tran'(\phi_1 * \phi_2)(\mathbf{N},\mathbf{B},\mathbf{S}) \triangleq \exists \mathbf{N_1}, \mathbf{N_2}. \left(\begin{array}{l} \mathbf{N} = \mathbf{N_1} + \mathbf{N_2}\ \wedge \\ tran'(\phi_1)(\mathbf{N_1},\mathbf{B},\mathbf{S}) \\ \wedge tran'(\phi_2)(\mathbf{N_2},\mathbf{B},\mathbf{S}) \end{array}\right)$$

$$tran'(\phi_1 -\!\!* \phi_2)(\mathbf{N},\mathbf{B},\mathbf{S}) \triangleq \forall \mathbf{M}, \mathbf{B_{L'}}. \left(\begin{array}{l} tran'(\phi_1)(\mathbf{M}, \mathbf{B_{L'}} \oplus \mathbf{B_S}, \mathbf{S}) \\ \wedge\ bounded(\mathbf{M_S} + \mathbf{N_S}) \wedge\ bounded(\mathbf{M}) \\ \wedge\ heap\left(\begin{array}{l} \mathbf{N_L} \oplus \mathbf{M_{L'}} \oplus (\mathbf{N_S} + \mathbf{M_S}), \\ \mathbf{B_L} \oplus \mathbf{B_{L'}} \oplus \mathbf{B_S} \end{array}\right) \\ \quad \Rightarrow tran'(\phi_2)\left(\begin{array}{l} \mathbf{N_L} \oplus \mathbf{M_{L'}} \oplus (\mathbf{N_S} + \mathbf{M_S}), \\ \mathbf{B_L} \oplus \mathbf{B_{L'}} \oplus \mathbf{B_S}, \\ \mathbf{S} \end{array}\right) \end{array}\right)$$

where

$$\begin{array}{l} \mathbf{M} = \mathbf{M_{L'}} \oplus \mathbf{M_S} \\ |\mathbf{M_{L'}}| = max(|\phi_1|, |\phi_2|), |\mathbf{M_S}| = |FV(\phi)| \\ |\mathbf{B_{L'}}| = 3(max(|\phi_1|, |\phi_2|)) \end{array}$$

$$bounded(n_1, \ldots, n_q) \triangleq \bigwedge_{i \in 1..q} [0 \leq n_i \leq 1]$$

$$heap(\mathbf{N},\mathbf{B}) \triangleq \left(\begin{array}{l} \bigwedge_{\substack{i \in 1..|\mathbf{N}| \\ j \in 1..|\mathbf{N}| \\ i \neq j}} [(n_i = 1 \wedge n_j = 1) \Rightarrow (b_i \neq b_j)] \\ \wedge \bigwedge_{i \in 1..|\mathbf{N}|} [(n_i = 1) \Rightarrow (b_i \neq 0)] \end{array}\right)$$

$$var(E) \triangleq \begin{cases} v_E & \text{if} E \in FV(\phi) \\ 0 & \text{if} E = nil \end{cases}$$

Table 8.8: Encoding a Separation Logic formula, ϕ, as a Presburger formula

formula, $\mathit{bounded}(\mathbf{N})$ is used to ensure that each location only occurs once in the heap, if it occurs at all. When testing for satisfaction, $\mathit{vector}_{\phi,|\phi|}(s,h)$ will always satisfy these constraints because we restrict the states to the required set — the reduction $\mathcal{R}_\phi(s,h)$ can be used if the state does not satisfy the required properties; consequently $\mathit{tran}'(\phi)$ must be satisfied for the state to satisfy ϕ. When checking validity, the antecedent ensures that we ignore all those vectors that are not heaps.

The simplest cases in this translation are $\phi = \text{emp}$, $\phi = (E_1 = E_2)$ and $\phi = \text{false}$. The translations of $\phi = (E_1 = E_2)$ and $\phi = \text{false}$ are direct: false translates to $\mathbf{F}$ and $E_1 = E_2$ translates to $\mathit{var}(E_1) = \mathit{var}(E_2)$, where $\mathit{var}(E)$ is the variable in the vector corresponding to the stack variable denoted by expression, E, or 0 if $E = \mathit{nil}$. In case $\phi = \text{emp}$ we assert that all multiplicities in the vector $\mathbf{N}$ are zero. That is, the heap contains no cells.

The final base case, where $\phi = (E \mapsto E_1, E_2)$ is a little more complicated. The translation of this case requires that one, and only one member of the vector $\mathbf{N}$ is one, and that the others are zero. This means that the heap will have exactly one cell. Furthermore, the cell that is present in the heap must map the value of E ($\mathit{var}(E)$) to the values of E_1 and E_2 ($\mathit{var}(E_1)$ and $\mathit{var}(E_2)$ respectively). Therefore, the vectors accepted will represent a heap that satisfies $(E \mapsto E_1, E_2)$.

There are three recursive cases. The simplest is $\phi = \phi_1 \Rightarrow \phi_2$. In this case the translation is a simple implication using the translations of ϕ_1 and ϕ_2.

In case $\phi = \phi_1 * \phi_2$ we specify the requirement that $h = h_1 * h_2$ for some h_1, h_2, using the clause, $\mathbf{N} = \mathbf{N_1} + \mathbf{N_2}$ for some $\mathbf{N_1}$, $\mathbf{N_2}$. This clause splits the heap into two since, for the clause to hold, if an element of $\mathbf{N}$ is one, then the corresponding element in $\mathbf{N_1}$ or $\mathbf{N_2}$ must be one, but not both. Similarly, if the element is zero, it must also be zero in $\mathbf{N_1}$ and $\mathbf{N_2}$. Each cell in $\mathbf{B}$ is only in the heap if its associated element in the multiplicities vector is one. To determine whether $h_i \models \phi_i$ (for $i \in \{1,2\}$) we use the clause, $\mathit{tran}'(\phi_i)(\mathbf{N_i}, \mathbf{B}, \mathbf{S})$.

The final and most involved case is $\phi = \phi_1 \mathrel{-\!\!*} \phi_2$. In this case we enumerate all $h_1 \in H'_{\phi,|L'|}$, check whether they satisfy ϕ_1 and that $h \# h_1$ holds, where h is the heap represented by the arguments to tran'. If h_1 satisfies these properties, we must then check whether $h * h_1$ satisfies ϕ_2.

So, we begin by enumerating the required h_1s. The size of the domain of h_1 will be less than $|L' \cup S_{|\phi|}|$. So, the vector, $\mathbf{B'}$, used to represent h_1 will contain a vector of new variables, $\mathbf{B_{L'}}$, denoting $|L'|$ new cells, and the vector, shared by h, $\mathbf{B_S}$, which denotes those cells representing $S_{|\phi|}$. The vector $\mathbf{M}$ contains a sub-component for each of the components of $\mathbf{B'}$. We use $\mathit{bounded}(\mathbf{M}) \wedge \mathit{heap}(\mathbf{N_L} \oplus \mathbf{M_{L'}} \oplus (\mathbf{N_S} + \mathbf{M_S}), \mathbf{B_L} \oplus \mathbf{B_{L'}} \oplus \mathbf{B_S})$ to ensure that h_1 is a heap and that the domains of $\mathbf{B_L}$, $\mathbf{B_{L'}}$ and $\mathbf{B_S}$ are disjoint. To ensure that $h \# h_1$ holds we notice that the domains are disjoint except for $S_{|\phi|}$. So, we use the clause $\mathit{bounded}(\mathbf{M_S} + \mathbf{N_S})$. If $\mathbf{M_S} + \mathbf{N_S}$ is bounded then we know that h and h_1 do not use any of the same cells, and so their domains must be disjoint.

Finally, we require that $s, h * h_1 \models \phi_2$. The size of the domain of $h * h_1$ is less than $|L \cup L' \cup S_{|\phi|}|$, and so we construct the vector representation of $h * h_1$

in a similar fashion: using the cells, $\mathbf{B_L} \oplus \mathbf{B_{L'}} \oplus \mathbf{B_S}$, and the corresponding vector of multiplicities. We then check whether the translation of ϕ_2 accepts the resulting vector.

The translation has the following properties that are analogous to the properties possessed by the translation from the Separation Logic to the Tree Logic presented in section 8.2.2:

- Given an assertion, ϕ,

$$\forall(s,h) \in \mathcal{S}^{\phi} \times H'_{\phi,|\phi|} \left[(s,h \models \phi \iff \forall v \in \mathit{vector}_{\phi,|\phi|}(s,h)\ v \models \mathit{tran}(\phi))\right]$$

- For an assertion, ϕ,

$$[\forall(s,h).s,h \models \phi] \iff [\forall(\mathbf{N},\mathbf{B},\mathbf{S}).(\mathbf{N},\mathbf{B},\mathbf{S}) \models \mathit{tran}(\phi)]$$

The proof of these properties is given in appendix B.3.1.

8.3.3 Translating Separation Sheaves Logic to $FOL_=$

The Presburger constraints derived in section 8.3.2 do not need to use the full expressive power of Presburger Arithmetic. In particular, we only use connectives that are not available in $FOL_=$ in two situations. Firstly, we use the $\leq$ connective to restrict the elements of the vector, $\mathbf{N}$, between zero and one. We can let the elements of $\mathbf{N}$ range over boolean atoms rather than natural numbers, meaning that we do not need $\leq$. Because the members of $\mathbf{N}$ are boolean, the addition present in the translation of $*$ can be replaced with the following formula, $(n_i \iff n_{1i} \vee n_{2i}) \wedge \neg(n_{1i} \wedge n_{2i})$, for all $i \in 1..|\mathbf{N}|$. Finally, clauses of the form $n = 1$ can be replaced with n, and $n = 0$ can be replaced with $\neg n$.

By ranging the elements of the vector $\mathbf{N}$ over boolean values, removing clauses of the form, *bounded*($\mathbf{N}$), and replacing the equality clauses for the boolean variables, the translation from Separation Logic to Presburger Arithmetic can be reduced to a logic containing propositional logic, quantification over the Boolean values, quantification over a finite domain of integers, and equality between these integers. This is First-Order Logic with Equality.

8.3.4 Complexity of the Translation

The translation into $FOL_=$ will generate a formula whose size is $\mathbf{O}(n^5)$ where n denotes the length of the Separation Logic assertion (not the size). This can be seen because, for each connective, the size of the vector (initially $\mathbf{O}(n)$) may increase by $\mathbf{O}(n)$ in the worst case (the $-\!\!*$ connective). Therefore, the size of the vector is always $\mathbf{O}(n^2)$. The translation of $E \mapsto E_1, E_2$ and *heap*($\mathbf{N},\mathbf{B}$) are $\mathbf{O}(v^2)$, where v is the size of the vector. So, these formulae are $\mathbf{O}(n^4)$. In the worse case, $\mathbf{O}(n)$ of these cases will occur, and therefore, the resulting $FOL_=$ formula will be $\mathbf{O}(n^5)$ in size.

$FOL_=$ is PSPACE-complete [11], and so the full decision procedure for the Separation Logic is PSPACE-complete. We do not hope to improve on

ϕ ::=	Assertion
$\phi \Rightarrow \phi$	implication
$\perp$	falsity
$x \hookrightarrow e_1, e_2$	allocation
$x = y$	equality
size $\geq n$	size

Table 8.9: A classical fragment of the Separation Logic

$(x \hookrightarrow e_1, e_2)$	$\triangleq$	$(x \mapsto e_1, e_2) * \top$
size $\geq n$	$\triangleq$	$\neg$emp $* \ldots * \neg$emp (n times)

Table 8.10: Classical connectives Separation Logic encoding

this complexity because we know that the Separation Logic is also PSPACE-complete [10].

8.4 A Classical Fragment of the Separation Logic

Recent independent work by Etienne Lozes defines a classical fragment of the Separation Logic and shows that this fragment is as expressive as the Separation Logic itself [9]. In this section we give an overview of this work. In section 8.5, we present a comparison of Lozes' work and the translation into $FOL_=$.

We begin by introducing the classical fragment of the Separation Logic, CL, and describing several properties that are used to show the expressivity of this fragment. Then we show how the Separation Logic can be expressed using CL.

8.4.1 CL

The classical fragment of the Separation Logic is given in table 8.9. This fragment has two new connectives: $(x \hookrightarrow e_1, e_2)$ and size $\geq n$. The encoding of these connectives in Separation Logic is shown in table 8.10.

Intuitively, $(x \hookrightarrow e_1, e_2)$, asserts that the location indicated by the variable x (with respect to the current stack) is allocated on the heap, and that its value matches the expression, (e_1, e_2). size $\geq n$ simply states that the heap has n cells. We write $w(\phi)$ to denote the largest n such that size $\geq n$ is a sub-assertion of ϕ.

In CL, expressions of the form, $(x = y)$, $(x \hookrightarrow e_1, e_2)$ or size $\geq n$ are considered atomic. The Separation Logic can be expressed using boolean combinations

of these atoms. The atoms themselves, however, cannot be encoded using only boolean combinations.

8.4.2 Intensional Equivalence

Given a finite set of variables, X, and an integer, w, we say that two states, σ, σ', are intensionally equivalent, if, for all CL assertions, ϕ, such that $FV(\phi) \subseteq X$ and $w(\phi) \leq w$, $\sigma \models \phi$ iff $\sigma' \models \phi$. We write $\sigma \approx_{X,w} \sigma'$ to denote this equivalence.

For a given assertion ϕ, we can define $X = FV(\phi)$ and $w = |\phi| + |X|$. Given two stores, σ, σ', if $\sigma \approx_{X,w} \sigma'$, then $\sigma \models \phi$ iff $\sigma' \models \phi$.

There are a finite number of classical atoms, p, such that $w(p) \leq w$ and $FV(p) \subseteq X$. This is easily seen. For example, consider the case where $X = \{x\}$ and $w = 1$. There are only two assertions of the form size $\geq n$ (when $n = 0$ and when $n = 1$), only four assertions of the form, $x = y$ or $x \hookrightarrow y, z$ (there are four possible pairs using the variable x and *nil* and x must appear on the left-hand side of the allocation connective). We write $\Phi_{X,w}$ to denote this finite set of atoms.

Consequently, given an assertion, ϕ, of the Separation Logic, there are only a finite number of stores such that no two stores are intensionally equivalent (defining $X = FV(\phi)$ and $w = |\phi| + |X|$). This follows from the finiteness of $\Phi_{X,w}$: intentionally equivalent stores will satisfy the same atomic assertions in $\Phi_{X,w}$. There are less that $2^{|\Phi_{X,w}|}$ subsets of $\Phi_{X,w}$, and therefore, there are a finite number of sets of intensionally equivalent stores.

8.4.3 Characteristic Formulae

The infinite set of all stores can be sub-divided into a finite number of sets (equivalence classes) containing all those stores that are intentionally equivalent, given a set of variables, X and an integer, w. These sets can be defined by a characteristic formula. Given any state, σ, $F_\sigma^{X,w}$ is the characteristic formula, such that:

$$\forall \sigma' \left[\sigma' \models F_\sigma^{X,w} \iff \sigma \approx_{X,w} \sigma'\right]$$

That is, a state, σ', satisfies $F_\sigma^{X,w}$ iff it is intentionally equivalent to σ.

We can define:

$$F_\sigma^{X,w} \triangleq \bigwedge_{\substack{\sigma \models \phi \\ \phi \in \Phi_{X,w}}} (\phi) \wedge \bigwedge_{\substack{\sigma \not\models \phi \\ \phi \in \Phi_{X,w}}} (\neg\phi)$$

This equations uses the observation that intentionally equivalent stores satisfy the same atomic assertions in $\Phi_{X,w}$. Therefore, if σ' is intentionally equivalent to σ it will satisfy all those atomic assertions that σ satisfies, and the negation of those that it does not satisfy. Conversely, if it does not satisfy the same atomic assertions, it will not be intensionally equivalent (by definition).

8.4.4 Expressing the Separation Logic in CL

There are a finite number of equivalence classes, for an assertion, ϕ, of the Separation Logic (defining X and w as described in section 8.4.2). We write $\mathbf{State}_{/\approx_{X,w}}$ to denote this set of classes.

We can then construct a classical assertion equivalent to the assertion, ϕ, by constructing the formula:

$$\bigvee_{\substack{C \in \mathbf{State}_{/\approx_{X,w}} \\ C \models \phi}} F_C^{X,w}$$

Intuitively, this formula works by enumerating all states that satisfy the given formula. The assertion is then constructed from a disjunction of the characteristic formulae for different states that satisfy the assertion, ϕ.

8.5 Comparison with CL

In this section we compare our encoding of the Separation Logic in $FOL_=$ with Etienne Lozes' classical fragment, CL, of the Separation Logic.

The first difference is the Logic into which the Separation Logic is translated. CL is a classical fragment of the Separation Logic and is dependent on the notion of a heap: it contains atomic assertions, $(x \hookrightarrow e_1, e_2)$ and size $\geq n$. This means that an automated decision procedure using this method will require a tool that reasons about heap structures. Such tools are not as developed as tools that solve problems in standard $FOL_=$. $FOL_=$ is a widely used logic and tools exist that are able to evaluate $FOL_=$ formulae.

Another important difference occurs when translating a Separation Logic assertion. The construction of its CL equivalent requires the evaluation of the assertion against an exponential number of states. This will require an external tool that can decide the logic. Conversely, the translation into $FOL_=$ does not require the evaluation of the original assertion.

These two differences lead to an important distinction between the two results: the translation into $FOL_=$ provides a decision procedure for the validity and satisfaction problems of the Separation Logic. A given assertion can be translated into $FOL_=$ without an external procedure that evaluates Separation Logic assertions, and several efficient tools are available to check the resulting $FOL_=$ formula.

The complexity of the two approaches is also worth considering. The translation into CL generates a formula from an exponential number of characteristic formulae, which are polynomial in size ($\Phi_{X,w}$ can contain $\mathbf{O}(w)$ size atoms, and $\mathbf{O}(|X|^3)$ allocation atoms). The subsets of $\Phi_{X,w}$ represent the sets of intentionally equivalent states. For each of these classes we must determine whether its characteristic formula is included in the translation into CL. This means that we must evaluate the original Separation Logic assertion an exponential number of times during the translation. Therefore, the resulting formulae is exponential in size, requiring a call to a Separation Logic checking tool an exponential

number of times, finishing with the evaluation of the CL formula. Conversely, the translation into $FOL_=$ will generate a formula whose size is $\mathbf{O}(n^5)$ where n denotes the length of the Separation Logic assertion (not the size). $FOL_=$ can be easily evaluated using existing tools that can evaluate $FOL_=$ formulae.

It can be noted that the translation to CL only has quantifiers on the outside of the formula. However, the translation into $FOL_=$ does not have this property.

In summary, the translation into CL provides an important theoretical result (that the spatial connectives, $*$ and $-\!\!*$, can be eliminated from the Separation Logic). The translation into $FOL_=$ shows the further result that a classical logic, without the notion of heaps, can express the Separation Logic, and also provides an effective decision procedure.

8.6 Evaluation

In this chapter we have shown two main results: the Separation Logic is expressible in the Tree Logic, and that the Separation Logic can be expressed in $FOL_=$.

The translation from Separation Logic to Tree Logic highlighted some important differences between the two logics. Firstly, we were unable to represent stacks adequately in the Tree Logic, and so, the test for validity requires the external enumeration of all stacks. To describe stacks in the Tree Logic we would have required existential quantification. For example, a translation of the Separation Logic assertion, $(x \mapsto x, x)$, that does not depend on an available stack, may yield the Tree Logic formula, $x[x[x[\mathbf{0}]]]$. To check the validity of this formula we require the formula to hold for all values of x; that is, $\forall x.(x \mapsto x, x)$. The Tree Logic with quantifiers has been shown to be undecidable [12].

A further distinction between the two logics lies in the semantics of composition. In the Separation Logic, the composition connective is a partial connective: not all heaps can be composed. Composition in the Tree Logic, however, is a total connective. As a result, we need to enforce explicitly the conditions under which two heaps can be composed. These conditions add to the complexity of the translation, although not significantly.

To enforce that the trees we consider represent valid heaps, we need to enumerate all of the possible values a location may take. This requires a large disjunctive formula of depth two. However, we can take advantage of the Sheaves Logic to reduce this formula to a single branch of depth two. This is because the Sheaves Logic uses sets rather than singleton labels for element formulae. This observation allows us to avoid the potentially huge Sheaves Logic formula resulting from the translation of the disjunctive formula, and instead construct a much smaller formula, where the set of possible values is the set of labels on an element formula in the Sheaves Logic. Additionally, the same sub-formula is used several times to represent all values at all locations. We may exploit the benefits of the Recursive Sheaves Logic by using the same recursive variable beneath each location branch, rather than many copies of the formula representing any value.

We then provided a second new decision procedure for the Separation Logic, taking Dal Zilio et al's approach as our inspiration. This procedure encodes the Separation logic in $FOL_=$. A benefit of this result over the translation into Tree Logic is that this encoding is able to express stacks, and so the enumeration of an exponential number of stacks is not required when determining validity. Additionally, the complexity of evaluating the $FOL_=$ formulae is significantly lower that the complexity of evaluating Tree Logic formulae: $FOL_=$ is less expressive than Presburger Arithmetic, and the complexity of the Tree Logic is doubly exponential when abstracting over the cost of Presburger Arithmetic.

Due to time constraints, an implementation of this new decision procedure for the Separation Logic has not been attempted. In chapter 9 an implementation is offered as a potential extension to the project. The resulting tool could then be used to evaluate the decision procedure against the existing decision procedure that involves the enumeration of all heaps and stacks that need to be considered when evaluating an assertion.

The translation of the Separation Logic into $FOL_=$ presented in this chapter differs from Dal Zilio et al's work in several ways. One such difference is the absence of support vectors. The translation into $FOL_=$ does not require a support vector that contains the element formulae required to express the tree structures, nor does it require automata to evaluate the formula. This is an expected result since heaps are flat structures, and the support vectors were required to express depth.

Another difference is that the resulting formulae required a smaller logic than the Presburger Arithmetic. This is because we only needed counts of zero and one — unlike the Sheaves Logic, where the element counts are unbounded. This meant that the addition in the translation from Separation Logic to Presburger Constraints could be replaced with boolean connectives. The counts range over zero and one because of the uniqueness constraints: a count of greater than one indicates that a location may occur more than once. This violates the requirement that locations are unique.

However, because counts can only range over zero and one, we lose the ability to denote a data structure of any size. In the Sheaves Logic, the notion of a basis was used. A basis describes the complete set of elements, and, therefore, we can express any tree as a set of counts of the element formulae in the basis. When the uniqueness constraints are introduced, this property is lost. To represent a heap of size n, we require $\mathbf{O}(n)$ elements in its vector translation. We were able to use the size of the given assertion to limit the size of the heaps that needed to be considered.

Chapter 9

Conclusions and Future Work

In this chapter we present the conclusions gained from the work described in the preceding chapters. We finish by describing several areas of future work that follow from this project.

9.1 Conclusions

The primary objective of this project was to test, via a prototypical implementation, the viability of Dal Zilio et al's decision procedure for the Tree Logic [8]. A naive implementation of Dal Zilio et al's work was shown to perform poorly: the doubly exponential complexity meant that only the simplest of formulae could be evaluated satisfactorily. However, we were able to show significant improvements through the use of several optimisations, reducing a run-time exceeding 30 minutes to less than four seconds. We were also able to identify several further optimisations that may reduce this time further.

We have observed that the tool performs well on several examples, but that the run-times increase exponentially, meaning that large formulae cannot be solved satisfactorily. Through testing we have shown that the depth of a formula contributes significantly to the run-time of the tool. We have identified that this slow-down may be caused by the repetition inherent in the Sheaves Logic, and that the Recursive Sheaves Logic may be extended to reduce this repetition. This is a good opportunity for future work.

Further to the implementation of Dal Zilio et al's decision procedure, we investigated the applications of Dal Zilio et al's work to the Separation Logic. The results of this work are two new decision procedures for the Separation Logic. The first procedure encodes the logic in the Tree Logic, and the second encodes the logic in First-Order Logic with equality.

The first decision procedure is a translation from the Separation Logic to the Tree Logic. This enables us to easily extend the functionality of the tool

produced during this project to include the evaluation of satisfaction and validity for the Separation Logic. Furthermore, by providing a translation from the Separation Logic to the Tree Logic we were able to identify several difficulties in representing unique locations in the Tree Logic. We found that we needed to exploit the size of a formula to be able to enforce the uniqueness constraints, and that we had to enumerate the assignment of values to the location variables externally.

The second translation uses the ideas in Dal Zilio et al's work to produce a novel decision procedure for the Separation Logic. This decision procedure encodes the Separation Logic in simple First-Order Logic with Equality and does not require the additional enumeration of variable assignments. This translation also produced the new result that the Separation Logic is expressible in $FOL_=$.

Recent independent work by Etienne Lozes [9] shows that the Separation Logic can be expressed in a classical fragment of the logic, which has as atoms the size of a heap, equality of stack variables and the values stored at locations in the heap. Our translation from the Separation Logic to $FOL_=$ has several advantages over this work: the translation to $FOL_=$ does not rely on the notion of a heap, and does not require an external decision procedure for the Separation Logic.

9.2 Future Work

There are many avenues of work that follow from this project. We outline some of the possibilities below:

- The Kleene Star — in section 2.4.11 we discuss the Kleene Star: a modality that was not implemented as part of this project. An extension to the project is to extend the functionality of the tool produced during this project to include the evaluation of the Kleene Star.

- Further optimisations — in section 7.4.6 we present several potential optimisations that may increase the efficiency of the tool produced during this project. Following on from this project, the full development, implementation and evaluation of these optimisations may be attempted. Additionally, we may wish to look for extra optimisations that have not been identified in this report.

- Separation Logic to Tree Logic — in section 8.2 we provide a translation from the Separation Logic to the Tree Logic. We may wish to implement this translation, utilising the tool implemented for this project to evaluate the resulting Tree Logic formula. We may then wish to compare this method of deciding the Separation Logic with existing decision procedures.

- Separation Logic to $FOL_=$ — in section 8.3 we reduce the Separation Logic to First-Order Logic with Equality. An implementation of this decision procedure, using an external tool to evaluate $FOL_=$, will allow us to

determine the viability of the procedure and compare it with existing decision procedures for the Separation Logic.

- 'Trees with pointers' — a 'trees with pointers' model brings together the heap-model and the tree-model by adding unique location identifiers and cross-references (XML idrefs) to the Tree Logic. It is probable that the translation from the Separation Logic to $FOL_=$ may be extended to this model by re-introducing the notion of a basis from Dal-Zilio et al's work.

  However, the 'trees with pointers' model is not enough to provide a Hoare logic [19] for a tree update language. The Context Logic [20] provides a solution to this problem through the introduction of 'contexts', that represent trees with 'holes'. This 'hole' may be created when deleting a sub-tree at a location specified by a variable whose value can only be known at run-time, or when adding a tree to a variable location.

  Because the size of a context cannot be restricted using the observations exploited for the Separation Logic and the Tree Logic, it is likely that new ideas will be required to provide a decision procedure for the Context Logic. Using the results presented in this report, or otherwise, finding a decision procedure for the full Context Logic presents a challenging extension to this project.

Bibliography

[1] John Harrison. Formal verification at intel. LICS 2003, Ottawa, 2003.

[2] Bob Bently and Rand Gray. Validating the intel® pentium® 4 processor. As published at: *http://developer.intel.com/technology/itj/q12001/articles/art_3.htm*, 2001.

[3] P. O'Hearn, J. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In L. Fribourg, editor, *Computer Science Logic (CSL'01)*, pages 1–19. Springer-Verlag, 2001. LNCS 2142.

[4] C. Calcagno, L. Cardelli, and A. D. Gordon. Deciding validity in a spatial logic for trees. In *ACM Workshop on Types in Language Design and Implementation*, pages 62–73. ACM Press, 2003. TLDI '03.

[5] Luca Cardelli and Andrew D. Gordon. Mobile ambients. In *Theoretical Computer Science, Volume 240, Issue 1*, pages 177–213. Elsevier Science Publishers Ltd, 2000.

[6] Josh Berdine, Cristiano Calcagno, Peter O'Hearn, and Hongseok Yang. The spatial assertion workbench. Talk presented at *The Second Workshop on Automated Verification of Critical Systems (AVoCS 2002)*, 2002.

[7] Mohammed Zeeshan Alam. Sltree - a language to manipulate tree data. Master's thesis, Imperial College London, Department of Computing, 2003. Available from the DoC technical library, ref 1484.

[8] Silvano Dal Zilio, Denis Lugiez, and Charles Meyssonnier. A logic you can count on. In *Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 135–146. ACM Press, 2004.

[9] Etienne Lozes. Separation logic preserves the expressive power of classical logic. As published at: *http://www.diku.dk/topps/space2004/space_final/etienne.pdf*, 2004.

[10] C. Calcagno, H. Yang, and P. O'Hearn. Computability and complexity results for a spatial assertion language for data structures. In *Foundations of Software Technology and Theoretical Computer Science (FSTTCS'01)*, pages 108–119. Springer, 2001. volume 2245 of Lecture Notes in Computer Science.

[11] C.H. Papadimitiou. *Complexity Theory*. Addison-Wesley, 1994. Reading, MA.

[12] W. Charatonik and J.M. Talbot. The decidability of model checking mobile ambients. In *CSL: 15th Workshop on Computer Science Logic, volume 2142 of LNCS*, page 339, 2001.

[13] LASH. The liège automata-based symbolic handler.
http://www.montefiore.ulg.ac.be/˜boigelot/research/lash/.

[14] The Omega Project. The omega project: Frameworks and algorithms for the analysis and transformation of scientific programs.
http://www.cs.umd.edu/projects/omega/.

[15] CVC Project Team. Cvc: a cooperating validity checker.
http://verify.stanford.edu/CVC/.

[16] Projet Cristal. Welcome to the o'caml language "the programming tool of choice for discriminating hackers".
http://www.ocaml.org/.

[17] Xavier Leroy. The objective caml system, release 3.07, documentation and user's manual, 2003.
http://caml.inria.fr/ocaml/htmlman/index.html.

[18] M.J. Fischer and M.O.Rabin. Super-exponential complexity of presburger arithmetic. In *Proceedings of the SIAM-AMS Symposium in Applied Mathematics, vol. 7*, pages 27–41, 1974.

[19] C. A. R. Hoare. An axiomatic basis for computer programming. In *Communications of the ACM, v.12 n.10*, pages 576–580, 1969.

[20] Cristiano Calcagno, Philippa Gardner, and Uri Zarfaty. A context logic for tree update. as published at:
http://www.dcs.qmw.ac.uk/˜ccris/ftp/ctxlogic.pdf, 2004.

Appendix A

Implementation

A.1 Translations

A.1.1 Translation of the Placement Modality

In section 5.3.1 a translation of $A@a$ into Sheaves Logic is given. The proof of the translation is given below:

Assume $A = \exists(n1, \ldots, np).\phi_A \cdot (\alpha_1[B_1], \ldots, \alpha_p[B_p])$
And that $B_i = \exists \mathbf{N_B}.\phi_{B_i} \cdot \mathbf{E}$ for $i \in 1..p$
where $\mathbf{N_B}$ and $\mathbf{E}$ are common to all B_i.

We define:
$A@a =_{\text{def}}$
$\exists \mathbf{N_B}. \left[(\exists(n_1, \ldots, n_p). \left(\phi_A \wedge \bigvee_{\substack{i \in 1..p \\ a \in \alpha_i}} \left(n_i = 1 \wedge \bigwedge_{\substack{j \in 1..p \\ i \neq j}} (n_j = 0) \wedge \phi_{B_i} \right) \right) \right] \cdot \mathbf{E}$

Proof:
For all trees, d, $d \models A@a$ iff $a[d] \models A$ iff $a[d] \in [\![A]\!]$

Because $A = \exists(n1, \ldots, np).\phi_A \cdot (\alpha_1[B_1], \ldots, \alpha_p[B_p])$
$a[d] \in [\![A]\!]$
iff $a[d] \in [\![\exists(n1, \ldots, np).\phi_A \cdot (\alpha_1[B_1], \ldots, \alpha_p[B_p])]\!]$
iff $\exists(n_1, \ldots, n_p).\phi_A \wedge \exists i \in 1..p. \left(a \in \alpha_i \wedge d \in [\![B_i]\!] \wedge n_i = 1 \wedge \bigwedge_{\substack{j \in 1..p \\ i \neq j}} (n_j = 0) \right)$
iff $\exists(n_1, \ldots, n_p).\phi_A \wedge \exists_{\substack{i \in 1..p \\ a \in \alpha_i}} \left(d \in [\![B_i]\!] \wedge n_i = 1 \bigwedge_{\substack{j \in 1..p \\ i \neq j}} (n_j = 0) \right)$

As $B_i = \exists \mathbf{N_B}.\phi_{B_i} \cdot \mathbf{E}$, for all $i \in 1..p$
where $\mathbf{N_B}$ and $\mathbf{E}$ are common to all B_i

$\exists(n_1, \ldots, n_p).\phi_A \exists_{\substack{i \in 1..p \\ a \in \alpha_i}} \left(d \in [\![B_i]\!] \wedge n_i = 1 \bigwedge_{\substack{j \in 1..p \\ i \neq j}} (n_j = 0) \right)$

iff $\exists(n_1,\ldots,n_p).\phi_A\exists_{\substack{i\in 1..p\\ a\in\alpha_i}}\left(d\in[\![\exists\mathbf{N_B}.\phi_{B_i}\cdot\mathbf{E}]\!]\wedge n_i=1\bigwedge_{\substack{j\in 1..p\\ i\neq j}}(n_j=0)\right)$

And, since $\mathbf{N_B}$ and $\mathbf{E}$ are common to all B_i:

$$\exists(n_1,\ldots,n_p).\phi_A\exists_{\substack{i\in 1..p\\ a\in\alpha_i}}\left(d\in[\![\exists\mathbf{N_B}.\phi_{B_i}\cdot\mathbf{E}]\!]\wedge n_i=1\bigwedge_{\substack{j\in 1..p\\ i\neq j}}(n_j=0)\right)$$

iff $d\in\left[\!\!\left[\exists\mathbf{N_B}.\left[(\exists(n_1,\ldots,n_p).\left(\phi_A\wedge\bigvee_{\substack{i\in 1..p\\ a\in\alpha_i}}\left(n_i=1\wedge\bigwedge_{\substack{j\in 1..p\\ i\neq j}}(n_j=0)\wedge\phi_{B_i}\right)\right)\right]\cdot\mathbf{E}\right]\!\!\right]$

as required.

■

A.1.2 Translating Sheaves Logic to Recursive Sheaves Logic

The translation from Sheaves Logic to Recursive Sheaves Logic is given in table 5.6. The proof of this translation is given below, the proof is by induction over the Sheaves Logic formula A:

- Base case: $A=\top$ — the RSL translation is $\langle X\leftarrow\exists N.(N\geq 0).\emptyset^{\perp}[X];X\rangle$. Proof by induction on the structure of trees:
  - Base case: $d=0$ — X is satisfied when $N=0$.
  - Inductive Step: assume d satisfies X when $N=n$ and d' satisfies X when $N=n'$.
    * Case 1: $d''=d|d'$ — d is of the form $(n)\cdot\emptyset^{\perp}[X]$ and d' is of the form $(n')\cdot\emptyset^{\perp}[X]$. Therefore d'' is of the form $(n+n')\cdot\emptyset^{\perp}[X]$. We know that $n+n'\geq 0$ as $n,n'\geq 0$.
    * Case 2: $d''=a[d]$ where a is any label. $a\in\emptyset^{\perp}$ trivially and d satisfies X by induction, therefore d'' is of the form $(1).\emptyset^{\perp}[X]$ (and $1\geq 0$).
- Case 2: $A=\exists\mathbf{N}.\phi(\mathbf{N})\cdot\mathbf{E}$ — that is, $A=\exists\mathbf{N}.\phi(\mathbf{N}).(\alpha_1[A_1],\ldots,\alpha_n[A_n])$

  By induction, we know that $A_i=\langle D_i;X_i\rangle$ for all $i\in 1..n$. We must also enforce that all recursive variable names are unique, that is (using the notation $rvn(A)$ to denote the recursive variable names used in A) $rvn(A_i)\cap rvn(A_j)\neq\emptyset\Rightarrow i=j$ for all $i,j\in 1..n$.

  Taking Y such that $Y\notin\bigcup_{i\in 1..n}rvn(A_i)$ (that is, a unique recursive variable name) we translate A to:

  $$\langle Y\leftarrow\exists\mathbf{N}.\phi(\mathbf{N})\cdot(\alpha_1[X_1],\ldots,\alpha_n[X_n]),D_1,\ldots,D_n;Y\rangle$$

Because we ensured that all recursive variable names are unique, the behaviour of the recursive formulae $X_1, \ldots, X_n$ will remain the same as the variable names used in each formula are unique to that formula.

The set of trees accepted by the translation is equal to the set of tree accepted by A because, since Y is unique, the formula which must be satisfied is $\exists \mathbf{N}.\phi(\mathbf{N}) \cdot (\alpha_1[X_1], \ldots, \alpha_n[X_n])$. The Presburger formula remains unchanged and the element formulae maintain the same αs. Each A_i is replaced by the X_i obtained from the induction step. Each X_i will accept the same set of trees as the corresponding A_i by induction and the use of unique names.

■

A.2 Optimisations

A.2.1 Extended Sheaves Logic Direct Translations

In section 6.1 we extended the Sheaves Logic to include $\perp$ and $\mathbf{0}$. For the cases when the translation from Tree Logic to Sheaves Logic is not the same as the translation given in section 2.4.5, the encoding is detailed explicitly in table A.1. Most of the translations are trivial, proofs for the interesting cases are given below:

- Case $\mathbf{0} \Rightarrow A$

  $\mathbf{0} \Rightarrow A$ iff $\neg\mathbf{0} \vee A$ iff $\exists \mathbf{N}.\mathbf{N} > \mathbf{O} \vee \phi_A(\mathbf{N}) \cdot \mathbf{E}$.

- Case $A \Rightarrow \perp$

  $A \Rightarrow \perp$ iff $\neg A$ iff $\exists \mathbf{N}.\neg\phi_A(\mathbf{N}) \cdot \mathbf{E}$.

- Case $A \Rightarrow \mathbf{0}$

  $A \Rightarrow \mathbf{0}$ iff $\neg A \vee \mathbf{0}$ iff $\exists \mathbf{N}.\neg\phi_A(\mathbf{N}) \vee \mathbf{N} = \mathbf{0} \cdot \mathbf{E}$.

- Case $\top \mid A$

  If $A = \exists \mathbf{N}.\phi_A(\mathbf{N}) \cdot \mathbf{E}$, we can express truth over the basis $\mathbf{E}$ as $\exists \mathbf{M}.\mathbf{M} \geq \mathbf{0} \cdot \mathbf{E}$. The composition of $A = \exists \mathbf{N}.\phi_A(\mathbf{N}) \cdot \mathbf{E}$ and $B = \exists \mathbf{M}.\mathbf{M} \geq \mathbf{0} \cdot \mathbf{E}$ by the standard Sheaves Logic translations is $\exists \mathbf{N}.\mathbf{N_1}, \mathbf{N_2}.(\mathbf{N} = \mathbf{N_1} + \mathbf{N_2} \wedge \mathbf{N_1} \geq 0 \wedge \phi_A(\mathbf{N_2})) \cdot \mathbf{E}$. This is equivalent to $\exists \mathbf{N}, \mathbf{M}.\mathbf{N} \geq \mathbf{M} \wedge \phi_A(\mathbf{M}) \cdot \mathbf{E}$.

- Case $\top \triangleright A$

  As above, let $A = \exists \mathbf{N}.\phi_A(\mathbf{N}) \cdot \mathbf{E}$ and $\top = \exists \mathbf{M}.\mathbf{M} \geq \mathbf{0} \cdot \mathbf{E}$. Then $\top \triangleright A = \exists \mathbf{N}.\forall \mathbf{M}.(\mathbf{M} \geq \mathbf{0} \Rightarrow \phi_A(\mathbf{N} + \mathbf{M})) \cdot \mathbf{E}$; as $\mathbf{M} \geq 0$ is always true, the formula reduces to $\exists \mathbf{N} \forall \mathbf{M}.\phi_A(\mathbf{N} + \mathbf{M}) \cdot \mathbf{E}$.

- Case $A \triangleright \perp$

  Let $A = \exists \mathbf{N}.\phi_A(\mathbf{N}) \cdot \mathbf{E}$ and $\perp = \exists \mathbf{N}.\mathbf{F} \cdot \mathbf{E}$. Then $A \triangleright \perp = \exists \mathbf{N} \forall M.\phi_A(\mathbf{M}) \Rightarrow \mathbf{F} \cdot \mathbf{E}$ which is equivalent to $\forall M.\neg\phi_A(\mathbf{M}) \cdot \mathbf{E}$.

Assume $A = \exists \mathbf{N}.\phi_A(\mathbf{N}) \cdot \mathbf{E}$

$$
\begin{array}{rcl|rcl|rcl}
\mathbf{0} \wedge \mathbf{0} & =_{def} & \mathbf{0} & \mathbf{0} \vee \mathbf{0} & =_{def} & \mathbf{0} & \mathbf{0} \mid \mathbf{0} & =_{def} & \mathbf{0} \\
\mathbf{0} \wedge \top & =_{def} & \mathbf{0} & \mathbf{0} \vee \top & =_{def} & \top & \mathbf{0} \mid \top & =_{def} & \top \\
\mathbf{0} \wedge \bot & =_{def} & \bot & \mathbf{0} \vee \bot & =_{def} & \mathbf{0} & \mathbf{0} \mid \bot & =_{def} & \bot \\
\top \wedge \top & =_{def} & \top & \top \vee \top & =_{def} & \top & \mathbf{0} \mid A & =_{def} & A \\
\top \wedge \bot & =_{def} & \bot & \top \vee \bot & =_{def} & \top & \top \mid \top & =_{def} & \top \\
\top \wedge A & =_{def} & A & \top \vee A & =_{def} & \top & \top \mid \bot & =_{def} & \bot \\
\bot \wedge \bot & =_{def} & \bot & \bot \vee \bot & =_{def} & \top & \bot \mid \bot & =_{def} & \bot \\
\bot \wedge A & =_{def} & \bot & \bot \vee A & =_{def} & A & \bot \mid A & =_{def} & \bot
\end{array}
$$

$$
\begin{array}{rcl}
\mathbf{0} \wedge A & =_{def} & \exists \mathbf{N}.\mathbf{N} = \mathbf{O} \wedge \phi_A(\mathbf{N}) \cdot \mathbf{E} \\
\mathbf{0} \vee A & =_{def} & \exists \mathbf{N}.\mathbf{N} = \mathbf{O} \vee \phi_A(\mathbf{N}) \cdot \mathbf{E} \\
\top \mid A & =_{def} & \exists \mathbf{N}, \mathbf{M}.\mathbf{N} \geq \mathbf{M} \wedge \phi_A(\mathbf{M}) \cdot \mathbf{E}
\end{array}
$$

$$
\begin{array}{rcl|rcl|rcl}
\mathbf{0} \triangleright \mathbf{0} & =_{def} & \mathbf{0} & \mathbf{0} \Rightarrow \mathbf{0} & =_{def} & \top & \neg\top & =_{def} & \bot \\
\mathbf{0} \triangleright \top & =_{def} & \top & \mathbf{0} \Rightarrow \top & =_{def} & \top & \neg\bot & =_{def} & \top \\
\mathbf{0} \triangleright \bot & =_{def} & \bot & \top \Rightarrow \top & =_{def} & \mathbf{0} & a[\bot] & =_{def} & \bot \\
\mathbf{0} \triangleright \bot & =_{def} & A & \top \Rightarrow \bot & =_{def} & \top & \mathbf{0}@a & =_{def} & \bot \\
\top \triangleright \top & =_{def} & \bot & \top \Rightarrow \bot & =_{def} & \bot & \top@a & =_{def} & \top \\
\top \triangleright \bot & =_{def} & \top & \top \Rightarrow A & =_{def} & A & \bot @a & =_{def} & \bot \\
\top \triangleright \bot & =_{def} & \bot & \bot \Rightarrow \bot & =_{def} & \top & & & \\
\bot \triangleright \bot & =_{def} & \top & \bot \Rightarrow \bot & =_{def} & \top & & & \\
\bot \triangleright \bot & =_{def} & \top & \bot \Rightarrow \bot & =_{def} & \top & & & \\
\bot \triangleright \bot & =_{def} & \top & \bot \Rightarrow A & =_{def} & \top & & & \\
\bot \triangleright A & =_{def} & \top & A \Rightarrow \top & =_{def} & \top & & & \\
A \triangleright \top & =_{def} & \top & & & & & &
\end{array}
$$

$$
\begin{array}{rcl}
\mathbf{0} \Rightarrow \bot & =_{def} & \exists \mathbf{N}.\mathbf{N} > \mathbf{O} \cdot AnyE \\
\mathbf{0} \Rightarrow A & =_{def} & \exists \mathbf{N}.\mathbf{N} > \mathbf{O} \vee \phi_A(\mathbf{N}) \cdot \mathbf{E} \\
\top \triangleright A & =_{def} & \exists \mathbf{N} \forall \mathbf{M}.\phi_A(\mathbf{N} + \mathbf{M}) \cdot \mathbf{E} \\
A \Rightarrow \bot & =_{def} & \exists \mathbf{N}.\neg\phi_A(\mathbf{N}) \cdot \mathbf{E} \\
A \Rightarrow \mathbf{0} & =_{def} & \exists \mathbf{N}.\neg\phi_A(\mathbf{N}) \vee \mathbf{N} = \mathbf{0} \cdot \mathbf{E} \\
A \triangleright \bot & =_{def} & \forall \mathbf{M}.\neg\phi_A(\mathbf{M}) \cdot \mathbf{E} \\
A \triangleright \mathbf{0} & =_{def} & \exists \mathbf{N} \forall \mathbf{M}.\neg\phi_A(\mathbf{M}) \vee \mathbf{N} + \mathbf{M} = \mathbf{0} \cdot \mathbf{E} \\
\neg\mathbf{0} & =_{def} & \exists \mathbf{N}.\mathbf{N} > \mathbf{O} \cdot AnyE \\
a[\top] & =_{def} & \exists (n_1, n_2).n_1 = 1 \wedge n_2 = 0 \cdot (a[\top], \{a\}^{\perp}[\top]) \\
a[\mathbf{0}] & =_{def} & \exists (n_1, n_2, n_3). \begin{pmatrix} n_1 = 1 \wedge \\ n_2 = 0 \wedge \\ n_3 = 0 \end{pmatrix} \cdot (a[\mathbf{0}], a[A_1], \{a\}^{\perp}[\top]) \\
 & & \text{where } A_1 = \exists (n).n > 0 \cdot AnyE
\end{array}
$$

Table A.1: Translating Tree Logic to Extended Sheaves Logic

- Case $A \triangleright \mathbf{0}$

  Let $A = \exists \mathbf{N}.\phi_A(\mathbf{N}) \cdot \mathbf{E}$ and $\mathbf{0} = \exists \mathbf{N}.\mathbf{N} = \mathbf{0} \cdot \mathbf{E}$. Then $A \triangleright \mathbf{0} = \exists \mathbf{N} \forall M.\phi_A(\mathbf{M}) \Rightarrow \mathbf{M} + \mathbf{N} = \mathbf{0} \cdot \mathbf{E}$ which is equivalent to $\exists \mathbf{N} \forall \mathbf{M}.\neg\phi_A(\mathbf{M}) \vee \mathbf{N} + \mathbf{M} = \mathbf{0} \cdot \mathbf{E}$.

- Case $\perp \triangleright B$

  For any B, $\perp \triangleright B$ is vacuously true: there are no trees which satisfy false, and therefore, all trees that do, when composed with a tree, satisfy any given formula.

- Case $a[\mathbf{0}]$

  The translation for $a[\mathbf{0}]$ is the formula produced when the standard Sheaves Logic translation is applied to the formula.

- Case $a[\top]$

  By applying the standard translation we acquire, $\exists(n_1, n_2, n_3).(n_1 = 1 \wedge n_2 = 0 \wedge n_3 = 0).(a[\top], a[\perp], a^{\perp}[\top])$. Because $a[\perp]$ is unsatisfiable, we must assert that $n_2 = 0$. This does not change the Presburger constraint, and, since $(a[\top], a^{\perp}[\top])$ refines $(a[\top], a[\perp], a^{\perp}[\top])$, we can redefine the translation over this reduced basis, giving $\exists(n_1, n_2).n_1 = 1 \wedge n_2 = 0 \cdot (a[\top], \{a\}^{\perp}[\top])$.

- Case $a[\perp]$

  By applying the standard translation we acquire, $\exists(n_1, n_2, n_3).(n_1 = 1 \wedge n_2 = 0 \wedge n_3 = 0).(a[\top], a[\perp], a^{\perp}[\top])$. Because $a[\perp]$ is unsatisfiable, we must assert that $n_1 = 0$. This requires that $n_1 = 0$ and $n_1 = 1$. This condition in unsatisfiable, and therefore the translation is $\perp$.

■

A.2.2 Translating Extended Sheaves Logic to Recursive Sheaves Logic

To prove the correctness of the translation (given in table 6.1) from Extended Sheaves Logic to Recursive Sheaves Logic it only remains to prove that $X_{\perp}$ is equivalent to falsity in the Tree Logic, and that $X_{\mathbf{0}}$ is equivalent to $\mathbf{0}$ in the Tree Logic. This is because we know that $X_{\top}$ is equivalent to truth by the proof given in appendix A.1.2. The proof for the case when $A = \exists \mathbf{N}.\phi(\mathbf{N}) \cdot \mathbf{E}$ also carries through to the Extended Sheaves Logic with the exception that the recursive variable names $X_{\top}$, $X_{\perp}$ and $X_{\mathbf{0}}$ are not unique to each sub-formula. This does not change the argument significantly since $X_{\top}$, $X_{\perp}$ and $X_{\mathbf{0}}$ are self-contained formulae that accept the same trees in each sub-formula. The presence or absence of any other formulae that do not redefine $X_{\top}$, $X_{\perp}$ and $X_{\mathbf{0}}$ will not alter their behaviour.

- Case $A = \perp$ — the RSL translation is $\langle X_\perp \leftarrow \exists N.(\mathbf{F}).\emptyset^\perp[X_\perp]; X_\perp \rangle$.

  We need to prove that $X_\perp \leftarrow \exists N.(\mathbf{F}).\emptyset^\perp[X_\perp]$ accepts no trees. The Presburger constraint, $\mathbf{F}$ is not satisfiable, and therefore the RSL translation will accept no trees.

- Case $A = \mathbf{0}$ — the RSL translation is $\langle X_\mathbf{0} \leftarrow \exists N.(N = 0).\emptyset^\perp[X_\mathbf{0}]; X_\mathbf{0} \rangle$.

  We need to prove that $X_\mathbf{0}$ accepts tree d iff $d \equiv \mathbf{0}$. Suppose for contradiction that $X_\mathbf{0}$ accepts a tree that is not equivalent to $\mathbf{0}$. Then it must be the case that $N > 0$. This is a contradiction (as required) since the Presburger constraint enforces $N = 0$.

■

A.2.3 Basis Optimisation

A relation $\mathcal{R}$ refines a basis $\mathbf{E} = (E_1, \ldots, E_n)$ to another basis $\mathbf{F} = (F_1, \ldots, F_m)$ iff for all $i \in 1..n$:

$$[\![E_i]\!] = \bigcup_{(i,j)\in\mathcal{R}} [\![F_j]\!]$$

We need to show that if this property holds, then $\mathcal{R}'$ also refines the basis $\mathbf{E}$ to the basis $\mathbf{F}'$, where:

$$\mathcal{R}' = \{(i,j) | F_j \equiv\perp \mathit{or} F_j = \emptyset[X]\}$$

and $\mathbf{F}'$ is the basis $\mathbf{F}$, with the elements F_j removed if $F_j \equiv \perp$ or $F_j = \emptyset[X]$.

Proof:

We need to show that for all $i \in 1..n$:

$$[\![E_i]\!] = \bigcup_{(i,j)\in\mathcal{R}'} [\![F'_j]\!]$$

It is easy to see that,

$$\bigcup_{(i,j)\in\mathcal{R}'} [\![F'_j]\!] = \bigcup_{(i,j)\in\mathcal{R}} [\![F_j]\!] - \bigcup_{\substack{(i,j)\in\mathcal{R} \\ F_j\equiv\perp}} [\![F_j]\!] - \bigcup_{\substack{(i,j)\in\mathcal{R} \\ F_j\equiv\emptyset[X]}} [\![F_j]\!]$$

If $F_j \equiv\perp$ then $[\![F_j]\!] = \emptyset$, and if $F_j = \emptyset[X]$ then $[\![F_j]\!] = \{a[D] | a \in \emptyset \wedge D \models X\} = \emptyset$. In both cases, $[\![F_j]\!]$ is the empty set, therefore:

$$\bigcup_{(i,j)\in\mathcal{R}'} [\![F'_j]\!] = \bigcup_{(i,j)\in\mathcal{R}} [\![F_j]\!] - \bigcup_{\substack{(i,j)\in\mathcal{R} \\ F_j\equiv\perp}} \emptyset - \bigcup_{\substack{(i,j)\in\mathcal{R} \\ F_j\equiv\emptyset[X]}} \emptyset$$

And so,

$$\bigcup_{(i,j)\in\mathcal{R}'} [\![F'_j]\!] = \bigcup_{(i,j)\in\mathcal{R}} [\![F_j]\!]$$

as required.

■

A.3 Testing

Three sets of test data were used when testing the system in section 7.1. This test data is provided in tables A.2 and A.3, A.4, A.5.

```
EXPTRUE 0 OFTYPE 0;
EXPFALSE a[0] OFTYPE 0;
EXPTRUE 0 OFTYPE T;
EXPTRUE a[0] OFTYPE T;
EXPTRUE a[0] | b[0] OFTYPE T;
EXPFALSE 0 OFTYPE F;
EXPFALSE a[0] OFTYPE F;
EXPFALSE a[0] | b[0] OFTYPE F;
EXPTRUE a[0] OFTYPE a[0];
EXPFALSE a[a[0]] OFTYPE a[0];
EXPTRUE a[0] OFTYPE a[0];
EXPTRUE a[a[0]] OFTYPE a[T];
EXPTRUE a[b[0]] OFTYPE a[T];
EXPFALSE b[0] OFTYPE a[T];
EXPFALSE a[0] | a[0] OFTYPE a[T];
EXPFALSE a[0] | b[0] OFTYPE a[T];
EXPFALSE a[0] OFTYPE a[0] | b[0];
EXPTRUE a[0] | b[0] OFTYPE a[0] | T;
EXPTRUE a[0] OFTYPE a[0] | T;
EXPFALSE b[0] OFTYPE a[0] | T;
EXPTRUE a[0] OFTYPE a[0] | 0;
EXPFALSE a[0] | a[0] OFTYPE a[0] | 0;
EXPFALSE a[0] OFTYPE a[0] | F;
EXPFALSE a[0] | 0 OFTYPE a[0] | F;
EXPFALSE 0 OFTYPE 0@a;
EXPFALSE a[0] OFTYPE 0@a;
EXPFALSE b[0] OFTYPE 0@b;
EXPFALSE a[a[0]] OFTYPE 0@a;
EXPFALSE a[0] OFTYPE a[0]@a;
EXPFALSE a[a[0]] OFTYPE a[0]@a;
EXPTRUE a[0] OFTYPE a[0] AND a[T];
EXPFALSE a[a[0]] OFTYPE a[0] AND a[T];
EXPTRUE a[0] OFTYPE a[0] -> a[T];
EXPTRUE a[0] OFTYPE 0 |> a[0];
EXPTRUE a[0] OFTYPE b[0] |> a[0] | T;
EXPTRUE a[0] OFTYPE F |> F;
EXPTRUE VALID (0 OR p[0]) | NOT(p[0]);
EXPTRUE VALID q[NOT 0] |> NOT(0);
EXPTRUE VALID (T |> NOT((q[0] OR T) |> 0))@q;
EXPTRUE VALID NOT(((0 OR p[0])@p)@p@p);
EXPTRUE VALID (NOT(p[T]) OR NOT(q[T]))@q;
```

Table A.2: Simple test cases

```
EXPTRUE VALID p[T] |> p[T] | T;
EXPTRUE VALID NOT(p[T] |> 0);
EXPTRUE VALID (T | (NOT(0) OR 0)) | T;
EXPTRUE VALID (T | q[T])@q OR 0;
EXPFALSE VALID a[T] -> a[0];
EXPFALSE VALID NOT(0) AND T;
EXPFALSE VALID T |> 0;
EXPFALSE VALID a[0]@a;
EXPTRUE SAT a[0]@a;
EXPTRUE VALID a[0] -> NOT a[0] OR NOT b[0];
```

Table A.3: Simple test cases continued

```
EXPTRUE a[0] | b[0] OFTYPE a[0] | b[0];
EXPTRUE b[0] | a[0] OFTYPE a[0] | b[0];
EXPFALSE a[0] | b[0] | b[0] OFTYPE a[0] | b[0];
EXPTRUE a[0] | a[0] OFTYPE a[0] | a[0];
EXPFALSE a[0] | a[0] | a[0] OFTYPE a[0] | a[0];
EXPFALSE a[0] | b[0] OFTYPE a[0] AND b[0];
EXPTRUE a[0] | b[0] OFTYPE (a[0] | T) AND (b[0] | T);
EXPTRUE a[0] | b[0] OFTYPE (a[0] | T) -> (b[0] | T);
EXPTRUE a[0] | b[0] OFTYPE a[0] -> (b[0] | T);
EXPFALSE a[0] | b[0] OFTYPE (a[0] | T) -> b[0];
EXPTRUE a[0] OFTYPE b[0] |> a[0] | b[0];
EXPFALSE a[0] OFTYPE b[0] |> a[0] | b[b[0]];
EXPTRUE VALID NOT((q[q[0]] | q[0])@q);
EXPTRUE a[b[0]|c[d[0]]] OFTYPE a[T|c[T]] | T;
EXPFALSE VALID a[T|c[T]] | T -> a[b[0]|c[d[0]]];
EXPTRUE SAT a[0]|b[c[0]|d[0]] |> a[0]|b[c[0]|d[0]] | b[c[0]];
EXPFALSE a[0] | b[0] | c[0] | a[0] | b[0] SUBTYPE T | b[0] | T | e[0];
EXPTRUE VALID (a[b[0]] |> a[T] | e[0]) |> (e[0] | T);
EXPFALSE SAT a[b[0] | c[0]] AND a[b[0] | d[0]];
EXPTRUE SAT a[b[0] | c[0]] OR a[b[0] | d[0]];
```

Table A.4: Moderate test cases

```
EXPFALSE SAT a[b[c[NOT e[0] AND NOT f[0]]]] AND (a[b[c[e[0]]]] OR
  a[b[c[f[0]]]]);
EXPTRUE VALID a[a[a[a[a[a[0]]]]]] -> a[a[a[a[T]]]];
EXPTRUE SAT a[b[(e[0] OR f[0]) | T]] -> a[b[T]] AND (a[b[e[0]]] OR
  a[b[f[0] | a[0] | b[f[T]]]]);
EXPFALSE VALID a[b[(e[0] OR f[0]) | T]] -> a[b[T]] AND (a[b[e[0]]]
  OR a[b[f[0] | a[0] | b[f[T]]]]);
EXPTRUE SAT a[b[a[0] | b[0] | c[d[T]] |> a[0] | b[0]| c[d[0]] | T]];
EXPTRUE SAT a[0] | a[b[c[0]]] | T AND a[T] | a[T] | 0 | 0;
EXPTRUE SAT a[0] | a[b[T]] | b[0] AND a[b[0]] | b[0] | a[0];
EXPTRUE SAT a[0] | a[b[T]] | b[0] AND a[b[0]] | b[0] | a[0] AND
  a[T] | a[T] | b[T] AND a[T] | b[0] | a[0];
EXPFALSE SAT a[0] | a[b[T]] | b[0] AND a[b[0]] | b[0] | a[0] AND
  a[T] | a[T] | b[T] AND a[0] | b[0] | a[0];
EXPFALSE SAT f[0] | a[b[T]] | b[0] AND a[b[0]] | b[0] | a[0] AND
  a[T] | a[T] | b[T] AND a[T] | b[0] | a[0];
EXPTRUE SAT a[0] | a[b[c[0]]] | T OR a[T] | a[T] | 0 | 0;
EXPTRUE SAT a[0] | a[b[T]] | b[0] OR a[b[0]] | b[0] | a[0];
EXPTRUE SAT a[0] | a[b[T]] | b[0] OR (a[b[0]] | b[0] | a[0] AND
  a[T] | a[T] | b[T] AND a[T] | b[0] | a[0]);
EXPTRUE SAT a[0] | a[b[T]] | b[0] OR a[b[0]] | b[0] | a[0] OR a[T]
  | a[T] | b[T] AND a[0] | b[0] | a[0];
EXPTRUE SAT f[0] | a[b[T]] | b[0] OR a[b[0]] | b[0] | a[0] OR a[T]
  | a[T] | b[T] OR a[T] | b[0] | a[0];
EXPTRUE SAT a[b[c[e[0] OR f[0]]]] AND (a[b[c[e[0]]]] OR
  a[b[c[f[0]]]]);
EXPTRUE SAT a[0] | b[c[d[c[e[0]]]]] -> a[T] | b[c[d[T]]] | T;
EXPFALSE VALID a[b[0]] | b[c[d[e[T]]]] -> a[b[T]] | T | 0 |
  b[c[d[e[0]]]];
EXPTRUE SAT a[b[0]] | b[c[d[e[T]]]] -> a[b[0]] | T | 0 |
  b[c[d[e[T]]]];
EXPTRUE SAT a[T] | b[a[0] OR c[d[e[b[0] | d[0]]]]] |> a[T] | b[a[0]]
  | b[c[d[e[b[0] | d[0]]]]] | T;
EXPTRUE SAT a[a[b[b[b[c[e[T]]]]]]];
EXPFALSE VALID a[a[b[b[c[e[T]]]]]];
```

Table A.5: Difficult test cases

Appendix B

Theory

B.1 Deciding the Separation Logic

B.1.1 Composition Adjunct ($-\!*$)

In section 8.1.2 we state that to evaluate an assertion of the form $s, h \models \phi_1 -\!* \phi_2$, we only need consider a finite set of heaps, h'. The domain of h' must be a subset of $L' \cup S_{\phi,s}$, where L' is a set of $max(|\phi_1|, |\phi_2|)$ locations, such that L and L' are disjoint, and $S_{\phi,s}$ is the set $s(FV(\phi))$. The set of values mapped to by this heap must be a subset of $S_{\phi,s} \cup \{nil, v\}$. The heap, $h * h'$, is then a heap whose domain is a subset of $L \cup L' \cup S_{\phi,s}$, and whose values are a subset of $S_{\phi,s} \cup \{nil, v\}$. The proof of this property follows.

We begin by introducing a result given by Calcagno, Yang and O'Hearn [10]: given a state (s, h) and assertions ϕ, ψ, let X be $FV(\phi) \cup FV(\psi)$ and B a finite set consisting of the first $max(|\phi|, |\psi|)$ locations in $Loc - (dom(h) \cup s(X))$ where the ordering is given by **ord**. Pick a value $V \in Val - s(X) - \{nil\}$. Then $(s, h) \models \phi -\!* \psi$ holds iff for all h_1 such that

- $h\#h_1$ and $(s, h_1) \models \phi$
- $dom(h_1) \subseteq B \cup s(X)$
- for all $l \in dom(h_1), h_1(l) \in (s(FV(X)) \cup \{nil, v\}) \times (s(FV(X)) \cup \{nil, v\})$

we have that $(s, h * h_1) \models \psi$.

The property that we wish to prove is a corollary of this result. It is easy to see that the set L is the set B, and that $s(X) \subseteq S_{\phi,s}$, since $FV(\phi_1) \cup FV(\phi_2) \subseteq FV(\phi)$, and $S_{\phi,s} = s(FV(\phi))$.

Finally, the domain of $h * h_1$ is a subset of $L \cup L' \cup S_{\phi,s}$ because the domain of h is a subset of $L \cup S_{\phi,s}$ and the domain of h_1 is a subset of $L' \cup S_{\phi,s}$. The domain of $h * h'$ is therefore a subset of $(L \cup S_{\phi,s}) \cup (L' \cup S_{\phi,s})$, or $L \cup L' \cup S_{\phi,s}$.

$\blacksquare$

B.1.2 Reducing Heaps

Also in section 8.1.2 we introduce the notation $\mathcal{R}_{\phi,s}(h)$. We use this notation to denote a heap that has been translated to an equivalent heap in the set $H_{\phi,s}$.

Intuitively, to translate a heap to an equivalent heap in $H_{\phi,s}$, we can relocate any heap cells that are not explicitly referred to in the given assertion, ϕ; we can de-allocate those cells that are redundant (that is, there only needs to be a finite number of cells that are not explicitly referred to by ϕ because ϕ can only refer to indirectly to a finite number of cells, via assertions of the form $\neg(x \mapsto y, z)$, for example); and we can replace all uninteresting values with the equally uninteresting value, v.

B.1.3 A Finite Set of Heaps

In section 8.1.2 we state that checking whether an assertion, ϕ, holds for all heaps, h, given a state, s, we only need to consider the set of heaps whose domain is a subset of a set $L \cup S_{\phi,s}$, where L is a set of $|\phi|$ locations not in $S_{\phi,s}$. Additionally, the heap must map all locations in its domain to values in the set $S_{\phi,s} \cup \{nil, v\}$, where v is a unique element not appearing in $S_{\phi,s}$, L or $\{nil\}$.

This property follows from the following results introduced by Calcagno, Yang and O'Hearn [10]:

- Given a state (s, h) and assertions ϕ, ψ, let X be $FV(\phi) \cup FV(\psi)$ and B a finite set consisting of the first $max(|\phi|, |\psi|)$ locations in $Loc - (dom(h) \cup s(X))$ where the ordering is given by **ord**. Pick a value $v \in Val - s(X) - \{nil\}$. Then $(s, h) \models \phi \mathrel{-\!\!*} \psi$ holds iff for all h_1 such that
  - $h \# h_1$ and $(s, h_1) \models \phi$
  - $dom(h_1) \subseteq B \cup s(X)$
  - for all $l \in dom(h_1), h_1(l) \in (s(FV(X)) \cup \{nil, v\}) \times (s(FV(X)) \cup \{nil, v\})$

  we have that $(s, h * h_1) \models \psi$.
- Given a stack, s, and an assertion, ϕ, checking $(s, h) \models \phi$ for all h is decidable. This corollary holds because $s, h \models \phi$ for all h iff $s, [] \models (\neg\phi) \mathrel{-\!\!*}$ false.

Therefore, $(s, h) \models \phi$ for all h iff $s, [] \models (\neg\phi) \mathrel{-\!\!*}$ false. $s, [] \models (\neg\phi) \mathrel{-\!\!*}$ false iff for all h' such that

- $[] \# h'$ and $(s, h') \models \neg\phi$
- $dom(h') \subseteq B \cup s(X)$
- for all $l \in dom(h'), h'(l) \in (s(X) \cup \{nil, v\}) \times (s(X) \cup \{nil, v\})$

we have that $(s, [] * h') \models$ false, where B is the set consisting of the first $max(|\neg\phi|, |\text{false}|)$ locations in $Loc - (dom([]) \cup s(X))$.

Since $max(|\neg\phi|, |\text{false}|) = |\phi|$, $X = FV(\neg\phi) \cup FV(\text{false}) = FV(\phi)$, and $[]\#h'$ always holds, we have, $(s, h) \models \phi$ iff for all h' such that

- $dom(h') \subseteq B \cup s(FV(\phi))$
- for all $l \in dom(h'), h'(l) \in (s(FV(\phi)) \cup \{nil, v\}) \times (s(FV(\phi)) \cup \{nil, v\})$

we have that $(s, h') \models \neg\phi \Rightarrow (s, h') \models$ false. The contrapositive of this statement is $(s, h') \models$ true $\Rightarrow (s, h') \models \phi$, which is equivalent to $(s, h') \models \phi$. The result follows since $L_{|\phi|} = B$ and $S_{\phi,s} = s(FV(\phi))$ by definition.

■

B.1.4 A Finite Set Of Stacks

In section 8.1.2 we also state that checking whether an assertion, ϕ, holds for all states, s, we only need to consider the stacks that map all free variables of ϕ to an element of a set $S_{|\phi|} \cup \{nil\}$ and all other variables to nil. We also state that we can translate any stack to a stack in this set.

This follows directly from a property introduced by Calcagno, Yang and O'Hearn [10] that follows:

To check $(s, h) \models \phi$ for all states, we can observe that the relationship of the variable is important, not their values. We define the relation, $\approx_X$, to denote that for two states, the relationship between the variables stored in X or in the heap are the same. It can be shown that, if $(s, h) \approx_{FV(\phi)} (s', h')$, then, if $(s, h) \models \phi$, then $(s', h') \models \phi$. It can also be shown that, for any state, (s, h), and assertion, ϕ, there is a state, (s', h') such that $(s, h) \approx_{FV(\phi)} (s', h')$ and $s'(Var - FV(\phi)) \subseteq \{nil\}$ and $s'(FV(\phi)) \subseteq B \cup \{nil\}$, where B is the set consisting of the first $|FV(\phi)|$ locations in Loc.

B.1.5 $H'_{\phi,n}$

We can show $H_{\phi,s,L_{|\phi|}} \subseteq H'_{\phi,|\phi|}$, since, if $h \in H_{\phi,s,L_{|\phi|}}$, then the domain of h is a subset of $L_{|\phi|} \cup S_{\phi,s}$. Because $|S_{\phi,s}| \leq |S_{|\phi|}|$, $|dom(h)| \leq |\phi| + |S_{|\phi|}|$ and so $h \in H'_{\phi,|\phi|}$. Similarly, $\mathcal{S} \subseteq \mathcal{S}^\phi$, since, if $s \in \mathcal{S}$, then s maps all variables not in $FV(\phi)$ to nil, and hence, $s \in \mathcal{S}^\phi$.

So, if $\forall s \in \mathcal{S}^\phi \forall h \in H'_{\phi,|\phi|} s, h \models \phi$, then $\forall s \in \mathcal{S} \forall h \in H_{\phi,s,L_{|\phi|}} s, h \models \phi$ and so $\forall s, h.(s, h) \models \phi$. In the other direction, $\forall s, h.(s, h) \models \phi \Rightarrow \forall s \in \mathcal{S}^\phi \forall h \in H'_{\phi,|\phi|} s, h \models \phi$, trivially.

B.2 Translating Separation Logic to Tree Logic

The translation from Separation Logic to Tree Logic is given in table 8.6. In this section we provide the proof that the translation has the following properties:

- given a formula, ϕ, and a stack, s,

$$\forall h \in H_{\phi,s}\,[(s,h \models \phi \iff heaptran(h) \models tran(\phi,s))]$$

- given a stack, s, and an assertion, ϕ,

$$[\forall h(s,h \models \phi)] \iff [\forall d(d \models tran(\phi,s))]$$

We first prove some useful properties that will be used throughout the proof. Then we prove that, if $h \in H_{\phi,s,L}$, $heaptran(h)$ will satisfy the formula, $tran'(\phi,s,\phi,L)$, iff the heap, h, satisfies ϕ. We then use this property to show the first property that we wish to prove. Finally, we show that the second property is a consequence of the first.

We begin by showing some useful properties:

1. If $L \subseteq L'$, then $heap(L,s,\phi) \Rightarrow heap(L',s,\phi)$.
2. If $d \models heap(L,s,\phi)$, then there is a heap, $h \in H_{\phi,s,L}$, such that $heaptran(h) = d$.
3. If $h \in H_{\phi,s,L}$, then $heaptran(h) \models heap(L,s,\phi)$.
4. If $h = h_1 * h_2$ and $h \in H_{\phi,s,L}$, then $h_1, h_2 \in H_{\phi,s,L}$.
5. If $d = d_1|d_2$ and $d \models heap(L,s,\phi)$, then $d_1 \models heap(L,s,\phi)$ and $d_2 \models heap(L,s,\phi)$.

The proof of these properties is as follows:

1. Assume $L \subseteq L'$ and a tree, d satisfies $heap(L,s,\phi)$, then it is a tree of the form $x_1[y_1[z_1[\mathbf{0}]]]|\ldots|x_n[y_n[z_n[\mathbf{0}]]]$, where $x_i \in L \cup S_{\phi,s}$, all x_i are unique, and $y_i, z_i \in S_{\phi,s} \cup \{nil, v\}$ for all $i \in 1..n$. Since $L \subseteq L'$, it is also the case that $x_i \in L' \cup S_{\phi,s}$, and $y_i, z_i \in S_{\phi,s} \cup \{nil, v\}$. Therefore, $d \models heap(L',s,\phi)$.
2. If a tree, d satisfies $heap(L,s,\phi)$, then it is a tree of the form $x_1[y_1[z_1[\mathbf{0}]]]|\ldots|x_n[y_n[z_n[\mathbf{0}]]]$, where $x_i \in L \cup S_{\phi,s}$, all x_i are unique, and $y_i, z_i \in S_{\phi,s} \cup \{nil, v\}$ for all $i \in 1..n$. Therefore, it is the translation of a heap, h, of the form, $h = (x_1 \mapsto y_1, z_1) * \ldots * (x_n \mapsto y_n, z_n)$. The domain of h is then a subset of $L \cup S_{\phi,s}$, and its values are all in the set $S_{\phi,s} \cup \{nil, v\}$. Therefore, $h \in H_{\phi,s,L}$, and such an h exists.
3. If heap, $h \in H_{\phi,s,L}$, then it is of the form, $h = (x_1 \mapsto y_1, z_1) * \ldots * (x_n \mapsto y_n, z_n)$, where $x_i \in L \cup S_{\phi,s}$, and $y_i, z_i \in S_{\phi,s} \cup \{nil, v\}$ for all $i \in 1..n$. $heaptran(h)$ will be of the form $x_1[y_1[z_1[\mathbf{0}]]]|\ldots|x_n[y_n[z_n[\mathbf{0}]]]$, and therefore, $heaptran(h)$ will satisfy $heap(L)$.
4. Since $h = h_1 * h_2$, it must be the case that $h_1, h_2 \in H_{\phi,s,L}$ since the domains of h_1 and h_2 are subsets of the domain of h, and their values are the same as the values in h.

5. Since $d \models heap(L, s, \phi)$ is must be the composition of elements satisfying a unique A_i in $heap(L, s, \phi)$. d_1 and d_2 will each be formed from a subset of the elements in d, and therefore, their elements will satisfy the same A_i in $heap(L, s, \phi)$. For those elements that are not in d_1 but in d_2 (and vice versa), the sub-formula $\mathbf{0}$ is satisfied instead of the sub-formula A_i, and hence $heap(L, s, \phi)$ is still satisfied.

We now prove:

$$\forall h \in H_{\phi,s,L}\, [s, h \models \phi \iff heaptran(h) \models tran'(\phi, s, \phi, L)]$$

for a given formula, ϕ and a stack, s, by induction on sub-formulae, ψ, of ϕ:

- Case $\psi = (E \mapsto E_1, E_2)$. We want $s, h \models (E \mapsto E_1, E_2) \iff heaptran(h) \models [\![E]\!]_s\big[[\![E_1]\!]_s\big[[\![E_2]\!]_s\big[\mathbf{0}\big]\big]\big]$.

  $\Rightarrow$: suppose $s, h \models (E \mapsto E_1, E_2)$, then $h = ([\![E]\!]_s \mapsto [\![E_1]\!]_s, [\![E_2]\!]_s)$ and $heaptran(h) = [\![E]\!]_s\big[[\![E_1]\!]_s\big[[\![E_2]\!]_s\big[\mathbf{0}\big]\big]\big]$ and so $heaptran(h) \models [\![E]\!]_s\big[[\![E_1]\!]_s\big[[\![E_2]\!]_s\big[\mathbf{0}\big]\big]\big]$.

  $\Leftarrow$: suppose $heaptran(h) \models [\![E]\!]_s\big[[\![E_1]\!]_s\big[[\![E_2]\!]_s\big[\mathbf{0}\big]\big]\big]$, then $heaptran(h) = [\![E]\!]_s\big[[\![E_1]\!]_s\big[[\![E_2]\!]_s\big[\mathbf{0}\big]\big]\big]$ and so $h = ([\![E]\!]_s \mapsto [\![E_1]\!]_s, [\![E_2]\!]_s)$, consequently $s, h \models (E \mapsto E_1, E_2)$.

- Case $\psi = (E_1 = E_2)$. We want $s, h \models (E_1 = E_2) \iff heaptran(h) \models tran'((E_1 = E_2), s, \phi, L)$.

  This property follows immediately since its value does not depend on the contents of h. The translation evaluates $E_1 = E_2$ and returns $\mathbf{T}$ or $\mathbf{F}$ as required. Therefore, the required property will hold by definition.

- Case $\psi = \text{false}$. We want $s, h \models \text{false} \iff heaptran(h) \models tran'(\text{false}, s, \phi, L)$.

  This is immediate since $tran'(\text{false}, s, \phi, L) = \mathbf{F}$.

- Case $\psi = (\phi_1 \Rightarrow \phi_2)$. We want $s, h \models (\phi_1 \Rightarrow \phi_2) \iff heaptran(h) \models tran'((\phi_1 \Rightarrow \phi_2), s, \phi, L)$. That is, $s, h \models (\phi_1 \Rightarrow \phi_2) \iff heaptran(h) \models (tran'(\phi_1, s, \phi, L) \Rightarrow tran(\phi_1, s, \phi, L)$. We assume inductively that $s, h \models \phi_i \iff heaptran(h) \models tran'(\phi_i, s, \phi, L)$ for $i \in \{1, 2\}$.

  $s, h \models (\phi_1 \Rightarrow \phi_2) \iff ((s, h \models \phi_1) \Rightarrow (s, h \models \phi_2)) \iff ((heaptran(h) \models tran'(\phi_1, s, \phi, L)) \Rightarrow (heaptran(h) \models tran'(\phi_2, s, \phi, L))) \iff heaptran(h) \models (tran'(\phi_1, s, \phi, L) \Rightarrow tran'(\phi_1, s, \phi, L))$, as required.

- Case $\psi = \text{emp}$. We want $s, h \models \text{emp} \iff heaptran(h) \models tran'(\text{emp}, s, \phi, L)$. That is, $s, h \models \text{emp} \iff heaptran(h) \models \mathbf{0}$.

  $\Rightarrow$: if $s, h \models \text{emp}$ then $h = []$, and so $heaptran(h) = \mathbf{0}$. Therefore, $heaptran(h) \models \mathbf{0}$.

  $\Leftarrow$: if $heaptran(h) \models \mathbf{0}$ then $heaptran(h) \equiv \mathbf{0}$ and so $h = []$. Therefore, $s, h \models \text{emp}$, as required.

- Case $\psi = \phi_1 * \phi_2$. We want $s, h \models \phi_1 * \phi_2 \iff \mathit{heaptran}(h) \models \mathit{tran}'((\phi_1 * \phi_2), s, \phi, L)$. That is, $s, h \models \phi_1 * \phi_2 \iff \mathit{heaptran}(h) \models (\mathit{tran}'(\phi_1, s, \phi, L)|\mathit{tran}'(\phi_2, s, \phi, L))$.

  $\Rightarrow$: suppose $s, h \models \phi_1 * \phi_2$. Then, there exists h_1, h_2 such that, $h_1 \# h_2$, $h = h_1 * h_2$, $(s, h_1) \models \phi_1$ and $(s, h_2) \models \phi_2$.
  $\mathit{heaptran}(h) = \mathit{heaptran}(h_1)|\mathit{heaptran}(h_2)$ since $h = h_1 * h_2$. By (4) and induction $\mathit{heaptran}(h_1) \models \mathit{tran}'(\phi_1, s, \phi, L)$ and $\mathit{heaptran}(h_2) \models \mathit{tran}'(\phi_2, s, \phi, L)$, and so $\mathit{heaptran}(h) \models \mathit{tran}'(\phi_1, s, \phi, L)|\mathit{tran}'(\phi_2, s, \phi, L)$.

  $\Leftarrow$: suppose $\mathit{heaptran}(h) \models (\mathit{tran}'(\phi_1, s, \phi, L)|\mathit{tran}'(\phi_2, s, \phi, L))$. Therefore, there exists d_1, d_2 such that $\mathit{heaptran}(h) = d_1|d_2$ and
  $d_1 \models \mathit{tran}'(\phi_1, s, \phi, L)$ and $d_2 \models \mathit{tran}'(\phi_2, s, \phi, L)$. Inductively we know that $d_1 \models \mathit{heap}(L, s, \phi)$ and $d_2 \models \mathit{heap}(L, s, \phi)$. So, by (2), there are h_1, h_2 such that $\mathit{heaptran}(h_1) = d_1$ and $\mathit{heaptran}(h_2) = d_2$. Since $\mathit{heaptran}(h) = \mathit{heaptran}(h_1)|\mathit{heaptran}(h_2)$, it follows that $h = h_1 * h_2$ (and $h_1 \# h_2$ since h is a heap). By induction, $s, h_1 \models \phi_1$ and $s, h_2 \models \phi_2$, consequently, $h \models \phi_1 * \phi_2$, as required.

- Case $\psi = \phi_1 -\!\!* \phi_2$. We want $s, h \models \phi_1 -\!\!* \phi_2 \iff \mathit{heaptran}(h) \models \mathit{tran}'((\phi_1 -\!\!* \phi_2), s, \phi, L)$. That is, $s, h \models \phi_1 -\!\!* \phi_2 \iff \mathit{heaptran}(h) \models (\mathit{tran}'(\phi_1, s, \phi, L') \wedge \mathit{heap}(L', s, \phi)) \triangleright (\mathit{heap}(L \cup L', s, \phi) \Rightarrow \mathit{tran}'(\phi_2, s, \phi, L \cup L'))$.

  $\Rightarrow$: assume $s, h \models \phi_1 -\!\!* \phi_2$, then for all h_1 such that $h \# h_1$ and $s, h_1 \models \phi_1$, $s, (h * h_1) \models \phi_2$. For $\mathit{heaptran}(h) \models (\mathit{tran}'(\phi_1, s, \phi, L') \wedge \mathit{heap}(L', s, \phi)) \triangleright (\mathit{heap}(L \cup L', s, \phi) \Rightarrow \mathit{tran}'(\phi_2, s, \phi, L \cup L'))$ to hold we require that for all d, if $d \models \mathit{tran}'(\phi_1, s, \phi, L) \wedge \mathit{heap}(L', s, \phi)$, then $d|\mathit{heaptran}(h) \models \mathit{heap}(L \cup L', s, \phi) \Rightarrow \mathit{tran}'(\phi_2, s, \phi, L \cup L')$.

  So, we assume $d \models \mathit{tran}'(\phi_1, s, \phi, L') \wedge \mathit{heap}(L', s, \phi)$. Since $d \models \mathit{heap}(L', s, \phi)$, by (2) we know that there is an h_1 such that $\mathit{heaptran}(h_1) = d$ and $h_1 \in H_{\phi, s, L'}$. Therefore, $\mathit{heaptran}(h_1) \models \mathit{tran}'(\phi_1, s, \phi, L)$ and so $s, h_1 \models \phi_1$ by induction. There are now two cases to consider: whether $h \# h_1$. If $h \# h_1$ does not hold, then $\mathit{heaptran}(h)|\mathit{heaptran}(h_1) \not\models \mathit{heap}(L \cup L')$ since $\mathit{heaptran}(h)|\mathit{heaptran}(h_1)$ will contain some element formulae with the same top branch — contradicting $\mathit{heap}(L \cup L', s, \phi)$ — and hence $\mathit{heaptran}(h)|\mathit{heaptran}(h_1) \models \mathit{heap}(L \cup L', s, \phi) \Rightarrow \mathit{tran}'(\phi_2, s, \phi, L \cup L')$ holds. In this case, the heap h_1 should not appear in the universal quantification over heaps, and so truth is the correct result. In the second case, $h * h_1$ will be a valid heap whose domain is the union of $\mathit{dom}(h)$ and $\mathit{dom}(h_1)$. Since L and L' are the disjoint components of the domains of the two heaps, $\mathit{heaptran}(h)|\mathit{heaptran}(h_1) \models \mathit{heap}(L \cup L', s, \phi)$. By our original assumption, $s, h * h_1 \models \phi_2$, so, by induction, $\mathit{heaptran}(h * h_1) \models \mathit{tran}'(\phi_2, s, \phi, L \cup L')$. $\mathit{heaptran}(h * h_1) = \mathit{heaptran}(h)|\mathit{heaptran}(h_1) = \mathit{heaptran}(h_1)|d$, and so $d|h_1 \models \mathit{heap}(L \cup L') \Rightarrow \mathit{tran}'(\phi_2, s, \phi, L \cup L')$ as required.

  $\Leftarrow$: assume $\mathit{heaptran}(h) \models (\mathit{tran}'(\phi_1, s, \phi, L') \wedge \mathit{heap}(L', s, \phi)) \triangleright (\mathit{heap}(L \cup L', s, \phi) \Rightarrow \mathit{tran}'(\phi_2, s, \phi, L \cup L')$. That is, for all trees d, such that

$d \models tran'(\phi_1, s, \phi, L') \wedge heap(L', s, \phi)$, then $d|heaptran(h) \models heap(L \cup L', s, \phi) \Rightarrow tran'(\phi_2, s, \phi, L \cup L')$.

We require that $h \models \phi_1 -\!\!* \phi_2$. To check this assertion we must evaluate $(s, h * h_1) \models \phi_2$ for all $h_1 \in H_{\phi,s,L'}$ where $L' = L_{max(|\phi_1|,|\phi_2|)}$, and for whom $h\#h_1$ and $s, h \models \phi_1$ hold.

Since $L' = L_{max(|\phi_1,\phi_2|)}$, h_1 we have $heaptran(h_1) \models heap(L', s, \phi)$ (by (3)). Since $h\#h_1$, and $(h * h_1) \in H_{\phi,s,L\cup L'}$ and by (3), we know that $heaptran(h)|heaptran(h_1) \models heap(L \cup L', s, \phi)$. Additionally, given $(s, h_1) \models \phi_1$, we know $heaptran(h_1) \models tran'(\phi_1, s, \phi, L')$ by induction. Consequently, $heaptran(h)|heaptran(h_1) \models tran'(\phi_2, s, \phi, L \cup L')$ by assumption.

Because $h\#h_1$, $heaptran(h)|heaptran(h_1) = heaptran(h * h_1)$, and so, by induction, $(s, h * h_1) \models \phi_2$, as required.

Finally, we show that, given a stack, s, and an assertion, ϕ:

$$[\forall h(s, h \models \phi)] \iff [\forall d(d \models tran(\phi, s))]$$

That is,

$$[\forall h(s, h \models \phi)] \iff [\forall d(d \models heap(L, s, \phi) \Rightarrow tran'(\phi, s, \phi, L)]$$

where $L = L_{|\phi|}$.

- $\Rightarrow$: suppose, $\forall h(s, h \models \phi)$. Also assume that $d \models heap(L, s, \phi)$. Then, by (2), there is a heap h such that $heaptran(h) = d$ and $h \in H_{\phi,s,L}$. We know $\forall h(s, h \models \phi)$, and that,

  $$\forall h \in H_{\phi,s,L}\, [s, h \models \phi \iff heaptran(h) \models tran'(\phi, s, \phi, L)]$$

  and so, since $d = heaptran(h)$, $d \models tran'(\phi, s, \phi, L)$, as required.

- $\Leftarrow$: suppose, $\forall d(d \models heap(L) \Rightarrow tran'(\phi, s, \phi, L))$. Since, to test satisfaction for all heaps, we only need to consider those heaps in $H_{\phi,s,L}$, we require $\forall h \in H_{\phi,s,L}(s, h \models \phi)$. We know that,

  $$\forall h \in H_{\phi,s,L}\, [s, h \models \phi \iff heaptran(h) \models heap(L, s, \phi) \Rightarrow tran'(\phi, s, \phi, L)]$$

  We also know that $heaptran(h) \models heap(L, s, \phi) \Rightarrow tran'(\phi, s, \phi, L)$ because $\forall d(d \models heap(L, s, \phi) \Rightarrow tran'(\phi, s, \phi, L))$. Therefore $\forall h \in H_{\phi,s,L}(s, h \models \phi)$, and so $\forall h.(s, h \models \phi)$

■

B.3 Translating Separation Logic to $FOL_=$

B.3.1 Translating Separation Logic to Presburger Arithmetic

In section 8.3 we give a translation from the Separation Logic to $FOL_=$. We now prove the following properties:

- Given an assertion, ϕ,

$$\forall(s,h) \in \mathcal{S}^\phi \times H'_{\phi,|\phi|} \left[(s,h \models \phi) \iff \forall v \in \mathit{vector}_{\phi,|\phi|}(s,h)\ v \models \mathit{tran}(\phi))\right]$$

- For an assertion, ϕ,

$$[\forall(s,h).s,h \models \phi] \iff [\forall(\mathbf{N},\mathbf{B},\mathbf{S})\ (\mathbf{N},\mathbf{B},\mathbf{S}) \models \mathit{tran}(\phi)]$$

We begin by showing some properties that will be used throughout the proof. We then show the first property using induction, and finish by showing how the second property can be derived from the first.

The properties that will be used throughout the proof are:

1. If $(\mathbf{N},\mathbf{B},\mathbf{S}) \models \mathit{bounded}(\mathbf{N}) \wedge \mathit{heap}(\mathbf{N},\mathbf{B})$, where $\mathbf{N} = \mathbf{N_L} \oplus \mathbf{N_S}$, $|\mathbf{N_L}| = p$, $|\mathbf{B_L}| = 3p$, $|\mathbf{N_S}| = |FV(\phi)|$ and $|\mathbf{B_S}| = 3|FV(\phi)|$, then there exists a state, (s,h) such that $s \in \mathcal{S}^\phi$, $h \in H'_{\phi,p}$ and $(\mathbf{N},\mathbf{B},\mathbf{S}) \in \mathit{vector}_{\phi,p}(s,h)$.
2. If $s \in \mathcal{S}^\phi$ and $h \in H'_{\phi,p}$, then for all $(\mathbf{N},\mathbf{B},\mathbf{S}) \in \mathit{vector}_{\phi,p}(s,h)$, $(\mathbf{N},\mathbf{B},\mathbf{S}) \models \mathit{bounded}(\mathbf{N}) \wedge \mathit{heap}(\mathbf{N},\mathbf{B})$ and $\mathbf{N} = \mathbf{N_L} \oplus \mathbf{N_S}$, $|\mathbf{N_L}| = p$, $|\mathbf{B_L}| = 3p$, $|\mathbf{N_S}| = |FV(\phi)|$ and $|\mathbf{B_S}| = 3|FV(\phi)|$.
3. If $\mathbf{B} = \mathbf{B'} \oplus \mathbf{B''}$, $\mathbf{N} = \mathbf{N'} \oplus \mathbf{N''}$ and $\mathit{heap}(\mathbf{N},\mathbf{B})$ holds, then both $\mathit{heap}(\mathbf{N'},\mathbf{B'})$ and $\mathit{heap}(\mathbf{N''},\mathbf{B''})$ hold.
4. If $\mathbf{N} = \mathbf{N_1} + \mathbf{N_2}$ and $\mathit{heap}(\mathbf{N},\mathbf{B})$, then $\mathit{heap}(\mathbf{N_1},\mathbf{B})$ and $\mathit{heap}(\mathbf{N_2},\mathbf{B})$ hold.

We state these properties without proof.

The property,

$$\forall(s,h) \in \mathcal{S}^\phi \times H'_{\phi,|\phi|} \left[(s,h \models \phi) \iff \forall v \in \mathit{vector}_{\phi,|\phi|}(s,h)\ v \models \mathit{tran}(\phi))\right]$$

for an assertion, ϕ, is equivalent to, $\forall(s,h) \in \mathcal{S}^\phi \times H'_{\phi,|\phi|}$

$$\left[\begin{array}{l}(s,h \models \phi) \iff \\ \forall(\mathbf{N},\mathbf{B},\mathbf{S}) \in \mathit{vector}_{\phi,|\phi|}(s,h)(\mathbf{N},\mathbf{B},\mathbf{S}) \models \left(\begin{array}{l}\mathit{bounded}(\mathbf{N}) \wedge \mathit{heap}(\mathbf{N},\mathbf{B}) \\ \Rightarrow \mathit{tran}'(\phi)(\mathbf{N},\mathbf{B},\mathbf{S})\end{array}\right)\end{array}\right]$$

By property (2) we know that $\mathit{bounded}(\mathbf{N}) \wedge \mathit{heap}(\mathbf{N},\mathbf{B})$ holds, so we need to prove that, $\forall(s,h) \in \mathcal{S}^\phi \times H'_{\phi,|\phi|}$

$$\left[(s,h \models \phi) \iff \forall(\mathbf{N},\mathbf{B},\mathbf{S}) \in \mathit{vector}_{\phi,|\phi|}(s,h)\ (\mathbf{N},\mathbf{B},\mathbf{S}) \models \mathit{tran}'(\phi)\right]$$

The proof of this property is by induction on the sub-formulae, ψ of ϕ. The case, $\psi = \phi_1 \multimap\!\!* \ \phi_2$ is presented below. We assume that $h \in H'_{\phi,p}$ for some p. At the top level $p = |\phi|$, but, recursively, p will increase in the case of the composition adjunct. Therefore, our induction hypothesis is:

$$[(s, h \models \phi) \iff \forall(\mathbf{N}, \mathbf{B}, \mathbf{S}) \in vector_{\phi,p}(s, h)\ (\mathbf{N}, \mathbf{B}, \mathbf{S}) \models tran'(\phi)]$$

Where $p \geq |\phi|$.

- Case $\psi = \phi_1 \multimap\!\!* \ \phi_2$: we require that $s, h \models \phi_1 \multimap\!\!* \ \phi_2$ iff, $\forall(\mathbf{N}, \mathbf{B}, \mathbf{S}) \in vector_{\phi,p}(s, h)$, where $\mathbf{N} = \mathbf{N_L} \oplus \mathbf{N_S}$, and $\mathbf{B} = \mathbf{B_L} \oplus \mathbf{B_S}$ and,

$$(\mathbf{N}, \mathbf{B}, \mathbf{S}) \models \forall \mathbf{M}, \mathbf{B_{L'}}. \left(\begin{array}{l} tran'(\phi_1)(\mathbf{M}, \mathbf{B_{L'}} \oplus \mathbf{B_S}, \mathbf{S}) \\ \wedge\ bounded(\mathbf{M_S} + \mathbf{N_S}) \wedge\ bounded(\mathbf{M}) \\ \wedge\ heap \left(\begin{array}{l} \mathbf{N_L} \oplus \mathbf{M_{L'}} \oplus (\mathbf{N_S} + \mathbf{M_S}), \\ \mathbf{B_L} \oplus \mathbf{B_{L'}} \oplus \mathbf{B_S} \end{array} \right) \\ \quad \Rightarrow tran'(\phi_2) \left(\begin{array}{l} \mathbf{N_L} \oplus \mathbf{M_{L'}} \oplus (\mathbf{N_S} + \mathbf{M_S}), \\ \mathbf{B_L} \oplus \mathbf{B_{L'}} \oplus \mathbf{B_S}, \\ \mathbf{S} \end{array} \right) \end{array} \right)$$

  where $\mathbf{M} = \mathbf{M_{L'}} \oplus \mathbf{M_S}$, $|\mathbf{M_{L'}}| = max(|\phi_1|, |\phi_2|), |\mathbf{M_S}| = |FV(\phi)|$, and $|\mathbf{B_{L'}}| = 3(max(|\phi_1|, |\phi_2|))$.

  $\Rightarrow$: assume that $s, h \models \phi_1 \multimap\!\!* \ \phi_2$. Therefore, for all h_1, where $s, h_1 \models \phi_1$ and $h \# h_1$ holds, $s, h * h_1 \models \phi_2$.

  We also assume that $tran'(\phi_1)(\mathbf{M}, \mathbf{B_{L'}} \oplus \mathbf{B_S}, \mathbf{S}) \wedge\ bounded(\mathbf{M_S} + \mathbf{N_S}) \wedge bounded(\mathbf{M}) \wedge\ heap(\mathbf{N_L} \oplus \mathbf{M_{L'}} \oplus (\mathbf{N_S} + \mathbf{M_S}), \mathbf{B_L} \oplus \mathbf{B_{L'}} \oplus \mathbf{B_S})$ holds. Let $q = max(|\phi_1|, |\phi_2|)$. We have that $|\mathbf{M_{L'}}| = q$, $|\mathbf{M_S}| = |FV(\phi)|$, $|\mathbf{B_{L'}}| = 3q$ and $|\mathbf{B_S}| = 3|FV(\phi)|$. We also know that $bounded(\mathbf{M})$ holds, and, since (by properties (3) and (4)) $heap(\mathbf{N_L} \oplus \mathbf{M_{L'}} \oplus (\mathbf{N_S} + \mathbf{M_S}), \mathbf{B_L} \oplus \mathbf{B_{L'}} \oplus \mathbf{B_S})$ implies $heap(\mathbf{M_{L'}} \oplus \mathbf{M_S}, \mathbf{B_{L'}} \oplus \mathbf{B_S})$. Consequently, we can use (1) to deduce that there exists a state, (s, h_1) such that $s \in \mathcal{S}^\phi$, $h_1 \in H'_{\phi,q}$ and $(\mathbf{M}, \mathbf{B_{L'}} \oplus \mathbf{B_S}, \mathbf{S}) \in vector_{\phi,q}(s, h_1)$. Since $tran'(\phi_1)(\mathbf{M}, \mathbf{B_{L'}} \oplus \mathbf{B_S}, \mathbf{S})$ holds and $q \geq |\phi_1|$, we know that $s, h_1 \models \phi_1$ by induction.

  Because $heap(\mathbf{N_L} \oplus \mathbf{M_{L'}} \oplus (\mathbf{N_S} + \mathbf{M_S}), \mathbf{B_L} \oplus \mathbf{B_{L'}} \oplus \mathbf{B_S})$ and $bounded(\mathbf{M_S} + \mathbf{N_S})$ hold, we know that $h \# h_1$ holds. This is because $heap(\mathbf{N_L} \oplus \mathbf{M_{L'}} \oplus (\mathbf{N_S} + \mathbf{M_S}), \mathbf{B_L} \oplus \mathbf{B_{L'}} \oplus \mathbf{B_S})$ enforces that the domains of the heaps represented by the vectors are disjoint. The vector $\mathbf{B_S}$ is the only component shared between the two bases. Because we know that $bounded(\mathbf{M_S} + \mathbf{N_S})$ holds, we know that the cells in $\mathbf{B_S}$ cannot be present both of the heaps, since, if a cell was present in both heaps, the summation of the vectors $\mathbf{N_S}$ and $\mathbf{M_S}$ would not be bounded.

  By the original assumption, and since $s, h_1 \models \phi_1$ and $h \# h_1$, we know that $s, h * h_1 \models \phi_2$. We can deduce that $h * h_1 \in H'_{\phi,p+q}$ since $h \in H'_{\phi,p}$ and $h_1 \in H'_{\phi,q}$. This is because the domain of $h * h_1$ is the union of $dom(h)$ and $dom(h_1)$. Since these domains contain at most p and q elements not in the

domain denoted by $\mathbf{B_S}$ respectively, $|dom(h*h_1)| \leq p+q+|FV(\phi)|$, and so $h*h_1 \in H'_{\phi,p+q}$. By induction, $\forall v \in vector_{\phi,p+q}(s, h*h_1)$, $v \models tran'(\phi_2)$.

It now remains to show that $(\mathbf{N_L} \oplus \mathbf{M_{L'}} \oplus (\mathbf{N_S}+\mathbf{M_S}), \mathbf{B_L} \oplus \mathbf{B_{L'}} \oplus \mathbf{B_S}, \mathbf{S}) \in vector_{\phi,p+q}(s, h*h_1)$.

We know that $(\mathbf{N_L} \oplus \mathbf{N_S}, \mathbf{B_L} \oplus \mathbf{B_S}, \mathbf{S}) \in vector_{\phi,p}(s,h)$ and that $(\mathbf{M_{L'}} \oplus \mathbf{M_S}, \mathbf{B_{L'}} \oplus \mathbf{B_S}, \mathbf{S}) \in vector_{\phi,q}(s,h_1)$. Therefore, we have the bijections R^h_L and R^h_S between h and $(\mathbf{N_L} \oplus \mathbf{N_S}, \mathbf{B_L} \oplus \mathbf{B_S}, \mathbf{S})$ (as defined in table 8.7), and similarly, the bijections $R^{h_1}_{L'}$ and $R^{h_1}_{S}$. We can define the appropriate bijections between $(\mathbf{N_L} \oplus \mathbf{M_{L'}} \oplus (\mathbf{N_S}+\mathbf{M_S}), \mathbf{B_L} \oplus \mathbf{B_{L'}} \oplus \mathbf{B_S}, \mathbf{S})$ and $vector_{\phi,p+q}(s, h*h_1)$ as follows:

$$\forall j \in 1..p \left[(l,j) \in R^{hh_1}_{LL'} \text{iff} (l,j) \in R^h_L\right]$$

$$\forall j \in 1..q \left[(l,p+j) \in R^{hh_1}_{LL'} \text{iff} (l,j) \in R^{h_1}_{L'}\right]$$

Similarly,

$$\forall j \in 1..|FV(\phi)| \left[(l,j) \in R^{hh_1}_{S} \text{iff} (l,j) \in R^h_S \vee (l,j) \in R^{h_1}_S\right]$$

It is clear that these bijections have the required properties. We know that for each $x \in FV(\phi)$, $v_x = s(x)$ since neither the stack nor the vector $\mathbf{S}$ differ.

$\Leftarrow$: we assume that the right-hand side of the iff holds. That is,

$$(\mathbf{N}, \mathbf{B}, \mathbf{S}) \models \forall \mathbf{M}, \mathbf{B_{L'}}. \left(\begin{array}{l} tran'(\phi_1)(\mathbf{M}, \mathbf{B_{L'}} \oplus \mathbf{B_S}, \mathbf{S}) \\ \wedge\ bounded(\mathbf{M_S}+\mathbf{N_S}) \wedge\ bounded(\mathbf{M}) \\ \wedge\ heap\left(\begin{array}{l} \mathbf{N_L} \oplus \mathbf{M_{L'}} \oplus (\mathbf{N_S}+\mathbf{M_S}), \\ \mathbf{B_L} \oplus \mathbf{B_{L'}} \oplus \mathbf{B_S} \end{array}\right) \\ \quad \Rightarrow tran'(\phi_2)\left(\begin{array}{l} \mathbf{N_L} \oplus \mathbf{M_{L'}} \oplus (\mathbf{N_S}+\mathbf{M_S}), \\ \mathbf{B_L} \oplus \mathbf{B_{L'}} \oplus \mathbf{B_S}, \\ \mathbf{S} \end{array}\right) \end{array}\right)$$

where $\mathbf{M} = \mathbf{M_{L'}} \oplus \mathbf{M_S}$, $|\mathbf{M_{L'}}| = max(|\phi_1|, |\phi_2|), |\mathbf{M_S}| = |FV(\phi)|$, and $|\mathbf{B_{L'}}| = 3(max(|\phi_1|, |\phi_2|))$.

We must now show that, for all heaps, $h_1 \in H'_{\phi,q}$, where $q = max(|\phi_1|, |\phi_2|)$, $h\#h_1$ and $s, h_1 \models \phi_1$, $s, h*h_1 \models \phi_2$.

So, for an $h_1 \in H'_{\phi,q}$, where $h\#h_1$ and $s, h_1 \models \phi_1$ hold, we know by induction that $\forall v \in vector_{\phi,q}(s,h_1) \models tran'(\phi_1)$. We know by the definition of $vector_{\phi,q}(s,h_1)$ that $v = (\mathbf{M}, \mathbf{B'}, \mathbf{S})$ where $\mathbf{M} = \mathbf{M_{L'}} \oplus \mathbf{M_S}$ and $\mathbf{B'} = \mathbf{B_{L'}} \oplus \mathbf{B_S}$. Since $h_1 \in H'_{\phi,q}$ and $s \in \mathcal{S}^{\phi}$ we also know by (2) that $bounded(\mathbf{M})$ and $heap(\mathbf{M_{L'}} \oplus \mathbf{M_S}, \mathbf{B_{L'}} \oplus \mathbf{B_S})$.

Since we know that $h\#h_1$ we can also deduce that $bounded(\mathbf{M_S}+\mathbf{N_S})$ and $heap(\mathbf{N_L} \oplus \mathbf{M_{L'}} \oplus (\mathbf{N_S}+\mathbf{M_S}), \mathbf{B_L} \oplus \mathbf{B_{L'}} \oplus \mathbf{B_S})$. $bounded(\mathbf{M_S}+\mathbf{N_S})$ holds because both $\mathbf{N_S}$ and $\mathbf{M_S}$ apply to the same vector of cells, $\mathbf{B_S}$. If

$bounded(\mathbf{M_S}+\mathbf{N_S})$ did not hold, then there would be a cell that occurs in both heaps, and hence their domains will not be disjoint and $h\#h_1$ will not hold. Similarly, $heap(\mathbf{N_L}\oplus\mathbf{M_{L'}}\oplus(\mathbf{N_S}+\mathbf{M_S}),\mathbf{B_L}\oplus\mathbf{B_{L'}}\oplus\mathbf{B_S})$ holds because we know that $heap(\mathbf{N_L}\oplus\mathbf{N_S},\mathbf{B_L}\oplus\mathbf{B_S})$ and $heap(\mathbf{M_{L'}}\oplus\mathbf{M_S},\mathbf{B_{L'}}\oplus\mathbf{B_S})$ hold. Because these properties hold, $heap(\mathbf{N_L}\oplus\mathbf{M_{L'}}\oplus(\mathbf{N_S}+\mathbf{M_S}),\mathbf{B_L}\oplus\mathbf{B_{L'}}\oplus\mathbf{B_S})$ will not hold only if the two heaps do not have disjoint domains. Since $h\#h_1$, this is never the case.

By our original assumption, and the properties shown above, we can conclude that $tran'(\phi_2)(\mathbf{N_L}\oplus\mathbf{M_{L'}}\oplus(\mathbf{N_S}+\mathbf{M_S}),\mathbf{B_L}\oplus\mathbf{B_{L'}}\oplus\mathbf{B_S},\mathbf{S})$ holds. Since $(\mathbf{N_L}\oplus\mathbf{N_S},\mathbf{B_L}\oplus\mathbf{B_S},\mathbf{S})\in vector_{\phi,p}(s,h)$ and $(\mathbf{M_{L'}}\oplus\mathbf{M_S},\mathbf{B_{L'}}\oplus\mathbf{B_S},\mathbf{S})\in vector_{\phi,q}(s,h_1)$, we know that $(\mathbf{N_L}\oplus\mathbf{M_{L'}}\oplus(\mathbf{N_S}+\mathbf{M_S}),\mathbf{B_L}\oplus\mathbf{B_{L'}}\oplus\mathbf{B_S},\mathbf{S})\in vector_{\phi,p+q}(s,h*h_1)$ (this was shown in the proof for $\Rightarrow$). Hence, by induction, $s,h*h_1\models\phi_2$, as required.

Finally, we show that, for an assertion, ϕ,

$$[\forall(s,h).s,h\models\phi]\iff[\forall(\mathbf{N},\mathbf{B},\mathbf{S}).(\mathbf{N},\mathbf{B},\mathbf{S})\models tran(\phi)]$$

That is,

$$[\forall(s,h).s,h\models\phi]\iff\left[\forall(\mathbf{N},\mathbf{B},\mathbf{S}).(\mathbf{N},\mathbf{B},\mathbf{S})\models\left(\begin{array}{l}bounded(\mathbf{N})\wedge heap(\mathbf{N},\mathbf{B})\\ \Rightarrow tran'(\phi)(\mathbf{N},\mathbf{B},\mathbf{S})\end{array}\right)\right]$$

- $\Rightarrow$: suppose, $\forall(s,h)(s,h\models\phi)$. Also assume that $(\mathbf{N},\mathbf{B},\mathbf{S})\models(bounded(\mathbf{N})\wedge heap(\mathbf{N},\mathbf{B}))$. Then, by (1), $(\mathbf{N},\mathbf{B},\mathbf{S})\in vector_{\phi,|\phi|}(s,h)$ and $(s,h)\in\mathcal{S}^\phi\times H'_{\phi,|\phi|}$. We have already proved that, for all $(s,h)\in\mathcal{S}^\phi\times H'_{\phi,|\phi|}$,

  $$\big[(s,h\models\phi)\iff\forall(\mathbf{N},\mathbf{B},\mathbf{S})\in vector_{\phi,|\phi|}(s,h)(\mathbf{N},\mathbf{B},\mathbf{S})\models tran'(\phi)\big]$$

  And, since $\forall(s,h)(s,h\models\phi)$, we can conclude, $(\mathbf{N},\mathbf{B},\mathbf{S})\models tran'(\phi)$.

- $\Leftarrow$: suppose,

  $$\forall(\mathbf{N},\mathbf{B},\mathbf{S}).(\mathbf{N},\mathbf{B},\mathbf{S})\models(bounded(\mathbf{N})\wedge heap(\mathbf{N},\mathbf{B})\Rightarrow tran'(\phi)(\mathbf{N},\mathbf{B},\mathbf{S}))$$

  To test validity, we only need to consider states, $(s,h)\in\mathcal{S}^\phi\times H'_{\phi,|\phi|}$. We know that, for all $(s,h)\in\mathcal{S}^\phi\times H'_{\phi,|\phi|}$,

  $$\left[\begin{array}{l}(s,h\models\phi)\iff\\ \forall(\mathbf{N},\mathbf{B},\mathbf{S})\in vector_{\phi,|\phi|}(s,h)(\mathbf{N},\mathbf{B},\mathbf{S})\models\left(\begin{array}{l}bounded(\mathbf{N})\wedge heap(\mathbf{N},\mathbf{B})\\ \Rightarrow tran'(\phi)(\mathbf{N},\mathbf{B},\mathbf{S})\end{array}\right)\end{array}\right]$$

  We also know that $vector_{\phi,|\phi|}(s,h)(\mathbf{N},\mathbf{B},\mathbf{S})\models(bounded(\mathbf{N})\wedge heap(\mathbf{N},\mathbf{B})\Rightarrow tran'(\phi)(\mathbf{N},\mathbf{B},\mathbf{S}))$ by the original assumption. Therefore, for all $(s,h)\in\mathcal{S}^\phi\times H'_{\phi,|\phi|}(s,h\models\phi)$. Consequently, $\forall(s,h)(s,h\models\phi)$, as required.

■

Appendix C

User Guide

In this section we present an introductory guide to the use of the tool produced during this project. We assume that the reader has knowledge of simple logic (conjunction, implication, etc.), but not of the Tree Logic itself.

We begin by introducing the notion of a tree. We then discuss the Tree Logic, followed by a discussion of the different commands offered by the tool. Finally we give several simple examples illustrating the use of the tool.

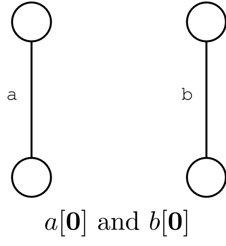
C.1 Trees

The representation of tree structures used by this tool is a simple one. We begin with an "empty tree", that is, a tree that has nothing in it. A tree can be built from the empty tree by adding a branch to the empty tree. Trees with more than one branch can be built by joining to smaller trees together. There are two ways to join trees together: they can be placed in parallel with each other, or they can be stacked, one on top of the other.

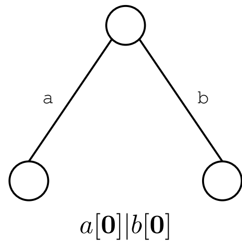
Figure C.1 shows the construction of a tree using these methods. First two trees with a single branch are created ($a[\mathbf{0}]$ and $b[\mathbf{0}]$). The notation, $a[\mathbf{0}]$ means that a branch, labelled a, has been put on top of the empty tree, resulting in a tree with a single branch. We then compose these two trees together using the composition ($|$) operator. The results in a tree with two branches, labelled a and b. It is worth noting that many branches may share the same label, and that there is no restriction upon the use of mutual labels. Finally, we construct a deeper tree by putting two copies of this tree on top of each other.

C.2 Tree Logic

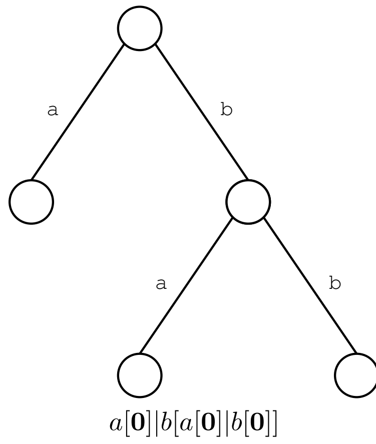
We use the Tree Logic to make assertions about trees. We can think of Tree Logic formulae as a type system for tree languages: if a tree satisfies a formula, A, then the tree is of type A, or that the tree matches type A.



$a[\mathbf{0}]$ and $b[\mathbf{0}]$



$a[\mathbf{0}]|b[\mathbf{0}]$



$a[\mathbf{0}]|b[a[\mathbf{0}]|b[\mathbf{0}]]$

Figure C.1: Constructing trees

$\mathcal{A}, \mathcal{B} ::=$	Formula
$\mathbf{F}$	False
$\mathcal{A} \wedge \mathcal{B}$	Conjunction
$\mathcal{A} \Rightarrow \mathcal{B}$	Implication
$\mathbf{0}$	Void
$\mathcal{A} \vert \mathcal{B}$	Composition
$\mathcal{A} \triangleright \mathcal{B}$	Guarantee
$n[\mathcal{A}]$	Location
$\mathcal{A}@n$	Placement

Table C.1: The Tree Logic syntax

This logic consists of the usual logical connectives (conjunction, disjunction, etc.) and adds several connectives that reason specifically about trees. The composition ($|$) and branch ($a[A]$) connectives correspond to the structure of trees directly. A formula of the form $a[A]$ matches any tree of the form $a[D]$, where D is a tree that matches the sub-formula, A. Similarly, a formula of the form, $A|B$, matches any tree of the form $D|D'$, where D is a tree that matches the formula, A, and D' matches B. At this point we should note that the order of the sub-trees does not matter. We can decompose a tree in anyway that we require. For example, a tree of the form, $D|D'|D''$, can be split into several pairs of trees: $(D|D', D''), (D.D'|D''), (D|D'', D')$ and all possible permutations of these pairs.

These two connectives have their adjuncts, guarantee ($\triangleright$) and placement(@). The guarantee connective is written $A \triangleright B$. A tree will match this formula if it will match B when composed with any tree that matches A. This can be vacuously true: if no trees match A, then $A \triangleright B$ will holds for any tree. For example, we may have a formula B that matches a deck of cards. Formula A may represent the set of black cards, and so $A \triangleright B$ matches the set of red cards, since, when the black cards are added, the result matches a complete deck.

The placement operator is written $A@a$. A tree matches this formula if the tree formed by placing a branch labelled a atop the tree matches the formula A. For example, if the formula A represents a box containing several toys, the formula, $A@box$, will match a collection of toys, since, when the box branch is put on top of the toys, the resulting tree is a box, with several toys as a sub-tree.

The full syntax and semantics of the Tree Logic are given in tables C.2 and C.2 respectively.

C.3 Using the Tool

The tool produced as part of this project has a command line interface. Input is read from the standard input, and output is sent to the standard input.

$d \models \mathbf{F}$		Never
$d \models \mathcal{A} \wedge \mathcal{B}$	$\triangleq$	$d \models \mathcal{A} \wedge d \models \mathcal{B}$
$d \models \mathcal{A} \Rightarrow \mathcal{B}$	$\triangleq$	$d \models \mathcal{A} \Rightarrow d \models \mathcal{B}$
$d \models \mathbf{0}$	$\triangleq$	$d \equiv \mathbf{0}$
$d \models \mathcal{A} \vert \mathcal{B}$	$\triangleq$	$\exists d', d''. d \equiv d' \vert d'' \wedge d' \models \mathcal{A} \wedge d'' \models \mathcal{B}$
$d \models \mathcal{A} \triangleright \mathcal{B}$	$\triangleq$	$\forall d'. d' \models \mathcal{A} \Rightarrow d' \vert d \models \mathcal{B}$
$d \models n[\mathcal{A}]$	$\triangleq$	$\exists d'. d \equiv n[d'] \wedge d' \models \mathcal{A}$
$d \models \mathcal{A}@n$	$\triangleq$	$n[d] \models \mathcal{A}$

Table C.2: Satisfaction of Tree Logic formulae

There are four different types of actions that this tool can perform: a typing test, a subtype test, a satisfiability test and a validity test. Each command is terminated by a semi-colon. Additionally, the commands may be preceded with the keywords `EXPTRUE` or `EXPFALSE`. These keywords tell the tool to expect that the result of the following command will be true or false respectively. The output of the tool will say that the outcome has met expectations, or it has not — rather than giving the true or false answer.

The four different commands are explained below:

- `VALID` — a validity problem takes a tree logic formula as its input and outputs 'true' if the formula is valid, 'false' otherwise.
- `SATISFIABLE` — a satisfaction problem takes a tree logic formula as its input and outputs 'true' if the formula is satisfiable, 'false' otherwise.
- `SUBTYPE` — a sub-typing problem takes two tree logic formulae, $\mathcal{A}$ and $\mathcal{B}$ as its input and returns 'true' if the formula $\mathcal{A}$ represents a sub-type of $\mathcal{B}$. This is equivalent to testing the validity of $\mathcal{A} \Rightarrow \mathcal{B}$.
- `OFTYPE` — a typing problem takes a tree, d, and a formula, $\mathcal{A}$ as its input, and returns 'true' if $d \models \mathcal{A}$, and 'false' otherwise.

The full grammar of the input that this tool accepts is given in table C.3.

C.4 Examples

In this section we present three simple examples to illustrate the use of the tool.

C.4.1 Types

Suppose we are dealing with an organiser. The organiser may contain meeting records, that contain the details of a person and the details of a meeting.

commands:	
problem `;`	A command terminated with a semi-colon.
`EXPTRUE` *problem* `;`	A command whose result is expected to be 'True'.
`EXPFALSE` *problem* `;`	A command whose result is expected to be 'False'.
problem:	
`VALID` *tl*	A validity problem.
`SATISFIABLE` *tl*	A satisfiability problem.
tl `SUBTYPE` *tl*	A sub-typing problem
tree `OFTYPE` *tl*	A satisfaction problem
`EXIT`	Quit the program.
tree:	
`0`	Void
a[*tree*]	Branch
tree `|` *tree*	Composition
`(`*tree*`)`	Parenthesis
tl:	
`0`	Void
`T`	Truth
`F`	Falsity
a[*tl*]	Branch
tl`@`*a*	Placement
tl `|` *tl*	Composition
tl `->` *tl*	Implication
tl `AND` *tl*	Conjunction
tl `OR` *tl*	Disjunction
tl `|>` *tl*	Guarantee
`NOT` *tl*	Negation
`(`*tl*`)`	Parenthesis

Table C.3: The input grammar

For simplicity, we will represent a person using the formula, $person[\mathbf{0}]$, and a meeting using the formula, $(person[\mathbf{0}]|details[\mathbf{0}])$. When given a tree, we may check that it represents a valid set of details using the formula, $person[\mathbf{0}] \triangleright (person[\mathbf{0}]|details[\mathbf{0}])$. (Of course, in this case, the formula, $details[\mathbf{0}]$ would suffice. For the purposes of this example, we assume that we don't know the format of *details* precisely.)

If we have a tree, $details[\mathbf{0}]$, we may want to ensure that we can construct a valid meeting record from it. Hence, we would give the tool the following command:

```
details[0] OFTYPE person[0] |> (person[0] | details[0]);
```

The following output is produced:

```
> details[0] OFTYPE person[0] |> (person[0] | details[0]);
Typing problem:  details[0] :  (person[0] |> (person[0] | details[0]))
true
```

C.4.2 Sub-types

We may wish to use the tool to check whether the type denoted by one formula is a sub-type of another. Suppose that we are dealing with *people* records, where each person simply has a name. We could represent a person using the following formula: $person[name[\mathbf{0}]]$. We may the wish to describe a musician, who has an instrument as well as a name. A musician can be represented with the formula, $person[name[\mathbf{0}]|ins[\mathbf{0}]]$. To check if a musician is a sub-type of a person we use the following command:

```
person[name[0] | ins[0]] SUBTYPE person[name[0]];
```

The following output is produced:

```
> person[name[0] | ins[0]] SUBTYPE person[name[0]];
Subtyping problem:  person[(name[0] | ins[0])] <= person[name[0]]
false
```

Note that the answer returned is 'false'. This is because our formula for a person is restricted to people who only have names. To correct the situation, the formula, $person[name[\mathbf{0}]|\mathbf{T}]$, should be used to represent a person.

C.4.3 Validity

Suppose that we have a formula, $box[toys[\mathbf{0}]]$, that denotes a box of toys. We may wish to query whether any toys that are put in a box represents a box of toys. This requires the following formula: $toys[\mathbf{0}] \Rightarrow box[toys[\mathbf{0}]]@box$, and the command,

```
VALID toys[0] -> box[toys[0]]@box
```

The output from the tool is,

```
> VALID toys[0] -> box[toys[0]]@box;
Validity problem:  (toys[0] -> (box[toys[0]])@box)
true
```