LOGICAL ABSTRACT INTERPRETATION

VIJAY D'SILVA

MAGDALEN COLLEGE

Dissertation for the degree of Doctor of Philosophy

Trinity 2012

In Memoriam

M. Srinivasa Char

1922 – 2007

ಬಾಲ ಹೋಯ್ತು ಕತ್ತಿ ಬಂತು
ಡುಂ ಡುಂಕಿ ಡುಂ

CREDIT

Credit for technical content and presentation of this dissertation is due to

MEHRNOOSH SADRZADEH for a discussion that led to Example 3.35 and Theorem 3.44.

SAM VAN GOOL for a tutorial on the representation of non-distributive lattices, which led to the material on conjunctive state structures.

LEOPOLD HALLER for two years of joint work that underlies the entire content of Chapter 6 and for Figure 24.

MITRA PURANDARE who got me to work on interpolation and who discovered the properties of proofs and interpolants arising in practice that led to Section 7.3.

GEORG WEISSENBACHER for joint work in proving the results in Section 7.3 and for Figures 21, 22 and 23.

ALASTAIR DONALDSON for asking if there was a sense in which existing interpolation systems were special, which I answered with Theorem 7.30.

GRETA YORSH for all round assistance with Chapter 7.

VINCENT NIMAL and HRISTINA PALIKAREVA for proof reading Chapters 2 and 3.

ALAN MYCROFT and SAMSON ABRAMSKY for detailed corrections, comments and suggestions.

ALGEBRAIC LOGICIANS for a deep and beautiful mathematical tradition that goes back to George Boole, especially for discrete duality theory.

PATRICK and RADHIA COUSOT for abstract interpretation.

ACKNOWLEDGEMENTS

Completing this degree and writing this dissertation have both taken significantly longer than I ever imagined. The process was made bearable through the efforts of friends, colleagues, the people around me in Oxford, the academic research community, and family. I was financially supported by a Microsoft Research European PhD scholarship.

The DPhil was long. Dark. Wearisome. The time in Oxford, short. Enchanted. Fleeting. The place, the people, the lab; always present, always close, through sun, through rain and mostly through the thing in between that is neither. If you were there, I am richer for it. As I leave, your echoes carry me. For fear of missing you out, I have made the perverse choice to mention only my colleagues. Forgive me, my friend.

In the Land of Oxford where Logicians lie is a lane under a bridge both by the name Logic. Logician is an orientation and their density is high in the Land of Oxford where Logicians lie. You can study every type of computing as long as it is logic. Everything has a reason and nothing makes sense. I came of age, I found my calling; logician am I. In the Land of Oxford where Logicians lie.

This journey has been a long one. This journey has been a shared one. I set out accompanied by Horatiu Jula, Gerard Basler, Nicolas Blanc, Georg Weissenbacher, Christoph Wintersteiger, Mitra Purandare, Angelo Brillout, Thomas Wahl, and Michele Mazzucchi in Zurich. The most intense highs and lows were borne in the company of Georg Weissenbacher, Thomas Wahl, Leopold Haller, Alastair Donaldson, Alexander Kaiser, and Philipp Ruemmer in Oxford. The final stretch. That long and final stretch is peppered by glimpses of Michael Tautschnig, Jade Alglave, Vincent Nimal, Ajitha Rajan, David Landsberg and Matt Lewis, and tea with Pamela Farries. There is a notion of academic family. They define it. Accost me in heat of day and assert "*Your advisor is ...*" and I will surely interject " *... a grand old patriarch in the true, academic sense*," although he is neither old nor grand, and that expression does not make any sense, academic or otherwise.

The journey did not begin in Oxford. Im Anfang war Marco Vrankic. And Marco Vrankic gave me a family. And Marco Vrankic is family. And so are you, Jonathan and Dana Landau. I will not hurry back, but you are with me.

I talk a lot about family. What of my own? I wish I could say I was with my family through it all. This DPhil has witnessed five weddings, four births, two cremations and a burial. The family I left is so different from the family I go back to now and I know that it is I who was not with them. I have travelled somewhere beyond your experience. The voice of your warmth is more constant than the grass. My parents. My sister. My uncles. My aunts. My cousins. My grandparents. My grandparents who showed me that life is an immense labour of love. My grandparent to whom I cannot return. My grandparent whose most fragile gestures enclose me. They have been waiting for me at the gates, but I should not forget those who brought me here.

Daniel Kroening took me on as an irresponsible Masters student and tolerated me as an irresponsible DPhil student. I have seen things you people wouldn't believe. *He codes without syntax highlighting*. His feet assemble x86 faster than the rest of us armed with GCC. *He codes without syntax highlighting*. I have watched vodka get drunk on Daniel to lift our spirits before a paper

deadline. *He codes without syntax highlighting*. We watched in awe as device drivers terminated because he thought they should. *He codes without syntax highlighting*. Time to graduate.

The Oxford academic system has been very generous with my various deadline transgressions and despite the eternity it has taken me, it would have been much longer if not for the firm yet gentle admonishment of Marta Kwiatkowska and Luke Ong. I received extensions when I needed them and learned to do without them when I didn't. The administrative staff of the Oxford University Computing Laboratory have been a model of kindness and efficiency. Shoshannah Holdom provided great assistance with teaching issues and turned a rather miserable situation into a fantastic one. One day, when Shoshannah Holdom is in charge of a modern university, I hope she sees fit to hire me. I cannot sing the praises of Julie Sheppard enough. Behind every graduating Computing Laboratory graduate student is Julie Sheppard.

The staff of Magdalen College has been equally generous and efficient. For all my years did Catherine Hughes shelter me in rooms with views. The bursary and the tutor for graduates have always been prompt in their assistance. In a moment of intense turmoil, Michael Piret came through to tell me that I need to feel in my own skin as I would with an old leather slipper. The image was a great comfort to me and I have never forgotten it.

My transfer and confirmation examiners Luke Ong and Ben Worrell were always supportive of my plans. I am particularly grateful to Luke for his sound counsel to include the chapter on interpolation. When I read the abstract of *Domain Theory and the Logic of Observable Properties*, I was overcome with great awe and a feeling of kinship. When I read *Completeness and Predicate-based Abstract Interpretation*, I dropped everything else I had been doing because I knew I found a dissertation topic. When Samson Abramsky and Alan Mycroft agreed to be my examiners, the circle was complete. They read my dissertation in such detail I feel flattered. I am deeply indebted to them for their patience and their kind mentoring. I hope they the final result meets their recommendations.

No mention of mentors is complete without S. Ramesh and Arcot Sowmya, who gave me my first break and brought me up to the trees. Sriram Rajamani hosted me twice in Microsoft Research, India, and gave me time to study the work of Roberto Giacobazzi and Francesco Ranzato, even though the connection to my visit was unclear. When the storm comes crashing in and the candle is about to flicker out, please intern with Sriram, and you will see that one can make it to the top and keep smiling. Aditya Nori believes in me, which is a gift that has seen me through many a dark and lonely night. I am here now because they were there then.

If the work before you is deemed worthy spare a thought for those who taught me worth. I did not know what detail was until I saw through the eyes of Felix Klaedtke. My physics teacher, Minnie James, set me on the path of technical illustration when she taught me to draw tangents in ray diagrams. Greta Yorsh put a stop to my circumlocution when she shepherded one of my papers. All my subsequent work has featured her as an imaginary reader.

It is time to talk about the work. A few words on a page cannot make up for the time I stole from Matthew Hague, Mehrnoosh Sadrzadeh, Nikos Tzevelekos, and Ben Worrell. I owe you and it may well be that the debt is never repaid. Ben is the soul and spirit of the lab. He is the sprite that moves between the walls. In shorts.

The greatest walls have, however, been the review process. Reviewers have not always been kind in their reception of this work. It appears too easy to

spew aggressive, negative rhetoric with complete disregard for an author's humanity. I have made my peace with the reviews I have received and hope I never subject my fellow researchers to the same.

Andy Gordon consoled on the worst occasion, while David Basin and Felix Klaedtke gave me very sensible advice. I found solace when Marta Kwiatkowska said a review was unacceptable but her strongest contribution to my education was to show me how to extract constructive feedback from a hurtful review. I spent the winter break of 2009 internalising Thomas Wahl's vivid advice to put the pain behind me. Prakash Panangaden is my angel of algebra among the demons of peer review. He is my guardian angel who helps me even when I don't help myself, particularly when I think my research fills a much needed gap in the literature.

This work owes its existence to the work of Patrick and Radhia Cousot on abstract interpretation. I know the names of many things but the few I understand I saw through the lens of abstract interpretation. I arrived in Paris on the 17^{th} of June 2011 with intense trepidation only to be received with a warmth and generosity I have only known from family and to be accorded the great honour of being treated as an equal. There is no greater triumph than to meet your maker and hear, "*We know that you know.*" It has been a year and I remind myself every single day: they saw that it was good. I muddled my way up their shoulders for a breathtaking view of deafening depth and painful intricacy that only the sunlight can understand. Then, I realised I was standing on their toes. I weep in awe of their loneliness in a world whose sight cannot follow where their vision went.

While I was exploring aloneness and isolation in the Lands of Logic, Leopold Haller found me and dragged me through the Ages of Abstraction where the seeds of Truth grow into a tree of Proof, which blooms bright with the flowers of And and Or, and grows heavy with the fruit of All and Some. When we sowed with Least and harvested with Greatest, our minds could not cope with the dream of Widening, nor our hearts with the reality of Narrowing. It drove us mad. So we tore apart our tortured souls and draped them on the stars and set the night sky on fire. The light was beautiful.

Vijay D'Silva
Berkeley, California

ABSTRACT

Logical deduction and abstraction from detail are fundamental, yet distinct aspects of reasoning about programs. This dissertation shows that the combination of logic and abstract interpretation enables a unified and simple treatment of several theoretical and practical topics which encompass the model theory of temporal logics, the analysis of satisfiability solvers, and the construction of Craig interpolants. In each case, the combination of logic and abstract interpretation leads to more general results, simpler proofs, and a unification of ideas from seemingly disparate fields.

The first contribution of this dissertation is a framework for combining temporal logics and abstraction. Chapter 3 introduces trace algebras, a new lattice-based semantics for linear and branching time logics. A new representation theorem shows that trace algebras precisely capture the standard trace-based semantics of temporal logics. We prove additional representation theorems to show how structures that have been independently discovered in static program analysis, model checking, and algebraic modal logic, can be derived from trace algebras by abstract interpretation.

The second contribution of this dissertation is a framework for proving when two lattice-based algebras satisfy the same logical properties. Chapter 5 introduces functions called subsumption and bisubsumption and shows that these functions characterise logical equivalence of two algebras. We also characterise subsumption and bisubsumption using fixed points and finitary logics. We prove a representation theorem and apply it to derive the transition system analogues of subsumption and bisubsumption. These analogues strictly generalise the well studied notions of simulation and bisimulation. Our fixed point characterisations also provide a technique to construct property preserving abstractions.

The third contribution of this dissertation is abstract satisfaction, an abstract interpretation framework for the design and analysis of satisfiability procedures. We show that formula satisfiability has several different fixed point characterisations, and that satisfiability procedures can be understood as abstract interpreters. Our main result is that the propagation routine in modern SAT solvers is a greatest fixed point computation involving abstract transformers, and that clause learning is an abstract transformer for a form of negation.

The final contribution of this dissertation is an abstract interpretation based analysis of algorithms for constructing Craig interpolants. We identify and analyse a lattice of interpolant constructions. Our main result is that existing algorithms are two of three optimal abstractions of this lattice. A second new result we derive in this framework is that the lattice of interpolation algorithms can be ordered by logical strength, so that there is a strongest and a weakest possible construction.

PUBLICATIONS

Part of the material in this dissertation has been published in the papers listed below.

1. *Interpolant Strength*, authored in collaboration with Daniel Kroening, Mitra Purandare and Georg Weissenbacher, appeared in the proceedings of the conference on Verification, Model Checking, and Abstract Interpretation in 2010, and includes part of the material in Chapter 7.
2. *Propositional Interpolation and Abstract Interpretation* appeared in the proceedings of the European Symposium on Programming in 2010, and includes part of the material in Chapter 7.
3. *Satisfiability Solvers are Static Analysers*, authored in collaboration with Leopold Haller and Daniel Kroening, appeared in the proceedings of the Symposium on Static Analysis in 2012, and includes part of the material in Chapter 6.
4. *Abstract Conflict Driven Learning*, authored in collaboration with Leopold Haller and Daniel Kroening, will appear in the proceedings of the Symposium on Principles of Programming Languages in 2013, and includes part of the material in Chapter 6.
5. *Abstraction of Syntax*, authored in collaboration with Daniel Kroening, will appear in the proceedings of the conference on Verification, Model Checking, and Abstract Interpretation in 2013, and includes part of the material in Chapter 4.

CONTENTS

1 INTRODUCTION 17
- 1.1 Towards a Marriage of Logic and Abstraction 18
- 1.2 Contribution and Overview 22

2 PRELIMINARIES 25
- 2.1 Sets and Functions 26
- 2.2 Lattices 27
- 2.3 Galois Connections 30
- 2.4 Representations of Lattices as Sets 32
- 2.5 Algebras 37
- 2.6 Abstract Interpretation 39
- 2.7 Bibliographic Notes 41

3 STRUCTURES 45
- 3.1 Overview 46
- 3.2 Propositional Structures 48
- 3.3 State Structures 56
- 3.4 Trace Structures 73
- 3.5 Bibliographic Notes 83

4 LANGUAGES 87
- 4.1 Overview 88
- 4.2 A Meta-Language for Syntax 90
- 4.3 Syntactic Construction of Abstract Domains 100
- 4.4 Temporal Logics 105
- 4.5 Bibliographic Notes 110

5 PROPERTIES 113
- 5.1 Overview 114
- 5.2 Subsumption 116
- 5.3 Bisubsumption 124
- 5.4 Representation Theorems 127
- 5.5 Abstraction 132
- 5.6 Bibliographic Notes 135

6 SOLVERS 137
- 6.1 Overview 138
- 6.2 Abstract Satisfaction 139
- 6.3 Satisfiability Routines 143
- 6.4 Bibliographic Notes 150

7 INTERPOLANTS 153
- 7.1 Overview 154
- 7.2 Propositional Interpolants 155
- 7.3 A Coloured Interpolation System 160
- 7.4 Interpolation and Abstract Interpretation 166
- 7.5 Logical Strength and Variable Elimination 172
- 7.6 Bibliographic Notes 175

8 CONCLUSION 177
- 8.1 Some Logical Conclusions 178
- 8.2 Immediate extensions 178
- 8.3 Longer Term Extensions 180

Index of Symbols 189

Index of Technical Terms 190

1

INTRODUCTION

Geschrieben steht: »Im Anfang war das Wort!«
Hier stock ich schon! Wer hilft mir weiter fort?
Ich kann das Wort so hoch unmöglich schätzen,
Ich muß es anders übersetzen,
Wenn ich vom Geiste recht erleuchtet bin.
Geschrieben steht: Im Anfang war der Sinn.
Bedenke wohl die erste Zeile,
Daß deine Feder sich nicht übereile!
Ist es der Sinn, der alles wirkt und schafft?
Es sollte stehn: Im Anfang war die Kraft!
Doch, auch indem ich dieses niederschreibe,
Schon warnt mich was, daß ich dabei nicht bleibe.
Mir hilft der Geist! Auf einmal seh ich Rat
Und schreibe getrost: Im Anfang war die Tat!

– Johann Wolfgang von Goethe, *Faust: Eine Tragödie*,
Erster Teil (ll. 1224-1237)

In the beginning was Logic.
And Logic was with God,
and Logic was God.

– *John 1:1*, translation by Gordon H. Clark

1.1 TOWARDS A MARRIAGE OF LOGIC AND ABSTRACTION

The analysis of programs involves the two fundamentally distinct activities of logical reasoning and abstraction from detail. The field of mathematical logic provides a rigorous basis for logical reasoning, and the theory of abstract interpretation does the same for abstraction from detail. There has, however, been little interaction between research in mathematical logic and abstract interpretation. This dissertation shows that a systematic, rigorous combination of mathematical logic with the theory of abstract interpretation provides a powerful, comprehensive framework for studying problems at the intersection of logic, program analysis and automated deduction.

The four problems we study are the development of algebraic semantics for temporal logics with linear- and branching-time modalities, the characterisation of algebraic structures that satisfy the same temporal logic properties, the analysis of practical Boolean satisfiability solvers and the construction of interpolants in propositional logic. For each problem we unify and generalise existing results by combining a new lattice-based semantics for the problem studied with abstract interpretation.

The problems mentioned above may appear to be theoretical but are all motivated by practical issues in program analysis and verification. The framework developed in this dissertation contains the mathematical tools to address these issues. The practical motivation behind the work in this dissertation is discussed next. The reader should be aware that not all the presented issues are resolved in this dissertation.

Model Checking and Static Analysis

The first motivating problem is the disparity between two families of techniques for program verification. One family of techniques is called model checking and another is called static analysis. The model checking problem was originally defined to be that of checking if a transition system satisfies a correctness property expressed in a temporal logic. The term model checking is now used for the broader problem of checking if programs satisfy correctness properties. The term static analysis refers to techniques that compute information about a program without running the program. In this dissertation, static analysis will refer to techniques that compute approximations of fixed points using lattices and transformers.

Model checkers and static analysers solve similar problems but are perceived as being different. A key feature of model checkers is their ability to generate counterexamples when a program violates a correctness property. A key feature of static analysers is their use of imprecise abstractions and generalisation techniques to compute program invariants. At present, there exist results about the equivalence of specific model checking and specific static analysis techniques [Steffen 1991; Schmidt 1998; Schmidt and Steffen 1998; Nielson and Nielson 2010]. However, there are no generic results that cover majority of the static analysis techniques that exist.

The disparity between model checkers and static analysers is both a practical and a conceptual problem. The ideal program verifier should combine the strengths of model checkers and static analysers. It should be able to check properties written in a rich specification language, compute invariants and generate counterexample traces. The conceptual obstacle to such a unification is that model checkers and static analysers appear to operate on different kinds of mathematical objects.

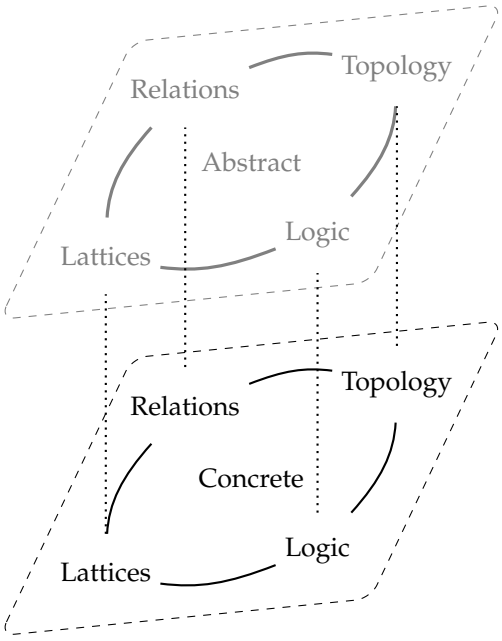


Figure 1: A perspective on Stone duality and abstraction. Structures related by Stone duality reside in the same plane and have the same information content. An abstraction of a structure contains less information. The picture shows that one can navigate between different structures in different planes by combining Stone duality and abstract interpretation.

This dissertation proposes a theoretical foundation for unifying model checking and static analysis techniques. A model checker takes as input a temporal logic formula and a transition system and uses data structures that represent sets of states of the transition system. A static analyser takes as input a program and computes properties of the program using a lattice and functions on the lattice called transformers. One route to unification is to find a translation between transition systems and lattices with transformers. An answer to this question has practical value because it provides a translation between model checking and static analysis.

The solution we present in this dissertation is to use discrete duality theory to translate between structures based on transition systems and structures based on lattices. Duality theory is the mathematical study of translations between mathematical structures. Stone duality is a specific family of results for translating between set-theoretic, lattice-theoretic, logical and topological structures. A conceptual contribution of this dissertation is to use the duality theory of algebraic logic to translate between structures used in model checkers and in static analysers. The solid lines in Figure 1 represent the translations provided by Stone duality.

A classic result in duality theory is a theorem of Jónsson and Tarski [1952] which can be applied to show that transition systems can be represented by complete, atomic, Boolean lattices with operators satisfying distributivity properties (BAOs). If the lattice in the Jónsson and Tarski theorem is considered as a lattice used in static analysis, the operators correspond to transformers. The representation theorem of Jónsson and Tarski shows that there is a systematic translation between model checking techniques

and static analysis over complete, atomic Boolean lattices. Weaker forms of this result have been rediscovered in model checking [Loiseaux et al. 1995] and static analysis [Cousot 1981]. What of non-Boolean lattices that are omnipresent in static analysis?

Stone duality has been studied for distributive lattices with operators [Abramsky 1987; Gehrke and Jónsson 1994], and non-distributive lattices [Dunn et al. 2005], albeit more recently. These results show that transition systems extended with an order and additional conditions can represent non-Boolean lattices with operators satisfying distributivity conditions. Figure 1 illustrates the relationship between these results and program verification. While non-Boolean lattices are used in program verification, they usually arise as abstractions of Boolean lattices. By combining abstract interpretation with Stone duality, ideas from model checking and static analysis can be lifted to non-Boolean lattices.

There is one impediment to linking existing results to obtain a translation of the form above. Branching-time logics such as CTL and classical modal logic can be interpreted either over states in a transition system or over a lattice of states. In a logic like CTL*, some formulae are interpreted over states and some over traces. Providing a lattice-based semantics for such a logic is non-trivial and there are only few proposals in the literature [Möller et al. 2006; von Karger 2002].

Chapter 3 is the first step towards a systematic translation between model checking for linear- and branching-time logics and static analysis techniques. The chapter introduces a new model, called a *trace algebra* and proves a representation theorem showing that trace algebras capture the standard semantics of temporal logics. The chapter also shows that various abstractions of transition systems arise as abstractions of trace algebras and identifies relational representations for these abstractions.

Compiler Optimisation

The second motivation for problems studied in this dissertation comes from two problems encountered in compiler optimisation. The first problem is to build an automated system to prove the correctness of compiler optimisations. Existing systems, such as those of Lacey et al. [2004] and Kundu et al. [2009] use techniques based on bisimulation to show that the source and target of a compiler optimisation have the same semantics. Practical evidence shows that such techniques are appropriate for the problem. However, the equivalence relation that is used varies with the optimisation considered. For example, Guo and Palsberg [2011] use a relation that mixes states and traces to study specific forms of Just-In-Time compilation. There is a need for a formal framework in which we can generate the notion of equivalence required for proving an optimisation correct and also generate logical characterisations usually required of such an equivalence.

The second compiler optimisation problem is about determining the algorithmic complexity required to implement a specific optimisation. A classic optimisation technique, *Partial Redundancy Elimination* (PRE), is used by compilers to soundly eliminate expressions that are redundant on some but not all paths of a program. PRE can be implemented in several ways, including a bidirectional data-flow analysis or a series of forwards and backwards analyses. There is currently no consensus about whether a bidirectional analysis is strictly necessary. See [Muchnick 1997] for a discussion.

It is well known that several compiler optimisations can be described in temporal logics. It is also known that techniques based on bisimulation or logical games can be used to prove if a certain property cannot be expressed in a logic. One route to proving what kind of analysis is required for PRE is to characterise PRE by a formula (or set of formulae). If the property expressed by those formulae is definable in a logic that has only forward or only backward modalities a bidirectional analysis is not strictly required for PRE. Moreover, the nesting depth in formulae used to express such properties provides insight into how many iterations of forward and backward analysis are necessary for PRE.

There are two challenges to realising this solution. The first is to formalise PRE in a logic. The second is to prove what is expressible in such a logic. The standard technique for doing such proofs is to use relations such as bisimulation or simulation. The obstacle to using such techniques to reason about PRE is that they are defined over transition systems, while the analyses in PRE operate over lattices.

This dissertation contains the first step towards solving the problem above. Chapter 5 introduces subsumption and bisubsumption, which generalise bisimulation and simulation to non-Boolean lattices. Subsumption and bisubsumption can be used to reason about which properties can be computed by the analyses implemented in compilers.

Boolean Satisfiability Solvers

A third motivation for this dissertation comes from algorithmic deduction procedures. The performance of solvers to determine satisfiability of a formula in propositional logic or a quantifier-free first-order theory has improved dramatically over the last decade. Satisfiability solvers have been applied to a range of problems in program analysis and verification. At present, satisfiability solvers do not support all the operations required to build a program verifier. The operations that are lacking include image computation and generalisation from examples. Image computation is required for reachability computation and generalisation is required to discover invariants heuristically. Static analysers implement image computation using transformers and generalisation using widening operators. An unsolved problem is to develop precise and efficient tools for reasoning about programs by combining the strengths of static analysers and satisfiability solvers.

This dissertation proposes an abstract interpretation based approach to integrating satisfiability techniques with lattice-based static analysis. The solution we present is to observe that several solvers assume a fixed set of structures that they search in order to find a model. These structures can be organised to form an abstract domain and formulae can be viewed as transformers on this domain. We develop this framework in Chapter 6, where we also show that the framework naturally models the behaviour of modern Boolean satisfiability solvers.

The surprising insight of Chapter 6 is that there is a precise sense in which satisfiability solvers can be understood as lattice-based static analysers. The practical consequence of this view is that the problem of combining satisfiability solvers with static analysers reduces to the much simpler and well studied problem of combining information from different static analysers. At the time of writing, we have learnt of independent work by Cousot et al. [2011], which shows that the Nelson-Oppen method used to combine solv-

ers for logical theories can be understood as an instance of a combination technique used in abstract interpretation.

Interpolation

The final problem motivating our work also comes from the combination of program verification and satisfiability solvers. McMillan [2003] showed that by constructing interpolants from resolution proofs, one can implement a reachability algorithm for finite-state systems that only uses a SAT solver. The efficiency and precision of such a verification algorithm is contingent on the size and logical strength of the interpolants used. Hence, it is important to understand the properties of different interpolation systems.

Prior to the work reported in this dissertation, there were two algorithms for constructing interpolants and little was known about the properties of these algorithms. Chapter 7 shows that existing interpolation algorithms are elements of a lattice. Properties of this lattice translate into properties of interpolants, such as their logical strength or the variables they may contain. This work does not use the duality perspective but does apply abstract interpretation to show that existing algorithms can be derived as optimal abstractions of a lattice of interpolation algorithms.

1.2 CONTRIBUTION AND OVERVIEW

The contribution of this dissertation is in developing a framework in which both logic and abstract interpretation are first class citizens, and in applying this framework to unify and generalise existing results from automated deduction and static analysis. The results provided in this dissertation have practical consequences that have been reported elsewhere by the author and by other researchers.

The unification programme pursued in this dissertation can be broken down into two steps. The standard treatment of logics involves the notions of syntax, semantics and proof. Our first step is to move away from the traditional structures used in logical investigations to lattices with transformers, as used by algebraic logicians. This move reduces a diverse array of objects such as formulae, states, traces, and first-order structures, to the simple notion of an element of a domain. The use of transformers dispenses with distinctions between forward, backward, deductive and abductive reasoning, and allows us to view all reasoning as fixed point computation.

The second step is to simplify the lattices and fixed points that we consider by using abstract interpretation. Applying abstract interpretation has the effect of allowing us to treat logical reasoning as a fixed point approximation problem. Rather than computing a correct or incorrect answer, we can view logical reasoning as the problem of computing and refining approximations of fixed points.

This dissertation develops the perspective described above for the model theoretic problem of finding algebras for temporal logics, for the proof theoretic problem of constructing interpolants from resolution proofs, and for the algorithmic problem of determining whether a formula in propositional logic is satisfiable. Our contributions are described in detail below.

CHAPTER 3: STRUCTURES introduces *trace algebras*, a new algebraic semantics for temporal logics with linear and branching time modalities, and relates these algebras to existing algebras and transition systems.

1. The new concepts are *conjunctive transition systems* (Definition 3.47), a two-sorted extension of transition systems, and *trace algebras*, an algebraic model for temporal logic that supports future and past modalities, as well as existential path quantification.

2. The main result is a representation theorem (Theorem 3.68) which shows that trace algebras correspond precisely to a set of traces defined by a transition system.

3. A series of additional results maps the landscape of structures based on transition systems and algebras. We show that algebras defined over Boolean, distributive and perfect lattices form a hierarchy of abstractions of trace algebras (Theorems 3.23, 3.54, and 3.71,). We also consolidate disparate results in the literature to show that algebras over perfect lattices have representations as conjunctive transition systems (Theorems 3.32, 3.31 3.45, 3.44, 3.53, and 3.52).

CHAPTER 4: LANGUAGES introduces *abstraction of syntax*, a framework for combining abstract interpretation with the syntax of formal grammars. The chapter shows how semantic overapproximations can be derived from syntactic underapproximations.

1. The conceptual contribution is *meta-syntax*, a system based on BNF for specifying formal languages. We specify the language of a meta-syntax grammar in terms of domains and transformers in order to open the door to abstract interpretation.

2. The main result is to show that syntactic underapproximations that define languages closed under infinitary conjunction define semantic overapproximations (Theorem 4.16).

3. The chapter contains two case studies. Section 4.3 applies abstraction of syntax to derive numeric abstractions used in practice from a variant of Presburger arithmetic. Section 4.4 applies abstraction of syntax to derive several temporal and modal logics used in practice from a very rich temporal logic.

CHAPTER 5: PROPERTIES studies the problem of determining when two algebras defined over lattices satisfy the same properties in a propositional logic with modalities. The chapter introduces several notions of equivalence between algebras and shows that these notions strictly generalise simulation and bisimulation.

1. The main conceptual contributions are subsumption and bisubsumption (Definitions 5.3 and 5.18), which are generalisations of simulation and bisimulation. We also introduce stratified subsumption and bisubsumption (Definitions 5.8 and 5.21), which are finitary analogues of subsumption and bisubsumption.

2. Further conceptual contributions are *ordered* and *two-sorted simulation*, which are generalisations of simulation to monotone and conjunctive transition systems, respectively.

3. Our first main result is that subsumption characterises property preservation in logics closed under infinitary conjunction (Theorem 5.5) and that bisubsumption characterises property preservation in logics closed under infinitary conjunction and negation (Theorem 5.20). We

also characterise property preservation in finitary logics using stratified subsumption and bisubsumption (Theorems 5.11 and 5.22).

4. Our second main result is a representation theorem showing that certain subsumptions and bisubsumptions generate simulations, ordered simulations and two-sorted simulations (Theorems 5.35, 5.39, and 5.43).

CHAPTER 6: SOLVERS applies abstract interpretation to analyse two algorithmic components of a modern satisfiability solver. The chapter shows that the main data structure in modern solvers represents elements of an abstract domain, and that Boolean Constraint Propagation (BCP) and clause learning can be understood as transformer application.

1. The conceptual contribution is the domain of assignments and four transformers introduced in Section 6.2, which allow for satisfiability to be formulated in terms of fixed points and fixed point approximation.

2. The chapter shows that satisfiability has several different fixed point characterisations (Theorem 6.2), and these fixed points can be understood as the concrete semantics of satisfiability procedures.

3. One result is that BCP is a greatest fixed point computation in a well known abstract domain (Theorem 6.16).

4. A second result is that clause learning can be viewed as a technique to prune the search space by deriving sound transformers from unsound fixed point computation (Lemma 6.17 and Theorem 6.18).

CHAPTER 7: INTERPOLANTS applies the abstract interpretation perspective to the problem of computing interpolants from propositional resolution proofs. We derive a family of new interpolant constructions and show that this family has a rich, lattice-based structure.

1. The conceptual contribution is to use colouring functions to generate interpolant constructions (Definition 7.14).

2. We give a correctness proof for our family of interpolant constructions (Theorem 7.16) and show that this family strictly generalises existing constructions (Section 7.3).

3. We show that the new interpolant constructions have a lattice structure and that they can be ordered by the strength or number of variables in interpolants (Theorems 7.30 and 7.5).

This dissertation is self contained. All the required mathematical background is summarised in Chapter 2. Chapter 3 contains an extensive, illustrated tutorial on discrete duality theory. This material is included as a reference for researchers in model checking and static analysis because the duality literature uses different terminology and lacks examples. The material in Chapter 5 is based on material presented in Chapters 3 and 4. All other chapters can be read independently.

2

PRELIMINARIES

> Never in the history of mathematics has a mathematical theory been the object of such vociferous vituperation as lattice theory. Dedekind, Jónsson, Kurosh, Malcev, Ore, von Neumann, Tarski, and most prominently Garrett Birkhoff have contributed a new vision of mathematics, a vision that has been cursed by a conjunction of misunderstandings, resentment, and raw prejudice.
>
> ...
>
> It is a miracle that families of sets closed under unions and intersections can be characterized solely by the distributive law and by some simple identities. Jaded as we are, we tend to take Birkhoff's discovery for granted and to forget that it was a fundamental step forward in mathematics.
>
> ...
>
> It is heartening to watch every nook and cranny of lattice theory coming back to the fore after a long period of neglect. One recent instance: MacNeille, a student of Garrett's, developed a theory of completion by cuts of partially ordered sets, analogous to Dedekind's construction of the real numbers. ... These developments, and several others that I have not mentioned, are a belated validation of Garrett Birkhoff's vision, which we learned in three editions of his *Lattice Theory*.
>
> – Gian-Carlo Rota, *The Many Lives of Lattice Theory*, 1997

The mathematical background required to read this dissertation is introduced in this chapter. The chapter contains no new material. Sections 2.1 to 2.3 summarise background on lattice theory. The representation theorem for Boolean algebras with operators is reviewed in sections 2.4 and 2.5. The results in those two sections are used heavily in Chapters 3 and 4. Abstract interpretation is covered in Section 2.6. The chapter concludes with a historical survey and a personal perspective on abstract interpretation.

CONVENTIONS The word 'if' and the symbol $\triangleq$ are used for definitions. The phrase 'if and only if' and its alternatives 'exactly if' and 'exactly when' are used only as logical connectives. The symbol $\dashv$ marks the end of a proof and $\lrcorner$ marks the end of an example. Superscripts, subscripts and other disambiguation symbols are *always dropped* when no ambiguity arises. Parentheses and braces are omitted when possible. Expressions involving singleton sets, such as $f(\{s\})$, are simplified to $f(s)$. In illustrations, sets are depicted without braces to reduce clutter. For instance, a set $\{1, 2, 3\}$ will be written as $1, 2, 3$ in an illustration.

2.1 SETS AND FUNCTIONS

This section covers basic notions from set theory. The set of truth values is $\mathbb{B} \triangleq \{\mathsf{true}, \mathsf{false}\}$, the set of natural numbers is $\mathbb{N}$ and the set of integers is $\mathbb{Z}$. The class of ordinals is $\mathbb{O}$, with 0 being the first ordinal and ω being the first limit ordinal. Ordinals that are strictly less than ω are finite and all other ordinals are infinite. The set $\mathbb{N} \cup \{\infty\}$ is denoted as $\mathbb{N}_\infty$ and the set $\mathbb{Z} \cup \{-\infty, \infty\}$ is denoted as $\mathbb{Z}_\infty$.

The subset order is denoted as $\subseteq$ and the strict subset order is denoted as $\subset$. Consider two sets A and C. The *powerset* of A is $\mathscr{P}(A)$, the *Cartesian product* of A and C is $A \times C$ and the product of A with itself n-times is A^n. The *inverse* of a binary relation R is R^{-1}. The *image* of a set $X \subseteq A$ with respect to a relation $R \subseteq A \times C$ is the set

$$R(X) \triangleq \{y \in C \mid \text{there exists } x \in X \text{ such that } (x, y) \in R\}$$

of elements of C that are related to an element of A by R. The *preimage* of $X \subseteq C$ with respect to R is the image $R^{-1}(X)$. The composition of the two relations $R \subseteq A \times B$ and $S \subseteq B \times C$ is the relation $R \circ S \subseteq A \times C$

$$R \circ S \triangleq \{(a, c) \mid (a, b) \in R \text{ and } (b, c) \in S \text{ for some } b \in B\}$$

of pairs in A and C that are related to a common element in B.

Consider a relation $R \subseteq A \times A$. R is *reflexive* if (a, a) is in R for every a in A. R is *symmetric* if (b, a) is in R whenever (a, b) is in R. R is *anti-symmetric* if whenever a and b are distinct and (a, b) is in R, the pair (b, a) is not in R. R is *transitive* if whenever two pairs (a, b) and (b, c) are in R, the pair (a, c) is also in R.

EQUIVALENCE RELATIONS An *equivalence relation* is a reflexive, symmetric and transitive relation. The *equivalence class* of an element x in an equivalence relation R, denoted $[x]_R$, is the image of $\{x\}$ under R. The *representative* of an equivalence class $[x]_R$ is a unique element in $[x]_R$, denoted $\langle x \rangle_R$. The *quotient* A/R of A with respect to R is the set of equivalence classes of R.

Equivalence relations can be viewed as partitions. A *partition* of a set A is a collection of disjoint, non-empty subsets of A whose union is A. Formally, a partition is a function $P : A \to \mathscr{P}(A)$. The *block* of the partition containing the element x is $P(x)$, often denoted $[x]_P$. The fundamental theorem of equivalence relations states that every equivalence relation defines a partition and every partition defines an equivalence relation.

FUNCTIONS The set of functions with *domain* A and *codomain* B is $A \to B$. The *identity function* $id : A \to A$ maps every element of A to itself. Let $f : A \to B$ be a function. The function f is treated as a set of mappings

$\{a \mapsto f(a) \mid a \in A\}$ when convenient. The braces are often dropped to reduce clutter. Let $X \subseteq A$ be a set. The *image* of X under f, written $f(X)$, is the set

$$\{f(x) \in B \mid x \in X\}$$

of elements of B. The *domain restriction* of f to X, written $f\restriction_X$, is the function $\{a \mapsto f(a) \mid a \in X\}$. The *range restriction* of f to $Y \subseteq B$, written $f\restriction_Y$, is the function $\{a \mapsto f(a) \mid f(a) \in Y\}$. The *substitution* $f[a \mapsto c]$ is the function $g : A \to B$ that maps a to c and x distinct from a to $f(x)$. The composition of $f : A \to B$ with $g : B \to C$ is the function

$$f \circ g \mathrel{\hat{=}} \{a \mapsto g(f(a)) \mid a \in A\}$$

in $A \to C$. The notation for composition differs from the standard mathematical convention but is similar to the notation for composing relations. An *n-ary operator on A* is a function in $A^n \to A$ and n is the *arity* of f. The identity function $id : A \to A$ maps every element to itself. A 0-arity function, also called a nullary function, is treated as an element of A.

SEQUENCES Sequences are used in this dissertation primarily to simplify notation. The notation for sequences used here is due to Jónsson and Tarski [1952]. Let A be a set and m be a natural number. An *m-termed A-sequence* is a function $\bar{s} : [0, m-1] \to A$. The *origin* of the sequence is $\bar{s}(0)$. An element $\bar{s}(i)$ of the sequence is written s_i and the sequence is written as $s_0, s_1, \ldots$ when convenient. An *infinite sequence* maps $\mathbb{N}$ to A and a *two-way sequence* maps $\mathbb{Z}$ to A. The length of a sequence is written as $len(\bar{s})$. The length of an m-termed sequence is m, of an infinite sequence is ω, and of a two-way sequence is (ω, ω).

The application $f(s_0, \ldots, s_{n-1})$ of a function $f : A^n \to C$ is written $f(\bar{s})$ with the implicit qualifier that $\bar{s}$ is an n-termed A-sequence. Given a function $g : A \to C$, a sequence of substitutions $g[a_0 \mapsto c_0][a_1 \mapsto c_1] \cdots$ where the elements a_i are pair-wise distinct, and the sequences $\bar{a}$ and $\bar{c}$ are of equal length, is denoted $g[\bar{a} \Mapsto \bar{c}]$. The *pointwise application* of a function $g : A \to C$ to an A-sequence $\bar{a}$ is the C-sequence $g\langle\bar{a}\rangle \mathrel{\hat{=}} g(a_0), g(a_1), \ldots$. Let $\bar{g}$ be a finite or one-way infinite sequence of functions such that each g_i in $\bar{g}$ has finite arity n_i. Let $\bar{x}$ be a sequence that is one-way infinite if $\bar{g}$ is one-way infinite, or has length $n_0 + \cdots + n_{len(\bar{g})-1}$ if $\bar{g}$ is finite. The sequence of elements $g_0(x_0, \ldots, x_{n_1-1}), g_1(x_{n_1}, \ldots x_{n_1+n_2-1}) \ldots$ is written $\bar{g}\langle\bar{x}\rangle$.

2.2 LATTICES

This section contains a review of lattice theory. Consult the textbooks by Davey and Priestley [1990] or Grätzer [2011] for details.

A *preorder* is a reflexive and transitive relation. A *partial order* is an antisymmetric preorder. Let $\sqsubseteq$ be a partial order. The *strict partial order* corresponding to $\sqsubseteq$ is the relation $\sqsubset$ satisfying

$x \sqsubset y$ exactly if $x \sqsubseteq y$ and x is not equal to y.

If $x \sqsubseteq y$ in a preorder, we say that x is less than y, or equivalently that y is greater than x. The phrases 'strictly less than' and 'strictly greater than' are used for strict partial orders. A *poset* $(L, \sqsubseteq)$ is a set equipped with a partial order. An element y in L *covers* x if y is strictly greater than x and every z satisfying $x \sqsubseteq z \sqsubseteq y$ is equal to x or to y.

A *lattice* $(L, \sqsubseteq, \sqcap, \sqcup)$ is a poset in which every pair of elements x and y has a greatest lower bound $x \sqcap y$, called the *meet*, and a least upper bound $x \sqcup y$, called the *join*. The meet and join operations extend to finite subsets of elements $S \subseteq L$ and are written $\sqcap S$ and $\sqcup S$. A lattice as above is written $(L, \sqsubseteq)$ or L as per convenience.

Example 2.1. A standard lattice is the powerset lattice $(\mathscr{P}(A), \subseteq, \cap, \cup)$ ordered by set inclusion. The lattice of truth values is $(\mathbb{B}, \Rightarrow, \wedge, \vee)$ with implication, conjunction and disjunction as the order, meet and join, respectively. The lattice of natural numbers is $(\mathbb{N}, \leq, \mathit{min}, \mathit{max})$ with the meet and join being the minimum and maximum functions. Note that infinite subsets of $\mathbb{N}$ do not have an upper bound in $\mathbb{N}$. ⌟

PROPERTIES OF LATTICES A lattice is *bounded* if it has a least element, called *bottom* and denoted $\bot$, and has a greatest element, called *top* and denoted $\top$. A lattice L is *complete* if every subset $S \subseteq L$ has a meet, denoted $\bigsqcap S$, and a join, denoted $\bigsqcup S$. Every complete lattice is bounded.

The *distributive law* is the identity

$$x \sqcap (y \sqcup z) = (x \sqcap y) \sqcup (x \sqcap z)$$

which is provably equivalent to the identity obtained by interchanging meets and joins. A lattice is *distributive* if every x, y and z satisfy the distributive law. A *complement* of an element x in a bounded lattice is an element y for which $x \sqcap y = \bot$ and $x \sqcup y = \top$. Complements may not exist and when they do, may not be unique. A unique complement, if it exists, is denoted $\neg x$. A lattice is *complemented* if every element has a complement. Complements in a distributive lattice are unique. A *Boolean lattice* is a complemented, distributive lattice. *Boolean lattices* are usually called *Boolean algebras*.

Example 2.2. The powerset lattice $(\mathscr{P}(A), \subseteq, \cap, \cup)$ over a set A is a complete Boolean lattice. The lattice $(\mathbb{N}, \leq, \mathit{min}, \mathit{max})$ is distributive but is not complete and is not complemented. ⌟

FUNCTIONS ON LATTICES Consider a lattice $(L, \sqsubseteq, \sqcap, \sqcup)$ and a function $f : L \to L$. The variable x is universally quantified in the definitions that follow. The function f is *extensive* if $x \sqsubseteq f(x)$ and is *reductive* if $f(x) \sqsubseteq x$. The function f is *idempotent* if $f(f(x)) = f(x)$ and is an *involution* if $f(f(x)) = x$.

The complement operator in a Boolean lattice is an involution. Consider a lattice $(M, \preccurlyeq, \curlywedge, \curlyvee)$ and a function $g : L \to M$. A *monotone function* g satisfies the condition

for all x and y in L, $x \sqsubseteq y$ implies $g(x) \preccurlyeq g(y)$

and an *order embedding* g satisfies the condition

for all x and y in L, $x \sqsubseteq y$ if and only if $g(x) \preccurlyeq g(y)$

and an *order isomorphism* g is a surjective order embedding. The lattices L and M are order isomorphic, denoted $L \cong M$, if there is an order isomorphism between them. The function g is *additive* if

every x and y satisfy the equality $g(x \sqcup y) = g(x) \curlyvee g(y)$

and is *completely additive* if

every set $S \subseteq L$ satisfies the equality $g(\bigsqcup S) = \bigcurlyvee g(S)$.

The definitions of *multiplicative* and *completely multiplicative* functions are obtained by replacing joins with meets. On a bounded lattice, a function g

is *bottom-strict* if $g(\bot) = \bot$ and is *top-strict* if $g(\top) = \top$. A function is *strict additive* if it is bottom-strict and completely additive, and is *strict multiplicative* if it is top-strict and completely multiplicative.

LATTICE EXTENSIONS The terms extension and lifting are used interchangeably for constructions that enrich the order-theoretic properties of a mathematical structure. Consider a set Q and a lattice L. *Pointwise extension* lifts operators and relations on L to operators and relations on $Q \to L$ as follows. The *pointwise order* between functions in $Q \to L$, denoted $f \mathbin{\dot\sqsubseteq} g$, holds if

for all x in Q, the order $f(x) \sqsubseteq g(x)$ holds.

The *pointwise meet* $f \mathbin{\dot\sqcap} g$ is the function that maps x in Q to $f(x) \sqcap g(x)$. The *pointwise join* is similarly defined. The set of functions from a set Q to a lattice L forms a lattice $(Q \to L, \mathbin{\dot\sqsubseteq}, \mathbin{\dot\sqcap}, \mathbin{\dot\sqcup})$ under pointwise extension. If L is a complete lattice, so is $Q \to L$. When no ambiguity arises, the dots are omitted and the same symbol is used for an operator or relation and its pointwise extension.

Powerset extension allows us to move from functions between sets to functions between lattices. The *powerset extension* of a function $f : A \to B$, is the function $F : \mathscr{P}(A) \to \mathscr{P}(B)$

$$F(X) \mathrel{\hat=} \{f(a) \in B \mid a \text{ is in } X\}$$

that maps subsets of A to their image under f. Functions in $\mathscr{P}(A) \to \mathscr{P}(B)$ will be discussed in greater detail shortly.

FIXED POINTS Monotone, additive and multiplicative functions are an important component of fixed point theorems. A *fixed point* of a function $f : L \to L$ on a lattice L is

an element x in L satisfying the equality $f(x) = x$.

The set of fixed points of f is $\mathsf{Fix}(f)$. The least element of $\mathsf{Fix}(f)$, if it exists, is the *least fixed point* and is denoted $\mathsf{lfp}(f)$. The greatest element of $\mathsf{Fix}(f)$, if it exists, is the *greatest fixed point* and is denoted $\mathsf{gfp}(f)$. An *extremal fixed point* is a least or a greatest fixed point. The Knaster-Tarski fixed point theorem recalled next is a statement about the fixed points of monotone functions on complete lattices.

Theorem 2.3. *The extremal fixed points of a monotone function $f : L \to L$ on a complete lattice have the following characterisation.*

$$\mathsf{lfp}(f) = \bigsqcap \{x \mid f(x) \sqsubseteq x\} \quad \textit{and} \quad \mathsf{gfp}(f) = \bigsqcup \{x \mid x \sqsubseteq f(x)\}$$

Moreover, the fixed points of f form the complete lattice $(\mathsf{Fix}(f), \sqsubseteq, \sqcap, \sqcup)$.

The Kleene fixed point theorem characterises least fixed points of completely additive functions. The completeness requirement on the lattice L below can be weakened but suffices for this dissertation.

Theorem 2.4. *On a complete lattice L,*

$$\mathsf{lfp}(f) = \bigsqcup \{f^n(\bot) \mid n \in \mathbb{N}\}$$

is the least fixed point of a completely additive function f.

DUALITY A fundamental concept in order theory is order duality. The *order dual* of an order $\sqsubseteq$ is the order $\sqsupseteq$. When no ambiguity arises, we use the term 'dual' instead of 'order dual'. The dual of an order-theoretic definition is obtained by replacing relations and functions by their order duals. A definition that is equivalent to its dual is *self dual*. The dual of $\sqcap$ is $\sqcup$ and of $\top$ is $\bot$. Distributive lattices, Boolean lattices and monotone functions are self dual concepts. The *duality principle* states that if a statement is valid for all posets, the dual statement is also valid for all posets.

Complementation in Boolean lattices provides another notion of duality. The *De Morgan dual* of a function $f : L^n \to M$ between Boolean lattices L and M is the function

$$\tilde{f}(x_0, \ldots, x_{n-1}) \mathrel{\hat{=}} \neg f(\neg x_0, \ldots, \neg x_{n-1})$$

that maps an n-termed sequence of elements of L to their image under f, combined with pointwise negation. The De Morgan dual of meet is join and the complement operator is its own De Morgan dual.

2.3 GALOIS CONNECTIONS

Abstractions used in program analysis can often be formalised by pairs of functions called Galois connections. Galois connections provide an order theoretic notion of approximation. See Cousot's [2005] course notes for proofs of the theorems in this section.

GALOIS CONNECTIONS Let $(L, \sqsubseteq)$ and $(M, \preccurlyeq)$ be posets. Two functions $\alpha : L \to M$ and $\gamma : M \to L$ form a *Galois connection* if

for all $x \in L$ and $y \in M$, $\alpha(x) \preccurlyeq y$ if and only if $x \sqsubseteq \gamma(y)$.

A Galois connection as above is written $(L, \sqsubseteq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (M, \preccurlyeq)$, or as a tuple $((L, \sqsubseteq), \alpha, \gamma, (M, \preccurlyeq))$. When the orders involved are clear, a Galois connection is written as $L \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} M$ or (L, α, γ, M). The function α is called the *left adjoint* and γ is called the *right adjoint* of the Galois connection.

Theorem 2.5. *The conditions below are equivalent for functions $\alpha : L \to M$ and $\gamma : M \to L$ between posets $(L, \sqsubseteq)$ and $(M, \preccurlyeq)$.*

1. *(L, α, γ, M) form a Galois connection.*
2. *The functions α and γ are monotone, $\gamma \circ \alpha$ is extensive, and $\alpha \circ \gamma$ is reductive.*

If the least upper bound of $X \subseteq L$ exists, so does the least upper bound of the image $\alpha(X)$ and the equality $\alpha(\bigsqcup X) = \bigvee \alpha(X)$ holds. As a consequence, on complete lattices, one adjoint uniquely determines the other. If L and M are complete lattices, the left adjoint is completely additive and the right adjoint is completely multiplicative. The definition of γ assuming α exists and the converse case are below.

Theorem 2.6. *In a Galois connection $((L, \sqsubseteq), \alpha, \gamma, (M, \preccurlyeq))$ one adjoint uniquely determines the other.*

$$\gamma(y) \mathrel{\hat{=}} \bigsqcup \{x \in L \mid \alpha(x) \preccurlyeq y\} \qquad \alpha(x) \mathrel{\hat{=}} \bigwedge \{y \in M \mid x \sqsubseteq \gamma(y)\}$$

Completely additive functions are left adjoints of Galois connections. Similarly a completely multiplicative function is the right adjoint of a Galois

connection. The *dual Galois connection* to $(L, \sqsubseteq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (M, \preccurlyeq)$ is the Galois connection $(M, \succcurlyeq) \underset{\gamma}{\overset{\alpha}{\leftrightarrows}} (L, \sqsupseteq)$.

In the Galois connections in this dissertation, one poset is often a powerset lattice. Galois connections defined with respect to a powerset lattice $(\mathscr{P}(A), \subseteq)$ are called *overapproximating* because a set $X \subseteq A$ is contained in the set $\gamma(\alpha(X))$. Dually, Galois connections defined with respect to the dual powerset $(\mathscr{P}(A), \supseteq)$ with the superset order are *underapproximating* because a set $X \subseteq A$ contains the set $\gamma(\alpha(X))$.

RESTRICTIONS OF GALOIS CONNECTIONS Galois connections are useful for designing sound program analyses. Discussions of precision involve additional properties of Galois connections which will be introduced next.

Consider a Galois connection (L, α, γ, M). The set M is an abstraction of L in the sense that α may map multiple elements of L to the same element of M. If γ maps multiple elements of M to the same element of L, the set M contains redundancy because one element has distinct, but semantically equivalent abstractions. In a Galois insertion, every element of M represents a unique element of L. A Galois connection is called a *Galois insertion*

if γ is injective, or equivalently, if α is surjective.

In a Galois injection, every element of L has a unique representation in M. A Galois connection is called a *Galois injection*

if α is injective, or equivalently, if γ is surjective.

A Galois connection is called a *Galois isomorphism*

if it is both a Galois insertion and a Galois injection.

Note that in a Galois insertion different elements of L may map to the same element of M. Similarly, in a Galois injection, different elements of M may map to the same element of L.

A Galois connection can be reduced to a Galois insertion, as shown next. The *reduction of a Galois connection* is a tuple $((L, \sqsubseteq), \alpha_{\equiv}, \gamma_{\equiv}, (M/{\equiv}, \preccurlyeq_{\equiv}))$ consisting of a poset $(M/{\equiv}, \preccurlyeq_{\equiv})$ and two functions $\alpha_{\equiv}$ and $\gamma_{\equiv}$ defined as below. Recall that $[x]_{\equiv}$ denotes the equivalence class of x in $\equiv$ and that $\langle x \rangle_{\equiv}$ denotes a representative of the equivalence class of x in $\equiv$.

$$
\begin{aligned}
\equiv &\text{ is the relation } \{(x, y) \mid \gamma(x) = \gamma(y)\} \\
M/{\equiv} &\text{ is the quotient of } M \text{ with respect to } \equiv \\
\preccurlyeq_{\equiv} &\text{ is the partial order } \{([x]_{\equiv}, [y]_{\equiv}) \mid \langle x \rangle_{\equiv} \preccurlyeq \langle y \rangle_{\equiv}\} \\
\alpha_{\equiv} &\text{ is a function mapping } x \text{ to } [\alpha(x)]_{\equiv} \\
\gamma_{\equiv}(x) &\text{ is } \gamma(\langle x \rangle_{\equiv})
\end{aligned}
$$

The properties of a Galois connection ensure that the construction above is well defined irrespective of how representatives are chosen.

Proposition 2.7. *The reduction of a Galois connection is a Galois insertion.*

CLOSURE OPERATORS Galois insertions are closely related to functions called closures. An operator on a lattice is an *upper closure* if

it is monotone, idempotent and extensive.

An operator on a lattice is a *lower closure* if

it is monotone, idempotent and reductive.

A *closure* is an upper or a lower closure.

If (L, α, γ, M) is a Galois insertion, the function $\gamma \circ \alpha : L \to L$ is an upper closure. Conversely, an upper closure $f : L \to L$ defines a Galois insertion $((L, \sqsubseteq), f, id, (f(L), \sqsubseteq))$, where $f(L)$ is the image of L under f and id is the identity function. An overapproximating Galois insertion defines an upper closure on $(\mathscr{P}(A), \subseteq)$ and an underapproximating one defines a lower closure on $(\mathscr{P}(A), \subseteq)$. Note that the subset order is used in both cases.

Closures provide a simple representation of Galois insertions. Instead of two posets and two maps one can work with one poset and a closure operator. Ward's theorem [1942] below captures important properties of closure operators.

Theorem 2.8. *The image of a complete lattice under a closure operator is a complete lattice. The set of upper closure operators on a complete lattice, ordered pointwise, is a complete lattice.*

2.4 REPRESENTATIONS OF LATTICES AS SETS

A representation theorem states that objects in one family of structures can be constructed from another family of structures. For example, the fundamental theorem of arithmetic asserts that natural numbers greater than 1 have representations in terms of prime numbers. The theorems reviewed next show that certain lattices have representations as families of sets. Perfect Boolean lattices have representations as sets closed under union, intersection and complement. Perfect distributive lattices have representations as sets closed under union and intersection. Perfect lattices have representations as intersection closed collections of sets. The term *perfect*, as used here, exists in the literature (see [Dunn et al. 2005]), but is not standard (it does not appear in the textbooks mentioned earlier).

IRREDUCIBLE ELEMENTS An element of a lattice is irreducible if it cannot be derived from other elements using meets and joins. The representation theorems we consider decompose a lattice into sets of irreducible elements.

Example 2.9. Consider a powerset lattice $(\mathscr{P}(A), \subseteq, \cap, \cup)$. Every set $X \subseteq A$ is the union of singleton sets. Singletons are irreducible elements that can be used to reconstruct the lattice using unions. Every set X is also the intersection of sets of the form $X \setminus \{a\}$; that is, complements of singleton sets. Complements of singletons are irreducibles that can be used to reconstruct the lattice using intersection.

Consider the lattice $(\mathbb{N}_\infty, \leq, min, max)$ of natural numbers extended with ω. This lattice is not complemented. All elements strictly greater than 0 are called join-irreducibles because they cannot be obtained from other elements by taking joins. These elements are also meet-irreducibles because they cannot be obtained by meets. ⌟

Fix a lattice L for this section. An *atom* in a bounded lattice is an element that covers $\bot$. A *coatom* is an element that is covered by $\top$. The sets of atoms and coatoms of L are $Atom(L)$ and $CoAtom(L)$, respectively. The set of atoms below an element x is

$$Atom(x) \triangleq \{y \in Atom(L) \mid y \sqsubseteq x\}$$

and the set of coatoms above an element x is

$$CoAtom(x) \triangleq \{y \in CoAtom(L) \mid x \sqsubseteq y\}$$

A bounded lattice is *atomic* if every element strictly greater than bottom is an atom or is greater than an atom. For example, the lattice $(\mathbb{N}_\infty, \geq, max, min)$, with the greater-than order, ω as the least element and 0 as the greatest, is bounded but is not atomic.

An element $x \neq \bot$ is *completely join-irreducible* if

for every subset S of L, $x = \bigsqcup S$ implies that x is in S.

A *join-irreducible* is an element satisfying a modification of the above definition, where arbitrary joins are replaced by finite joins. As no ambiguity arises in this dissertation, we will use the term join-irreducible in place of completely join-irreducible. The set of join-irreducibles of L is denoted $Irr_\sqcup(L)$. The set of join-irreducibles below x is

$$Irr_\sqcup(x) \mathrel{\hat{=}} \{y \in Irr_\sqcup(L) \mid y \sqsubseteq x\}.$$

Meet-irreducibles are dually defined. The set of meet-irreducibles of L is denoted $Irr_\sqcap(L)$. The set of meet-irreducibles above x is denoted $Irr_\sqcap(x)$ and defined as expected.

Join-dense sets contain building blocks for deriving lattice elements by joins. A subset S of a lattice L is *join-dense* if

every x in L is equal to $\bigsqcup Q$ for some subset Q of S.

A *meet-dense* set is dually defined. A complete lattice L is *join-perfect* if

$Irr_\sqcup(L)$ is join-dense,

is *meet-perfect* if

$Irr_\sqcap(L)$ is meet-dense,

and is perfect if

$Irr_\sqcup(L)$ is join-dense and $Irr_\sqcap(L)$ is meet-dense.

In a join-perfect lattice, every element is the join of join-irreducibles. In a meet-perfect lattice, every element is the meet of meet-irreducibles. In a perfect lattice, every element is both the join of join-irreducibles and the meet of meet-irreducibles.

PERFECT BOOLEAN ALGEBRAS A *perfect Boolean lattice* is a complete, perfect Boolean algebra. Perfect Boolean lattices are atomic and are more commonly called *complete, atomic, Boolean algebras*. All finite Boolean lattices are perfect. The lattice of regular languages ordered by inclusion is Boolean but is not complete. See [Hopenwasser 1990] for an example of a complete Boolean lattice that is not perfect.

Proposition 2.10. *A perfect Boolean lattice is complete and atomic.*

The proposition holds because join-irreducibles of a Boolean lattice are atoms (Lemma 5.3 in [Davey and Priestley 1990]). Perfect Boolean lattices have representations as powerset lattices.

Theorem 2.11. *A perfect Boolean lattice L is isomorphic to the powerset lattice of atoms, $\mathscr{P}(Atom(L))$.*

The proof proceeds by showing that the map $x \mapsto Atom(x)$ is an order isomorphism from L to $\mathscr{P}(Atom(L))$. For details, consult the proof of Theorem 10.24 in [Davey and Priestley 1990].

PERFECT DISTRIBUTIVE LATTICES A *perfect distributive lattice* is a complete, perfect, distributive lattice. These lattices are more commonly called doubly algebraic, distributive lattices, or DL$^+$. The representation theorem for perfect distributive lattices is considerably more technical than that for perfect Boolean lattices.

Example 2.12. The lattice $(\mathbb{N}_\infty, \leq, max, min)$ is complete and distributive. Every element x and subset Y of elements satisfies the identity

$$min(x, max(Y)) = max(\{min(x, y) \mid y \in Y\})$$

and its dual. The identity states that meets distribute over arbitrary joins and joins distribute over arbitrary meets. This is a stronger property than the standard distributive identity because it involves arbitrary joins and meets. Every natural number n in $\mathbb{N}_\infty$ is a join-irreducible because $max(S) = n$ entails that S contains n. The element ω is join-irreducible because it is not equal to $max(S)$ for any finite subset S of $\mathbb{N}$ but is not completely join-irreducible because it equals $max(\mathbb{N})$. ⌟

Perfect distributive lattices satisfy several distributive laws. A lattice L satisfies the *infinite join-distributive* law if for every set of indices I, indexed set $\{x_i \mid i \in I\}$ and element y, the equality

$$y \sqcap (\bigsqcup \{x_i \mid i \in I\}) = \bigsqcup \{y \sqcap x_i \mid i \in I\})$$

is satisfied. The *infinite meet-distributive* law is the dual condition. In the law above, the meet of y with the join of an indexed set is equal to the join of the elements $y \sqcap x_i$. The generalisation of this law to arbitrary meets and joins requires two sets of indices I and J and the functions $I \to J$. A lattice is *completely distributive* if the identity

$$\bigsqcap \left\{ \bigsqcup \{x_{i,j} \mid j \in J\} \mid i \in I \right\} = \bigsqcup \left\{ \bigsqcap \left\{ x_{i,f(i)} \mid i \in I \right\} \mid f \in I \to J \right\}$$

is satisfied by every doubly indexed set $\{x_{i,j} \mid i \in I, j \in J\}$.

Proposition 2.13. *A perfect distributive lattice is completely distributive.*

Theorem 2.11 provides a representation of perfect Boolean lattices via atoms and the powerset operator. In the representation of perfect distributive lattices, atoms are replaced by join-irreducibles and powersets are replaced by special posets called downsets.

DOWNSETS Let $(M, \preccurlyeq)$ be a poset. A subset S of M is *downwards closed* if

for every x in S and y in M, $y \preccurlyeq x$ implies that y is also in S.

A downwards closed set is also called a *downset*. The smallest downset containing a set S is denoted $S{\downarrow}$. A *principal downset*, denoted $x{\downarrow}$, is the downset of a singleton. The set of downsets of M is denoted $\mathscr{D}(M, \preccurlyeq)$, usually abbreviated to $\mathscr{D}(M)$. The set of downsets of M with the subset order, written $(\mathscr{D}(M), \subseteq, \cap, \cup)$, is a complete lattice called the *downset lattice*.

The dual of a downwards closed set is an *upwards closed* set, also called an *up-set*. The smallest up-set containing S is written $S{\uparrow}$, and a *principal up-set* is denoted $x{\uparrow}$. The set of up-sets of a poset $(M, \preccurlyeq)$ is denoted $\mathscr{U}(M)$, and $(\mathscr{U}(M), \subseteq, \cap, \cup)$ is the *up-set lattice*. The up-set lattice is complete.

Downsets and up-sets are related to powersets as follows. Consider the poset $(M, =)$ where $=$ is the identity relation. The posets of downsets $(\mathscr{D}(M), \subseteq)$, up-sets $(\mathscr{U}(M), \subseteq)$, are both isomorphic to the powerset lattice

$\mathscr{P}(M)$. Thus, powersets are specific instances of downsets and up-sets. Example 2.14 presents a downset lattice that is not a Boolean lattice. It follows that downsets and up-sets strictly generalise powersets.

Example 2.14. In the lattice $(\mathbb{N}_\infty, \leq, max, min)$ the poset of join-irreducibles is $(\mathbb{N}, \leq)$. The downset lattice $\mathscr{D}(\mathbb{N}, \leq)$ contains one set for each element of $\mathbb{N}$ and exactly one infinite downset, namely $\mathbb{N}$. The downset lattice $\mathscr{D}(\mathbb{N}_\infty, \leq)$ contains two infinite downsets, $\mathbb{N}$ and $\mathbb{N}_\infty$. Observe that $(\mathscr{D}(\mathbb{N}), \subseteq)$ is isomorphic to $(\mathbb{N}_\infty, \leq)$ but $(\mathscr{D}(\mathbb{N}_\infty), \subseteq)$ is not. Neither downset lattice is a Boolean lattice because the only element with a complement is $\{0\}$. ⌟

Proposition 2.15. *The lattice $(\mathscr{D}(M), \subseteq, \cap, \cup)$ of downsets of a poset $(M, \preccurlyeq)$ is a perfect distributive lattice.*

The set of join-irreducibles of a lattice $(L, \sqsubseteq)$ with the lattice order forms the poset $(Irr_\sqcup(L), \sqsubseteq)$. Perfect distributive lattices have representations as downsets of join-irreducibles.

Theorem 2.16. *A perfect distributive lattice $(L, \sqsubseteq)$ is isomorphic to the lattice $\mathscr{D}(Irr_\sqcup(L), \sqsubseteq)$ of downsets of join-irreducibles.*

The proof for this theorem shows that the map $x \mapsto Irr_\sqcup(x)$ is an order isomorphism from the lattice L to $\mathscr{D}(Irr_\sqcup(L))$. See the proof of Theorem 10.29 in [Davey and Priestley 1990] for details.

PERFECT LATTICES The complete distributivity identity is self-dual. Therefore, in a distributive lattice the poset of join-irreducibles is dual to the poset of meet-irreducibles. It follows that a perfect, distributive lattice can be represented either as downsets of join-irreducibles or as up-sets of meet-irreducibles. The posets of join- and meet-irreducibles are not order isomorphic in arbitrary perfect lattices. Both types of irreducibles are required to derive set-based representations of perfect lattices. The material on representation of perfect lattices is based on Chapters 7 and 10 of [Davey and Priestley 1990] and on results in [Dunn et al. 2005].

Example 2.17. This example illustrates the representation of a perfect, non-distributive lattice by irreducibles. The set of integer intervals *Intv* contains pairs $[a, b]$ such that $a \leq b$ for a and b in $\mathbb{Z} \cup \{-\infty, \infty\}$. The order on intervals is $[a, b] \sqsubseteq [c, d]$ if $c \leq min(a, c)$ and $max(b, d) \leq d$. Let $\bot$ denote the empty interval, which is the least interval. The poset $(Intv, \sqsubseteq)$ is a complete lattice.

The sets of join- and meet-irreducibles of *Intv* are shown below.

$$Irr_\sqcup(Intv) = \{[a, a] \mid a \in \mathbb{Z}\}$$
$$Irr_\sqcap(Intv) = \{[-\infty, a], [a, \infty] \mid a \in \mathbb{Z}\}$$

All elements of $Irr_\sqcup(Intv)$ cover $\bot$ and must be join-irreducible. Every finite interval $[a, b]$ is the join of $[a, a]$ and $[b, b]$. Every one-way infinite interval satisfies the equality below.

$$[a, \infty] = \bigsqcup \{[a, a], [a + 1, a + 1], [a + 2, a + 2], \cdots\}$$

It follows that $Irr_\sqcup(Intv)$ is join-dense in *Intv*. Consider the interval $[a, \infty]$ in $Irr_\sqcap(Intv)$. Every set of intervals S that satisfies $\bigsqcap S = [a, \infty]$ must contain $[a, \infty]$, so $Irr_\sqcap(Intv)$ contains completely join-irreducibles. Every one-way infinite interval is meet-irreducible and every finite interval $[a, b]$ is the meet of $[-\infty, b]$ and $[a, \infty]$ so $Irr_\sqcap(Intv)$ is meet-dense.

Observe that elements of $Irr_{\sqcup}(Intv)$ are pairwise incomparable while elements of $Irr_{\sqcap}(Intv)$ such as $[a, \infty]$ and $[a+1, \infty]$ are comparable. Consequently, there is no order isomorphism between $Irr_{\sqcup}(Intv)$ and $Irr_{\sqcap}(Intv)$. The lattice of downsets $\mathscr{D}(Irr_{\sqcup}(Intv), \sqsubseteq)$ is isomorphic to the powerset lattice $\mathscr{P}(\mathbb{Z})$ but is not isomorphic to $Intv$. The interval lattice is not isomorphic to downsets of its join-irreducibles.

The lattice $Intv$ can be represented by combining information about join- and meet-irreducibles. Let L contain the join-irreducibles (lower elements) of $Intv$ and U contain the meet-irreducibles (upper elements). Consider a function $u : \mathscr{P}(L) \to \mathscr{P}(U)$ that maps a subset of L to its upper approximations in U, and a function $\ell : \mathscr{P}(U) \to \mathscr{P}(L)$ that maps a subset of U to its lower approximations in L.

$$u \mathrel{\hat{=}} X \mapsto \{y \in U \mid x \sqsubseteq y \text{ for every } x \text{ in } X\}$$
$$\ell \mathrel{\hat{=}} Y \mapsto \{x \in L \mid x \sqsubseteq y \text{ for every } y \text{ in } Y\}$$

Every element of $u(X)$ is above every element of X. Every element of $\ell(Y)$ is below every element of Y. A set X satisfying the equality $X = \ell(u(X))$ is defined by its upper approximations and by its lower approximations. Extending this observation leads to the representation theorem for perfect lattices in which a set X is represented by $\ell(u(X))$. ⌟

DEDEKIND-MACNEILLE COMPLETION The construction in the previous example is called the Dedekind-MacNeille completion of a poset. The Dedekind-MacNeille completion can be defined in several ways. The definition that follows is based on [Davey and Priestley 1990] but the term 'twoset' is newly introduced here.

A *two-sorted poset* (twoset) $M = (L, U, \preccurlyeq)$ is

a poset $(L \cup U, \preccurlyeq)$ in which L is join-dense and U is meet-dense.

The sets L and U need not be disjoint. A twoset gives rise to two functions u and ℓ that map a set to its upper and lower approximations, respectively.

$$u : \mathscr{P}(L) \to \mathscr{P}(U) \qquad u \mathrel{\hat{=}} X \mapsto \{y \in U \mid x \preccurlyeq y \text{ for all } x \text{ in } X\}$$
$$\ell : \mathscr{P}(U) \to \mathscr{P}(L) \qquad \ell \mathrel{\hat{=}} Y \mapsto \{x \in L \mid x \preccurlyeq y \text{ for all } y \text{ in } Y\}$$

The proposition below shows that the two approximation functions form a Galois connection. Note that $(\mathscr{P}(U), \supseteq)$ uses the superset order.

Proposition 2.18. *The upper and lower powersets with the approximation functions form a Galois connection* $(\mathscr{P}(L), \subseteq) \underset{u}{\overset{\ell}{\leftrightarrows}} (\mathscr{P}(U), \supseteq)$.

A subset X of L in a twoset $(L, U, \preccurlyeq)$ is *Galois-stable*

if X satisfies the equality $X = \ell(u(X))$.

The set of Galois-stable subsets of a twoset M is denoted $\mathscr{G}(M)$. The Galois-stable subsets are closed under arbitrary intersection, though not necessarily under union. The lattice of Galois-stable subsets of an *arbitrary poset*, with L and U being the same, is the *Dedekind-MacNeille completion* of a poset. When applied to twosets this completion yields a representation of perfect lattices [Dunn et al. 2005].

Theorem 2.19. *The Galois stable subsets of a twoset M form a perfect lattice $(\mathscr{G}(M), \subseteq, \sqcap, \sqcup)$ of sets closed under intersection.*

Consult Section 7.30 in [Davey and Priestley 1990] for a proof. The next theorem shows that perfect lattices have representations as Galois-stable subsets of twosets.

Theorem 2.20. *A perfect lattice $(L, \sqsubseteq)$ is isomorphic to the lattice of Galois-stable subsets $\mathcal{G}(Irr_{\sqcup}(L), Irr_{\sqcap}(L), \sqsubseteq)$ of join- and meet-irreducibles of L.*

The proof given by Davey and Priestley [1990] shows that the map $x \mapsto \ell(u(Irr_{\sqcup}(x)))$ is an isomorphism of perfect lattices. For details, see Theorem 7.41 in [Davey and Priestley 1990] and Section 4 of [Dunn et al. 2005] (where twosets are called perfect posets).

2.5 ALGEBRAS

An algebra is a set equipped with a collection of operators. Groups, rings and lattices are examples of algebras. Universal algebra is the study of properties common to all algebras. See [Burris and Sankappanavar 2000] for an introduction to universal algebra.

Algebras are defined over signatures. A *signature* is a set of symbols *Sig* with an arity function $ar : Sig \to \mathbb{O}$. A *Sig-algebra* $\mathcal{A} = (A, O_A)$ is a set A, called the *domain*, and a collection of operators $f^{\mathcal{A}} : A^{ar(f)} \to A$, such that there is one operator for each symbol $f \in Sig$. The signature of an algebra is its *type*. For this section, fix a signature *Sig* and two *Sig*-algebras $\mathcal{A} = (A, O_A)$ and $\mathcal{B} = (B, O_B)$.

A homomorphism is a structure preserving map between algebras of the same type. A function $h : A \to B$ is a *Sig-homomorphism* if

$$h(f^{\mathcal{A}}(a)) = f^{\mathcal{B}}(h(a)) \text{ for every } f \text{ in } Sig \text{ and } a \text{ in } A.$$

A function is a *Sig-isomorphism* if it is a *Sig*-homomorphism and its inverse is a *Sig*-homomorphism. Two *Sig*-algebras are isomorphic, denoted $\mathcal{A} \cong \mathcal{B}$, if there exists a *Sig*-isomorphism between them.

CONGRUENCE A congruence is a structure-preserving equivalence relation. An equivalence relation $R \subseteq A \times A$ on the algebra $\mathcal{A}$ is a *Sig-congruence* if for every symbol $f \in Sig$ and every two $ar(f)$-termed sequences $\bar{x}$ and $\bar{y}$ satisfying that (x_i, y_i) is in R for all $i \in dom(\bar{x})$, it holds that $(f^{\mathcal{A}}(\bar{x}), f^{\mathcal{A}}(\bar{y}))$ is in R. The *quotient* of an algebra $\mathcal{A}$ with respect to a congruence R, is the algebra $\mathcal{A}/R = (A/R, O_{A/R})$ defined below.

$$A/R \triangleq \{[x]_R \mid x \in A\}$$
$$f^{\mathcal{A}/R}([x_0]_R, \ldots, [x_{n-1}]_R) \triangleq [f(x_0, \ldots, x_{n-1})]_R$$

Boolean Algebras with Operators

Boolean algebras with operators were studied by Jónsson and Tarski [1952]. A restriction of these algebras to perfect Boolean algebras will be used in this dissertation. The definitions and results that follow are from [Jónsson and Tarski 1952].

Definition 2.21. A *Boolean algebra with additive operators* (BAO) $\mathcal{A} = (A, O_A)$ is a perfect Boolean lattice A, with a set O_A of strict additive operators.

Consider two BAOs $\mathcal{A} = (A, O_A)$ and $\mathcal{C} = (C, O_C)$. Two functions $f : A \to C$ and $g : C \to A$ are *conjugate* if

for all a in A and c in C, $f(a) \sqcap c = \bot$ if and only if $a \sqcap g(c) = \bot$.

If f and g are conjugate, g is called the conjugate of f and f is called the conjugate of g. A function is *self-conjugate* if it is conjugate with itself. The *dual-conjugate* condition is obtained by replacing meets with joins and $\bot$ with $\top$ in the definition of conjugate. Lemma 2.22 shows that a strict additive function has a unique conjugate.

Lemma 2.22. *Conjugate functions are strict additive.*

Lemma 2.23. *A strict additive function has a unique conjugate.*

Theorem 2.24 provides alternate characterisations of conjugate functions. The third condition below is sometimes called the *Dedekind law*, after a similar identity in group theory.

Theorem 2.24. *The conditions below are equivalent for two functions $f : A \to C$ and $g : C \to A$ between Boolean algebras.*

1. *The functions f and g are conjugate.*
2. *For every a in A and c in C, it holds that $f(x \sqcap \neg g(y)) \sqsubseteq f(x) \sqcap \neg y$ and $g(y \sqcap \neg f(x)) \sqsubseteq g(y) \sqcap \neg x$.*
3. *The functions f and g are strict additive and for every a in A and c in C, it holds that $f(x) \sqcap y \sqsubseteq f(x \sqcap g(y))$ and $g(y) \sqcap x \sqsubseteq g(y \sqcap f(x))$.*

Lemma 2.25 below relates Galois connections between complete Boolean algebras with conjugate functions. This lemma is new and does not appear in Jónsson and Tarski's paper.

Lemma 2.25. *In a Galois connection (A, α, γ, C) between two complete Boolean algebras, α and $\tilde{\gamma}$ are conjugate, where $\tilde{\gamma}$ is $\lambda x.\neg\gamma(\neg x)$.*

Proof. Consider $a \in A$ and $c \in C$ such that $\alpha(a) \sqcap c = \bot$. It follows that $\alpha(a) \sqsubseteq \neg c$, so $a \sqsubseteq \gamma(\neg c)$ by the Galois connection, whence $a \sqcap \neg\gamma(\neg c) = \bot$, showing that α and $\tilde{\gamma}$ are conjugate. ⊣

RELATIONAL STRUCTURES Relational structures are mathematical models that use tuples of sets and relations. Finite automata, Kripke structures and first-order structures are examples of relational structures.

A *Sig-relational structure* $\mathcal{S} = (S, Rel_S)$ is a set S with a collection of relations Rel_S such that there is a relation $F^{\mathcal{S}}$ of arity $ar(f) + 1$ for each symbol $f \in Sig$. Let $\mathcal{S} = (S, Rel_S)$ and $\mathcal{T} = (T, Rel_T)$ be relational structures. Recall the notation $h\langle\bar{s}\rangle$ from Section 2.1, denoting the application of h to each element of $\bar{s}$. A function $h : S \to T$ is a *relational homomorphism* if

for every symbol f in Sig and S-termed sequence $\bar{s}$ of length $ar(f) + 1$, whenever $\bar{s}$ is in $F^{\mathcal{S}}$, it holds that $h\langle\bar{s}\rangle$ is in $F^{\mathcal{T}}$.

Relational isomorphisms are similarly defined. Let $\mathcal{S} \cong \mathcal{T}$ denote isomorphism of two relational structures.

THE JÓNSSON–TARSKI THEOREM Just as perfect Boolean lattices have representations as sets of atoms, the Jónsson–Tarski representation theorem shows that BAOs have representations as relational structures. The modern mathematical presentation of the Jónsson–Tarski theorem is as a duality of categories. The category theoretic view is not used here.

Let $\mathcal{A} = (A, O_A)$ be a BAO with atoms $Atom(A)$. A function $f^{\mathcal{A}}$ in O_A, defines the relation $rel(f)$ of arity $ar(f)+1$ over $Atom(A)$ given below. The relational structure defined by $\mathcal{A}$ is given below.

$$rel(f) \triangleq \left\{(a_0, \ldots, a_n) \in Atom(A)^{n+1} \mid a_0 \sqsubseteq f(a_1, \ldots, a_n)\right\}$$
$$rel(\mathcal{A}) \triangleq (Atom(A), \{rel(f) \mid f \in O_A\})$$

Conversely, a relational structure $\mathcal{S} = (S, Rel_S)$ defines a BAO as below. A relation F in Rel_S defines a function $g_F : S^{ar(f)} \to \mathscr{P}(S)$ and this function defines an operator $balg(F)$ of arity $ar(f)+1$ on $\mathscr{P}(S)$. The operators $balg(F)$ along with $\mathscr{P}(S)$ define the BAO $balg(\mathcal{S})$.

$$g_R \triangleq \{(s_0, \ldots, s_{n-1}) \mapsto \{s \mid (s, s_0 \ldots, s_{n-1}) \in F\}\}$$
$$balg(F) \triangleq \left\{\bar{X} \mapsto \bigcup \{g_R(s_0, \ldots, s_{n-1}) \mid s_i \in X_i \text{ and } i \in dom(\bar{X})\}\right\}$$
$$balg(\mathcal{S}) \triangleq (\mathscr{P}(S), \{balg(F) | F \in Rel_S\})$$

The two constructions show that a BAO defines a relational structure and a relational structure defines a BAO. The Jónsson–Tarski representation theorem for BAOs states that every BAO is isomorphic to the algebra generated by a relational structure.

Theorem 2.26. *Relational systems and* BAOs *are related as follows.*

1. *Every* BAO *is isomorphic to the algebra generated by its relational representation:* $\mathcal{A} \cong balg(rel(\mathcal{A}))$.
2. *Every relational system is isomorphic to the one generated by algebra:* $M \cong rel(balg(\mathcal{A}))$.

The first statement can be proved by showing that the map $x \mapsto Atom(x)$ is an isomorphism of BAOs. The second statement can be proved by showing that the map $s \mapsto \{s\}$ is an isomorphism of relational systems. The Jónsson–Tarski representation theorem provides a systematic way to translate between properties of relations and strict-additive operators. This translation has important algorithmic consequences because some operations, such as taking a cut in a graph, are natural on relations , while other operations, such as approximating a fixed point, are natural on lattices. A specific example that will be used later is the representation of a relation and its inverse by conjugate functions.

Theorem 2.27. *Two strict additive, unary operators f and g are conjugate if and only if the relation $rel(f)$ is the inverse of $rel(g)$.*

A second property connecting relations and their operators is a characterisation of functions.

Theorem 2.28. *A strict additive, unary operator f is a perfect Boolean algebra homomorphism if and only if $rel(f)$ is a function.*

2.6 ABSTRACT INTERPRETATION

Approximations are used to derive partial information regarding problems that are difficult to solve in a precise manner. When dealing with geometric or arithmetic problems, approximations are defined with respect to a metric that can be used to measure the distance between the approximate solution and the actual solution. The structures studied in logic and program analysis lack natural metrics and therefore a different notion of approximation is

required. Abstract interpretation is a theory of approximation based on ordered sets. The material in this section is based on a tutorial by Cousot and Cousot [1992a] and Cousot's course notes [2005].

The abstract interpretation approach to problem solving can be described as a three-step process. The first step is to characterise a solution to a problem as a fixed point of an operator in a lattice. This fixed point defines the *concrete semantics* of the problem. The second step is to identify a space of approximate properties and another fixed point that approximates the concrete semantics. This approximate fixed point defines the *abstract semantics* of the problem. The third step is to compute or further approximate the abstract semantics. The abstract interpretation literature contains several results that aid in the design and implementation of abstract semantics and algorithms to approximate the abstract semantics.

Complete lattices equipped with monotone functions are called *domains* in abstract interpretation. The term *domain* is also used for just the lattice. For this section, fix two lattices $(C, \sqsubseteq, \sqcap, \sqcup)$ and $(A, \preccurlyeq, \curlywedge, \curlyvee)$ and a Galois connection (C, α, γ, A) between C and A. The lattice C is called the *concrete domain* and A is called the *abstract domain*. Furthermore, fix two algebras $\mathcal{C} = (C, O_C)$ and $\mathcal{A} = (A, O_A)$ over the same signature. The operators in O_C are called *concrete transformers* and those in O_A are called *abstract transformers*.

SOUNDNESS An abstract transformer $f^{\mathcal{A}}$ is a *sound overapproximation* of a concrete transformer $f^{\mathcal{C}}$

if the pointwise constraint $\alpha \circ f^{\mathcal{C}} \preccurlyeq f^{\mathcal{A}} \circ \alpha$ is satisfied.

The properties of a Galois connection imply that a sound overapproximation also satisfies $f^{\mathcal{C}} \circ \gamma \sqsubseteq \gamma \circ f^{\mathcal{A}}$. A foundational result of abstract interpretation is that sound overapproximations of transformers imply overapproximations of fixed points. In the context of program analysis and verification, sound, overapproximate, abstract transformers are often called sound transformers because the other properties are understood.

Theorem 2.29. *If $f^{\mathcal{C}} : C \to C$ and $f^{\mathcal{A}} : A \to A$ are monotone and $f^{\mathcal{A}}$ is a sound overapproximation of $f^{\mathcal{C}}$, it holds that $\alpha(\mathsf{lfp}(f^{\mathcal{C}})) \preccurlyeq \mathsf{lfp}(f^{\mathcal{A}})$.*

The Galois connection formulation has several implications for the design of program analyses. One implication is different notions of optimality. The *best abstract transformer* is a function

$f^{\mathcal{A}} : A \to A$ defined as $\alpha \circ f^{\mathcal{C}} \circ \gamma$.

The adjective 'best' signifies that there is no sound overapproximation of $f^{\mathcal{C}}$ that is less than $f^{\mathcal{A}}$ in the pointwise order.

Proposition 2.30. *If $f^{\mathcal{A}} : A \to A$ is a sound overapproximation of a monotone transformer $f^{\mathcal{C}} : C \to C$, it holds that $\alpha \circ f^{\mathcal{C}} \circ \gamma \sqsubseteq f^{\mathcal{A}}$.*

By monotonicity, fixed points obtained using best transformers are at least as precise as fixed points defined by arbitrary sound transformers. If $\mathcal{C} = (C, O_C)$ is an algebra with monotone transformers and A is a lattice in a Galois connection with C, the *best abstraction* of $\mathcal{C}$, given A, is the algebra $\mathcal{A} = (A, O_A)$ consisting of A and best abstract transformers for each transformer in O_C.

COMPLETENESS Let $f^{\mathcal{C}}$ be a concrete transformer and $f^{\mathcal{A}}$ be an abstract transformer. The abstract transformer is *α-complete*

if the pointwise constraint $f^{\mathcal{A}} \circ \alpha \preccurlyeq \alpha \circ f^{\mathcal{C}}$ is satisfied,

and is *γ-complete*

if the pointwise constraint $\gamma \circ f^{\mathcal{A}} \sqsubseteq f^{\mathcal{C}} \circ \gamma$ is satisfied.

Unlike soundness the two notions of completeness are not equivalent. If a transformer is both sound and complete, the inequalities above collapse to equalities. A sound transformer that is α-complete satisfies the identity $\alpha \circ f^{\mathcal{C}} = f^{\mathcal{A}} \circ \alpha$, and a sound transformer that is γ-complete satisfies the identity $f^{\mathcal{C}} \circ \gamma = \gamma \circ f^{\mathcal{A}}$.

CLOSURE OPERATORS The relationship between Galois connections and closure operators provides several insights about the space of abstractions. The reduction of a Galois connection yields a reduction of abstract domains that eliminates distinct abstract elements with the same concretisation. The reduction of a Galois connection being a Galois insertion defines a closure operator. The lattice of abstract domains, following reduction, is isomorphic to the lattice of closure operators.

Theorem 2.31. *The set of abstract domains in a Galois insertion with a fixed concrete domain forms a complete lattice.*

In particular, a reduced abstract domain is the image of the concrete domain under an upper closure operator. The notions of sound and best transformers can be rephrased in terms of closure operators. Let ρ be an upper closure on C. The abstract domain defined by ρ is $\rho(C)$. The best abstract transformer for $f^{\mathcal{C}}$ is $\rho \circ f^{\mathcal{C}} \circ \rho$.

2.7 BIBLIOGRAPHIC NOTES

LATTICES AND DUALITY In the early nineteenth century, Boole's mathematical analysis of logic [1847; 1854] led to the concept of Boolean algebras. Dedekind independently discovered complete lattices in his work on algebraic numbers [1897]. The idea that a lattice was an object worthy of independent study was articulated by Pierce, who also discovered the distributive law. Schröder showed that the distributive law was equivalent to its dual. Birkhoff, in a series of papers, showed that lattice theory provided a framework for unifying several disparate developments in mathematics. Much of his work is summarised in his book [Birkhoff 1967].

The study of distributive lattices has led to a rich framework of results connecting lattice theory, algebra, logic and topology. Rota's comment on the distributive law (from the beginning of this chapter) is worth repeating.

> *It is a miracle that families of sets closed under unions and intersections can be characterised solely by the distributive law and by some simple identities.*

The intimate connection between families of sets and lattices is one of the many insights that representation theorems bring to the landscape of mathematics. Duality theorems in category theory significantly extend the scope and insight provided by representation theorems. Lattice-theoretic representation theorems are used in this dissertation in place of their more general and aesthetically simpler category theoretic counterparts because the mathematical foundation of program analysis is lattice-theory. Consult Johnstone's book [1986] for a masterful introduction to Stone duality from a category-theoretic perspective.

The representation of perfect Boolean lattices by powersets is due to Tarski [1935] (who also credits Lindenbaum). Birkhoff [1967] proved a representation theorem for finite distributive lattices, relating them to finite posets. The generalisation of Birkhoff's theorem to arbitrary distributive lattices was achieved by Stone [1936] who proved the representation theorem for arbitrary Boolean algebras. The family of results generalising his theorem is called Stone duality in his honour. Stone's proof used methods from topology. A restriction of Stone duality to the translation between set-theoretic and lattice-theoretic objects is called discrete duality. Discrete duality for completely distributive lattices was first studied by Raney [1952].

Jónsson and Tarski [1952] gave a representation theorem for Boolean algebras with operators. Their paper introduced a technique for saturating an algebraic structure to obtain one that has a discrete representation. This technique is now called *canonical extension* and has been extended to obtain discrete representations for several families of lattices with operators [Gehrke and Jónsson 1994; Dunn et al. 2005]. The representation of non-distributive lattices given here was taken from Davey and Priestley [1990] and heavily influenced by the presentation in [Dunn et al. 2005].

ABSTRACT INTERPRETATION Lattices were first applied to reason about programs by Scott [1969] in work that eventually led to the development of domain theory [Abramsky and Jung 1994] and denotational semantics. While the work of semanticists was largely theoretical, Kildall [1973] demonstrated the practical utility of the lattice-theoretic perspective by unifying several different compiler optimisations. Kam and Ullman [1977] extended Kildall's approach by showing that analysis with distributive transformers produced optimal results and that in the non-distributive case the optimal solution was not always computable. Initial work on lattice-based program analysis was restricted to finite-height lattices, required separate soundness proofs for each analysis, and lacked a general notion of optimality. New ideas were required to significantly expand the scope of this framework.

There is a tongue-in-cheek expression that credit in science goes to the last person to make a discovery. In the history of program analysis, the last papers discovering lattice-based analysis were decidedly written by Cousot and Cousot [1977; 1979]. An early technical report [1975] shows that their work was independent of that of Kildall or of Kam and Ullman. The theory of abstract interpretation brought several fresh conceptual and mathematical perspectives to program analysis. Three decades later, abstract interpretation is widely regarded as the basis of program analysis.

Prior to abstract interpretation, soundness proofs were conducted in a mathematical setting distinct from the setting in which a program analysis was designed. In abstract interpretation, soundness was formulated using the lattice-theoretic notion of Galois connections. Using Galois connections decouples the design of a sound analysis from the soundness proof itself, achieving a valuable separation of concerns. Moreover, the design of the analysis and the proof of its soundness reside in the same mathematical universe, a coincidence which has significant consequences. For instance, an analysis can be designed using objects that appear in the soundness proof, such as abstraction and concretisation functions. It is now routine to take abstraction and concretisation functions for granted, but the introduction of this paradigm shift is only one of the many conceptual contributions of abstract interpretation.

Another consequence of the Galois connection formulation is the notion of a best abstract transformer, which describes the maximal precision that an abstraction can deliver [Cousot and Cousot 1979]. The same Galois connection that is used to prove soundness can also be used to prove optimality. Though completeness was regarded as an infeasible goal for program analysis, best abstract transformers allowed for a precise notion of relative-completeness.

Over the last decade there has been significant progress in understanding completeness aspects of abstract interpretation. Giacobazzi et al. [2000] and Giacobazzi and Quintarelli [2001] showed that in the Galois connection framework, completeness of a transformer can be characterised as a property of an abstract domain. This characterisation simplifies greatly the study of completeness because instead of studying fixed point iteration, transformers, and domains, it is sufficient to focus on the domain. A result of greater practical relevance is that the completion of an abstract domain and transformers has a constructive, greatest fixed point characterisation. A series of papers revealed that domain completion algorithms are as prevalent as least fixed point iteration. Instances of domain completion include counterexample guided abstraction refinement [Giacobazzi and Quintarelli 2001], bisimulation and simulation quotients [Ranzato and Tapparo 2007], and partition refinement algorithms [Ranzato and Tapparo 2008].

The prevalence of fixed points in computer science has led to applications of abstract interpretation far beyond data flow analysis. The literature concurrent to abstract interpretation is rife with papers specifying and verifying properties of programs using fixed points [Clarke 1977; Flon and Suzuki 1978; Kozen 1982; Queille and Sifakis 1982]. Abstract interpretation stands out for its focus on approximation. The crucial idea was that a fixed point expression in a different lattice yields an approximation of the desired fixed point, provided that certain soundness requirements are met.

A third consequence of the abstract interpretation framework was the introduction of systematic techniques for combining program analyses. Several combinations were described in purely mathematical terms in the final section of [Cousot and Cousot 1979]. Compared to the literature devoted to designing abstract domains and computing fixed points, the composition of program analyses has received relatively little attention. This dissertation will sadly do nothing to change that state of affairs.

3

STRUCTURES

> Nowadays, most people will associate modal logic primarily with relational structures, but, as with other branches of logic, the 19th century infancy of modern symbolic logic was completely algebraic. Somehow, during the 20th century however, the traditions of algebraic and modal logic got separated, and for decades proceeded without any interaction whatsoever. In particular, while Jónsson and Tarski introduced not only Boolean algebras with operators and their representation over relational structures, but also the rudiments of canonicity and correspondence theory, this seminal work did not mention modal logic and it was overlooked by modal logicians for many years. ... And it would even have to wait until the 1990s before the algebraic and modal traditions would be completely rejoined, with collaborations between modal and algebraic logicians
>
> ...
>
> But no matter how well-behaved these algebras are, most modal logicians will still prefer the relational semantics, either because they find it more intuitive, or because frames simply happen to be the structures in which they take an (application-driven) interest.
>
> – Yde Venema, *Algebras and Coalgebras*, 2006

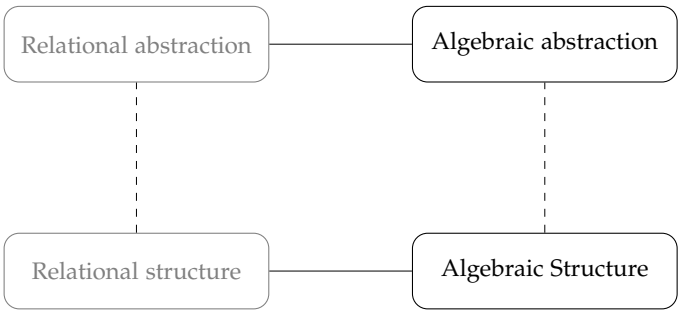
This chapter recalls existing mathematical structures for representing the behaviour of programs and introduces new structures with different expressive power. One conceptual contribution of this chapter is to introduce *conjunctive transition systems*, which are transition system representations of non-distributive abstractions. A second contribution is to introduce *trace algebras*, which provide algebraic models of linear- and branching-time behaviour. The main result of this chapter is a representation theorem, which shows that trace algebras are isomorphic to the sets of traces generated by a transition system. An additional contribution of this chapter is to show that structures arising in modal logic, program analysis and model checking arise as abstract interpretations of trace algebras.

3.1 OVERVIEW

Static analysis with lattices and model checking of transition systems are two families of approaches for automatically reasoning about programs. In approaches based on model checking, the semantics of a program is represented by a transition system or by a set of traces. In approaches based on abstract interpretation, the semantics of a program is represented by a lattice of states equipped with monotone functions called transformers. The results of Steffen [1991; 1993], which were popularised in expositions by Schmidt [1998] and by Schmidt and Steffen [1998], show that certain bit-vector data-flow analysis problems can be viewed as model checking problems. These results are often incorrectly interpreted as demonstrating an equivalence between lattice-based static analysis and model checking.

In this chapter, we apply the representation theorem of Jónsson and Tarski [1952] to show an isomorphism between labelled transition systems and perfect Boolean lattices with distributive transformers. A consequence of this result is that static analyses that use non-Boolean lattices cannot be directly translated to model checking problems over transition systems. Conversely, model checking problems for linear-time logics or for logics that combine linear- and branching-time modalities cannot be translated to static analysis problems over a lattice of states. The main problem we tackle in this chapter is to extend transition systems to model reasoning in non-Boolean lattices, and to develop a lattice-based model that supports reasoning about linear- and branching-time behaviour.

The approach we take to address these problems is summarised by the schema below. Every structure introduced in this chapter is either defined in terms of transition systems, in which case we call it *relational*, or is defined using lattices, in which case we call it *algebraic*. The solid lines below denote a representation theorem. We present representation theorems to translate between structures used in static analysis and in model checking. The dashed lines below denotes abstraction. The different algebraic structures we consider are related by Galois connections. It follows that the relational structures can also be viewed as abstractions of each other.



SUMMARY OF CONCEPTS The concepts introduced in this chapter are summarised in Figure 2 and recalled in detail below.

1. Transition systems with state and transition labels are presented in Definition 3.25 and are extended with a partial order to monotone transitions systems in Definition 3.34. Both these structures exist in the literature, but we are not aware of the saturation conditions in Definition 3.34 being articulated before.

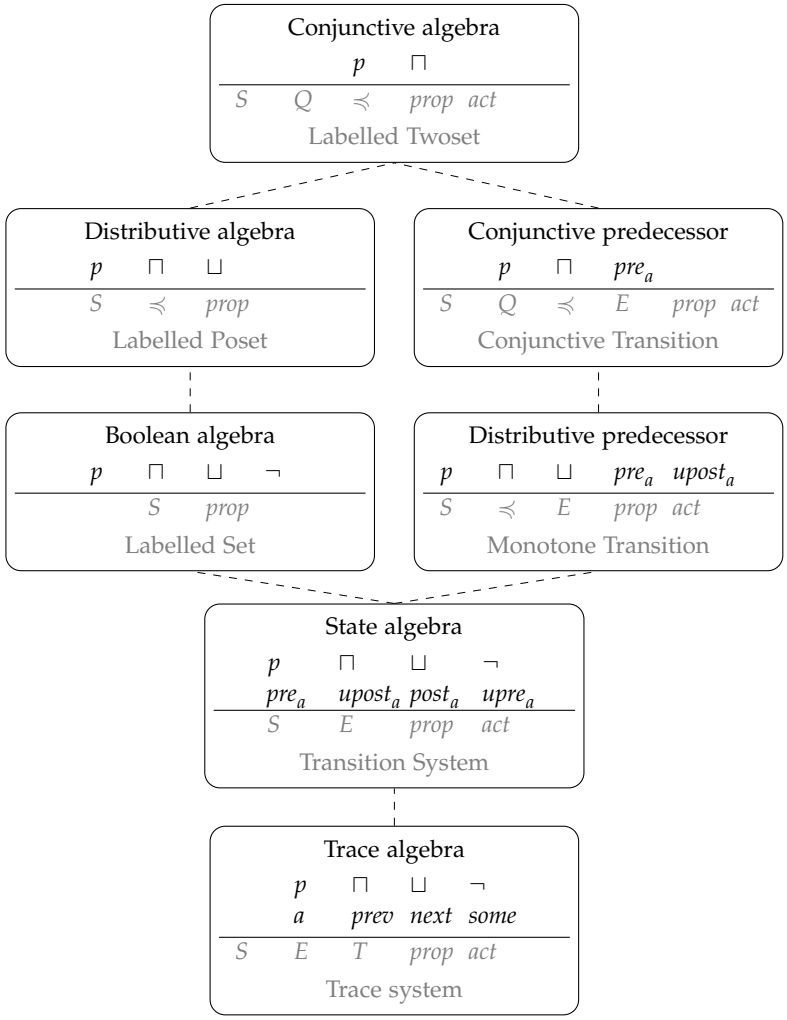


Figure 2: A guide to the contents of this chapter. Each box represents an algebra and a relational structure defined in this chapter. Trace algebras and conjunctive transition systems are new but all other structures exist in the literature. Each box summarises a representation theorem presented in this chapter.

2. Conjunctive transition systems, presented in Definition 3.47, extend monotone transitions to twosets. A related notion has recently been introduced in the modal logic literature by Gehrke [2006], but no similar notion exists in the program analysis or model checking literature.
3. State algebras, distributive predecessor algebras and conjunctive predecessor algebras all exist in the literature and are recalled in Definitions 3.28, 3.40 and 3.49, respectively.
4. The new notion of a trace system, which represent a set of traces generated by a transition system appears in Definition 3.56. Trace algebras, which appear in Definition 3.58, are also new.

SUMMARY OF RESULTS Each box in Figure 2 also summarises a representation theorem proved in this chapter. The representation theorems for conjunctive, distributive and Boolean algebras, and for state algebras exist in

the literature and are included in the chapter in the interest of completeness. The results contributed by this chapter are detailed below.

1. Theorem 3.23 shows that every conjunctive propositional algebra arises as a *γ-complete abstraction* of a distributive propositional algebra and that every distributive propositional algebra arises as a *γ-complete abstraction* of a propositional algebra. The notion of abstraction is made precise using closure operators.
2. Example 3.35 shows that there is a difference between *pre*-transition systems and saturated *pre*-transition systems. Example 3.36 illustrates the differences between *pre*-transition systems and *post*-transition systems. Neither of these distinctions have been pointed out in the literature.
3. Lemma 3.41 gives necessary and sufficient conditions for a transformer generated by a transition system to preserve downsets.
4. Proposition 3.48 strictly generalises the saturation conditions of monotone transition systems to conjunctive transition systems using Galois stable sets. Lemma 3.50 strictly generalises Lemma 3.41 by providing necessary and sufficient conditions for transformers generated by a transition system to preserve Galois stable sets.
5. Theorem 3.54 extends the abstraction perspective to structures over states. The theorem shows that every conjunctive predecessor algebra is an abstraction of a distributive predecessor algebra and that every distributive predecessor algebra is an abstraction of a predecessor algebra. It follows from the representation theorems in the chapter that conjunctive transition systems are abstractions of monotone transition systems, which in turn are abstractions of transition systems.
6. Theorem 3.68 is a new representation theorem for trace algebras. We are not aware of existing algebraic structures that model traces and admit such a representation theorem.
7. The final result of this chapter, Theorem 3.71, shows that state algebras corresponding to two-way total transition systems arise as abstractions of trace algebras.

CHAPTER ORGANISATION The chapter is organised as follows.

1. Algebraic and relational structures that represent atomic propositions and their Boolean combination are recalled in Section 3.2. The section recalls representation theorems for every type of structure introduced. The concepts and results in Section 3.2 are not new but are used in later sections to reduce notational overhead and to simplify proofs.
2. Structures that can represent states and state-transitions are introduced in Section 3.3. The section contains representation theorems relating extensions of transition systems and lattices with transformers, and results characterising the different types of orders that can be imposed on a transition system.
3. Trace algebras are introduced in Section 3.4. The main result of that section is a representation theorem.

Every section above concludes with an examination of the structures that were introduced from an abstract interpretation perspective.

3.2 PROPOSITIONAL STRUCTURES

Propositional logic can express atomic propositions, their conjunction, disjunction and negation. We study fragments of propositional logic that we call Boolean, monotone and conjunctive. This section presents equivalent

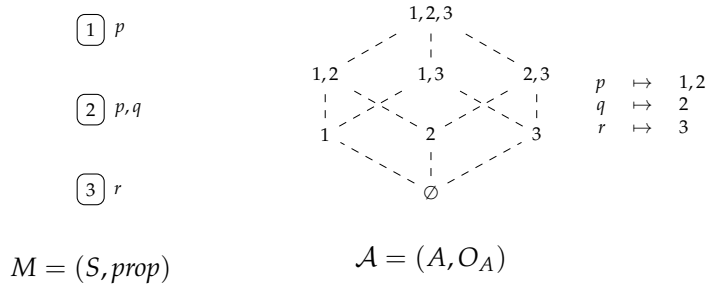


Figure 3: A labelled set $M = (S, prop)$ and its representation as a propositional algebra $\mathcal{A} = (A, O_A)$

set-theoretic and algebraic models for each fragment. The results in this section are not new but will be used later in the chapter to derive new results.

The set-theoretic model of propositional logic we use is a set labelled with propositional symbols. The algebraic models are perfect Boolean algebras (from Section 2.4). Propositional symbols become elements of the algebra and Boolean operations become meet, join and complement. The representation theorem for perfect Boolean algebras asserts that these views are equivalent.

If the formulae of interest do not contain negation, it suffices to consider sets closed under union and intersection. Algebraically, propositional symbols are elements of a perfect distributive lattice and conjunction and disjunction are meet and join. Such lattices have been studied in modal logic [Dunn 1995], static analysis of logic programs [Schachte and Søndergaard 2006] and in abstract interpretation [Cousot and Cousot 1979; Cousot and Cousot 1992a]. The representation theorem for perfect distributive lattices shows that such structures have poset representations.

It may sound strange to consider a logic with only propositional symbols and conjunction. The utility of this restriction becomes clear when quantifier-free first-order theories such as equality or linear arithmetic are considered. The satisfiability problem for such logics is NP-hard. The conjunctive fragments of theories such as linear real arithmetic, or the theory of equality with uninterpreted functions admit efficient, satisfiability algorithms that are the cornerstone of modern, high-performance satisfiability solvers. The algebraic models of propositional logic with only conjunction are perfect lattices. These algebras have representations as twosets (see Section 2.4 for the definition of twosets).

Boolean Propositional Structures

The example that follows illustrates a translation between an algebraic and a set-theoretic structure.

Example 3.1. A *labelled set* M is shown in Figure 3. It contains a set $S = \{1, 2, 3\}$ and a labelling function *prop* that maps each element of S to a subset of $\{p, q, r\}$. An algebraic representation $\mathcal{A}$ of M is also shown. The domain of $\mathcal{A}$ is the powerset of S. We omit the braces around lattice elements to reduce clutter. The symbols in $\{p, q, r\}$ are represented as lattice elements. ⌟

Formally, a *propositional signature* is a set

Prop

containing zero-arity *propositional symbols*. The set of propositional symbols used in the remainder of the chapter will always be *Prop*. Therefore, *Prop* is not mentioned when it is clear from the context. Labelled sets are defined next.

Definition 3.2. A *Prop-labelled set* $M = (S, prop)$ consists of a set S and a propositional labelling function $prop : S \to \mathscr{P}(Prop)$.

A notion of isomorphism between labelled sets is required to formalise that no information is lost in translating between sets and algebras. A homomorphism between the labelled sets $M_1 = (S_1, prop_1)$ and $M_2 = (S_2, prop_2)$ is a function $h : S_1 \to S_2$ satisfying $prop_1(s) = prop_2(h(s))$ for all s in S_1. An isomorphism is defined as expected and two sets are isomorphic, denoted $M_1 \cong M_2$, if there exists an isomorphism between them.

Definition 3.3. A *propositional algebra* $\mathcal{A} = (A, O_A)$ over a set of propositional symbols *Prop* is a perfect Boolean algebra A with an element $p^{\mathcal{A}}$ in A for each propositional symbol p in *Prop*.

A propositional algebra can be generated from a labelled set by taking the powerset of the set and by mapping propositional symbols to sets of elements based on their labels. A labelled set $M = (S, prop)$ defines an algebra $palg(M)$, also denoted $\mathcal{M}$, shown below.

$$palg(M) \mathrel{\hat{=}} (\mathscr{P}(S), O_M)$$
$$p^{\mathcal{M}} \mathrel{\hat{=}} \{s \mid p \text{ is in } prop(s)\}\text{, where } p^{\mathcal{M}} \text{ is in } O_M$$

Lemma 3.4. *A labelled set M generates a propositional algebra $palg(M)$.*

A labelled set can be obtained from a propositional algebra by taking the atoms of the algebra as set elements. The labelling function maps an atom to the set of propositional symbols greater than or equal to the atom. A propositional algebra $\mathcal{A} = (A, O_A)$ defines the following labelled set.

$$lset(\mathcal{A}) \mathrel{\hat{=}} (Atom(A), prop)$$
$$prop \mathrel{\hat{=}} \left\{x \mapsto \left\{p \mid x \sqsubseteq p^{\mathcal{A}}\right\} \mid x \in Atom(A)\right\}$$

The constructions above allow one to move between representations without loss of information (up to isomorphism).

Proposition 3.5. *A labelled set is isomorphic to the set generated by its propositional algebra: $M \cong lset(palg(M))$*

For the proof, observe that the map $s \mapsto \{s\}$ is an isomorphism between the labelled sets in Proposition 3.5. A corollary of the representation theorem for perfect Boolean algebras (Theorem 2.11) is a representation of propositional algebras as labelled sets.

Proposition 3.6. *Every propositional algebra is isomorphic to the algebra of its labelled set: $\mathcal{A} \cong palg(lset(\mathcal{A}))$.*

The proof is the same as that for perfect Boolean algebras (Theorem 10.24 in [Davey and Priestley 1990]). The map $x \mapsto Atom(x)$ is an isomorphism of propositional algebras.

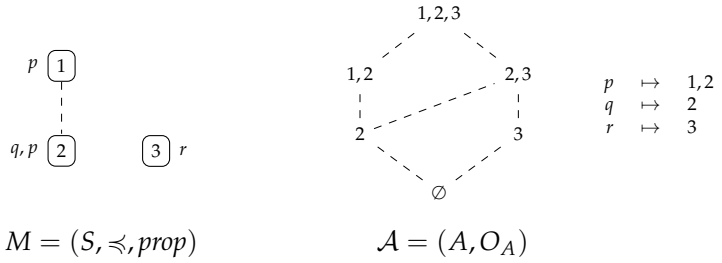


Figure 4: A labelled poset $M = (S, \preccurlyeq)$ and its representation as a distributive propositional algebra $\mathcal{A} = (A, O_A)$

Monotone Propositional Structures

A labelled set does not explicitly encode that some states are labelled with more symbols than others. Such information is useful because we can prove a fact about one element and conclude that the fact is true of other elements. In this section, elements in a labelled set are ordered by the symbols they satisfy to obtain labelled posets. The algebraic representations of labelled posets are perfect distributive lattices with propositional symbols. These structures are called *monotone*.

Example 3.7. A labelled poset M is shown in Figure 4. The element 1 is labelled with p, the element 2 is labelled with p and q and 3 is labelled with r. Every label of 1 is a label of 2 but the converse is not true. Let us order elements of S by inclusion of their labels. The order $s \preccurlyeq t$ holds if $prop(t) \subseteq prop(s)$. In the resulting poset, 2 is less than 1 because it is labelled with p as well as q. The downset lattice of $(S, \preccurlyeq)$ is shown in Figure 4. The lattice is not a Boolean algebra because $\{2, 3\}$ has no complement. ⌟

Definition 3.8. A *labelled poset* $M = (S, \preccurlyeq, prop)$ is a poset $(S, \preccurlyeq)$ with a labelling function that satisfies $prop(t) \subseteq prop(s)$ whenever $s \preccurlyeq t$.

The order inversion above is intentional and encodes that if $s \preccurlyeq t$, then every symbol labelling t is also a label of s. A labelled set homomorphism, when extended to a poset, must respect the order; the homomorphism must be monotone. A *labelled poset homomorphism* from $M = (S_1, \sqsubseteq_1, prop_1)$ to $M_2 = (S_2, \sqsubseteq_2, prop_2)$ is a monotone function $h : S_1 \to S_2$ that satisfies $prop_1(s) = prop_2(h(s))$ for every element s of S_1. Isomorphism is similarly defined and $M_1 \cong M_2$ denotes that M_1 and M_2 are isomorphic.

Definition 3.9. A *distributive propositional algebra* $\mathcal{A} = (A, O_A)$ is a perfect distributive lattice A with an element $p^{\mathcal{A}}$ in O_A for each symbol p in *Prop*.

A distributive propositional algebra is obtained from a labelled poset by taking downsets of the poset. The propositional symbols are defined as in the labelled set case. This construction is illustrated next.

Example 3.10. Revisit the poset $(S, \preccurlyeq)$ in Figure 4. The downsets of this poset constitute the lattice A in the figure.

$$\mathscr{D}(S) \triangleq \{\emptyset, \{2\}, \{3\}, \{1, 2\}, \{2, 3\}, \{1, 2, 3\}\}$$

Observe that $\mathscr{D}(S)$ does not contain $\{1\}$ because $\{1\}$ is not a downset. The poset $(S, \preccurlyeq)$ can be reconstructed from the algebra $\mathcal{A}$ by taking the

join-irreducibles of $\mathcal{A}$ as elements. The labelling function *prop* maps join-irreducibles to the propositional symbols above them.

$$\begin{aligned} Irr_{\sqcup}(A) &\triangleq \{\{2\},\{3\},\{1,2\}\} \\ prop &\triangleq \{2\} \mapsto \{p,q\},\{3\} \mapsto \{r\},\{1,2\} \mapsto \{p\} \end{aligned}$$

The labelled poset obtained is isomorphic to $(S, \preccurlyeq)$. ⌟

The constructions in the example are now formalised. A labelled poset $M = (S, \preccurlyeq, prop)$ defines an algebra $palg(M)$, also denoted $\mathcal{M}$.

$$\begin{aligned} palg(M) &\triangleq (\mathscr{D}(S), O_M) \\ p^{\mathcal{M}} &\triangleq \{s \in S \mid p \text{ is in } prop(s)\}, \text{ where } p^{\mathcal{M}} \text{ is in } O_M \end{aligned}$$

Lemma 3.11. *A labelled poset M defines a distributive propositional algebra $palg(M)$.*

Proof. We have to show that $\mathscr{D}(S)$ is a perfect distributive lattice and that each propositional symbol maps to a downset. From Proposition 2.15, the lattice of downsets is a perfect distributive lattice. If t is in $p^{\mathcal{M}}$, and $s \preccurlyeq t$, s is in $p^{\mathcal{M}}$ because $p \in prop(s)$ in a labelled poset, from which it follows that propositional symbols map to downsets. ⊣

Atoms are the building blocks of perfect Boolean algebras because every element is the join of the atoms below it. The elements of a distributive lattice, such as $(\mathbb{N}_\infty, \leq)$, are not join-generated by atoms but by the join-irreducibles. Consequently, the building blocks of a perfect distributive lattice are the join irreducibles. A distributive propositional algebra $\mathcal{A} = (A, O_A)$ defines the labelled poset below.

$$\begin{aligned} lset(\mathcal{A}) &\triangleq (Irr_{\sqcup}(A), \preccurlyeq, prop) \\ \preccurlyeq &\triangleq (Irr_{\sqcup}(A) \times Irr_{\sqcup}(A)) \cap \sqsubseteq \\ prop &\triangleq \left\{ x \mapsto \left\{ p \mid x \sqsubseteq p^{\mathcal{A}} \right\} \mid x \in Irr_{\sqcup}(A) \right\} \end{aligned}$$

Lemma 3.12. *The structure $lset(\mathcal{A})$ defined by a distributive propositional algebra $\mathcal{A}$ is a labelled poset.*

Proof. We need to show that $(Irr_{\sqcup}(\mathcal{A}), \preccurlyeq)$ is a poset, and that *prop* respects the order. The representation of perfect distributive lattices by posets shows that the join-irreducibles of A are a poset (Theorem 2.16). Let us verify the labelling condition. Consider join irreducibles x and y such that $x \preccurlyeq y$ and p is in $prop(y)$. Since $\preccurlyeq$ is a subset of the partial order $\sqsubseteq$ we have $x \sqsubseteq y$, and p being in $prop(y)$ implies $y \sqsubseteq p^{\mathcal{A}}$ yielding that p is in $prop(x)$ as required. ⊣

The statements that follow show that translating between labelled posets and distributive propositional algebras preserves structure up to isomorphism. As statements about labelled objects, these statements are new. However, they follow easily from existing results.

Theorem 3.13. *A labelled poset is isomorphic to the one generated by its algebra: $M \cong lset(palg(M))$.*

Proof. The algebra $\mathcal{M} = palg(M)$ contains downsets of M with the principal downsets $s\!\downarrow$ being join-irreducibles. From the representation of posets as downsets (Proposition 2.15), the function $h : s \mapsto s\!\downarrow$ is an order isomorphism from M to $lset(palg(M))$. Consider the labelling functions *prop* of M and

$prop_{\mathcal{M}}$ generated by $\mathcal{M}$. We show that $prop(s) = prop_{\mathcal{M}}(h(s))$ for every s in M. If $prop(s)$ contains the symbol p, the set $s{\downarrow}$ is a subset of $p^{\mathcal{M}}$ in the algebra $\mathcal{M}$, so p is in $prop_{\mathcal{M}}(h(s))$. On the other hand, if p is an element of $prop_{\mathcal{M}}(h(s))$, the downset $s{\downarrow}$ is a subset of $p^{\mathcal{M}}$ in the algebra, and by construction, the symbol p is an element of $prop(s)$. ⊣

Theorem 3.14. *A distributive propositional algebra is isomorphic to the algebra generated by its labelled poset:* $\mathcal{A} \cong palg(lset(\mathcal{A}))$.

Proof. Consider the distributive propositional algebras $\mathcal{A} = (A, O_A)$ and $\mathcal{M} = palg(lset(\mathcal{A}))$. The representation of perfect distributive lattices (Theorem 2.11) shows that map $h : x \mapsto Irr_{\sqcup}(x){\downarrow}$ is an isomorphism from the lattice A in $\mathcal{A}$ to the downset lattice in $\mathcal{M}$. It remains to show that $h(p^{\mathcal{A}}) = p^{\mathcal{M}}$. A join-irreducible x of A is labelled p in $lset(\mathcal{A})$ exactly if $x \sqsubseteq p^{\mathcal{A}}$. The set $h(p^{\mathcal{A}})$ is precisely $p^{\mathcal{M}}$. ⊣

Conjunctive Propositional Structures

A labelled poset generates a family of sets closed under union and intersection. The order tells us which sets can be disregarded for models of a logic without negation. To model intersection closed sets, additional information is required to dictate which unions can be disregarded. Recall that a twoset (Section 2.4) is a poset containing join- and meet-irreducible elements. Labelled twosets are set-theoretic representations of conjunctive structures. The algebraic conjunctive structures are perfect lattices with propositional symbols. Observe that when fewer logical properties are considered the set-theoretic structures become more sophisticated while algebraic structures become simpler.

Example 3.15. This example illustrates the perfect lattice generated by a labelled twoset. A labelled twoset $M = (S, Q, \preccurlyeq, prop)$ is given in Figure 5. The sets S and Q are identical and $\preccurlyeq$ is the identity relation. Recall from Section 2.4 that the order in a twoset defines two functions u and ℓ that map elements of S and Q, respectively, to their upper and lower approximations in the other set. The function $u : \mathscr{P}(S) \to \mathscr{P}(Q)$ is defined below.

$$u \triangleq \{\emptyset \mapsto \{1,2,3\}, \{1\} \mapsto \{1\}, \{2\} \mapsto \{2\}, \{3\} \mapsto \{3\}, \\ \{1,2\} \mapsto \emptyset, \{1,3\} \mapsto \emptyset, \{2,3\} \mapsto \emptyset, \{1,2,3\} \mapsto \emptyset\}$$

The function $\ell : \mathscr{P}(Q) \to \mathscr{P}(S)$ is identically defined. A Galois-stable subset $X \subseteq S$ is one satisfying $X = \ell(u(X))$. The Galois-stable subsets of M, denoted $\mathscr{G}(M)$ and shaded in the figure, consist of S, the empty set, and the singleton sets.

$$\mathscr{G}(M) \triangleq \{\emptyset, \{1\}, \{2\}, \{3\}, \{1,2,3\}\}$$

The generated lattice does not satisfy the distributive law.

$$\{3\} = (\{1\} \sqcap \{2\}) \sqcup \{3\}, \text{ but } \quad \{1,2,3\} = (\{1\} \sqcup \{3\}) \sqcap (\{2\} \sqcup \{3\})$$

Consequently, $\mathscr{G}(M)$ lacks a powerset or downset representation. ⌟

Definition 3.16. A *labelled twoset* $M = (S, Q, \preccurlyeq, prop)$ is a twoset with a labelling function $prop : S \to \mathscr{P}(Prop)$ satisfying that, for every s and t in S, and every r in Q, if $s \preccurlyeq r$ implies $t \preccurlyeq r$, then $prop(s)$ is contained in $prop(t)$.

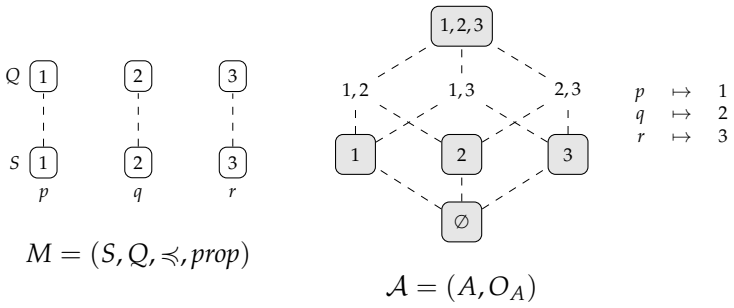


Figure 5: A labelled twoset $M = (S, Q, \preccurlyeq, prop)$ and its representation as a conjunctive propositional algebra $\mathcal{A} = (A, O_A)$. A is the lattice of shaded elements, which are also Galois stable subsets of $\mathscr{P}(S)$

Note that the labelling function is only defined over elements of S and not over elements of Q. In a labelled poset, the order $s \preccurlyeq t$ intuitively implies that s satisfies all the properties that t satisfies. A twoset can also represent elements s and t that satisfy the same properties but are incomparable. To do this, we introduce a new relation between elements of S to Q. Define the *Galois relation* of a twoset M to contain a pair (y, x) if every element in Q that is greater than or equal to x is also greater than or equal to y.

$$G_M \triangleq \{(y, x) \in S \times S \mid \text{ for all } z \text{ in } Q, x \preccurlyeq z \text{ implies } y \preccurlyeq z\}$$

In the representation of posets, the join-irreducible generated by an element s is the principal downset $\mathscr{D}(s)$, which is the preimage of s with respect to the partial order. In the representation of a twoset, the corresponding notion is of a *principal Galois-stable set*, which is the preimage of s with respect to the Galois-relation.

Proposition 3.17. *The Galois relation of a labelled twoset M contains (y, x) exactly if y is in $\ell(u(x))$.*

An isomorphism between a labelled twoset $M_1 = (S_1, Q_1, \preccurlyeq_1, prop_1)$ and $M_2 = (S_2, Q_2, \preccurlyeq_2, prop_2)$, denoted $M_1 \cong M_2$, is an order isomorphism h between $(S_1 \cup Q_1, \preccurlyeq_1)$ and $(S_2 \cup Q_2, \preccurlyeq_2)$ such that $prop_1(s) = prop_2(h(s))$ for all s in S_1. The algebraic counterpart of a labelled twoset is a conjunctive algebra, introduced next.

Definition 3.18. A *conjunctive propositional algebra* $\mathcal{A} = (A, O_A)$ is a perfect lattice A with an element $p^{\mathcal{A}}$ in O_A for each propositional symbol p.

Just as labelled sets generate powerset lattices, and labelled posets generate downset lattices, labelled twosets generate a lattice of Galois-stable sets. A labelled twoset $M = (S, Q, \preccurlyeq, prop)$ generates an algebra $palg(M)$ of Galois-stable sets, also denoted $\mathcal{M}$.

$$palg(M) \triangleq (\mathscr{G}(M), O_M)$$
$$p^{\mathcal{M}} \triangleq \{s \in S \mid p \text{ is in } prop(s)\}, \text{ where } p^{\mathcal{M}} \text{ is in } O_M$$

Lemma 3.19. *A labelled twoset M defines a conjunctive propositional algebra $palg(M)$.*

Proof. The domain of $palg(M)$ is a perfect lattice because the Galois stable subsets of a twoset form a perfect lattice (Theorem 2.19). We show that $p^{\mathcal{M}}$

is a Galois stable set. Suppose s is in $p^{\mathcal{M}}$ and t is in the same Galois-stable set as s. For every r in Q, the inequality $s \preccurlyeq r$, if true, implies $t \preccurlyeq r$. In a labelled twoset p is in $prop(t)$ for such t. ⊣

In order to generate twosets from a conjunctive propositional algebra we need to consider both join and meet-irreducibles. Example 2.17 in Section 2.4 illustrates that unlike the distributive case, the sets of join- and meet-irreducibles of perfect lattices need not be isomorphic. The labelling function is defined only over join-irreducibles as before. A conjunctive propositional algebra $\mathcal{A} = (A, O_A)$ generates the structure $lset(\mathcal{A})$ below.

$$\begin{aligned} lset(\mathcal{A}) &\triangleq (Irr_{\sqcup}(A), Irr_{\sqcap}(A), \preccurlyeq, prop) \\ \preccurlyeq &\triangleq (Irr_{\sqcup}(A) \cup Irr_{\sqcap}(A))^2 \cap \sqsubseteq \\ prop &\triangleq \left\{ x \mapsto \left\{ p \mid x \sqsubseteq p^{\mathcal{A}} \right\} \mid x \in Irr_{\sqcup}(A) \right\} \end{aligned}$$

Lemma 3.20. *The structure $lset(\mathcal{A})$ defined by a conjunctive propositional algebra $\mathcal{A}$ is a labelled twoset.*

Proof. The domain of $\mathcal{A}$ is a perfect lattice, so $lset(\mathcal{A})$ contains a twoset. Let us verify the condition on labels. Consider a symbol p in the label $prop(x)$ of a join irreducible x. Consider also a join irreducible y that satisfies $y \sqsubseteq z$ for all meet irreducibles z such that $x \sqsubseteq z$. The set $Irr_{\sqcap}(x)$ is contained in $Irr_{\sqcap}(y)$, so $\bigsqcap Irr_{\sqcap}(y) \sqsubseteq \bigsqcap Irr_{\sqcap}(x)$, in turn implying the inequality $y \sqsubseteq x$, because the lattice is perfect. Since $x \sqsubseteq p^{\mathcal{A}}$, it also holds that p is in $prop(y)$. ⊣

The translations between propositional conjunctive algebras and labelled twosets preserve structure, up to isomorphism.

Theorem 3.21. *A conjunctive propositional algebra is isomorphic to the algebra of its labelled twoset: $\mathcal{A} \cong palg(lset(\mathcal{A}))$.*

Proof. Let M be the twoset generated by $\mathcal{A}$ and $\mathscr{G}(M)$ be the Galois-stable subsets of M. From the representation of perfect lattices by twosets (Theorem 2.20) the map $h : x \mapsto Irr_{\sqcup}(x)$ is an isomorphism. Let us show that this isomorphism preserves symbols. The elements of $lset(\mathcal{A})$ labelled with a symbol p are join-irreducibles satisfying $x \sqsubseteq p^{\mathcal{A}}$. These are exactly the elements x satisfying $Irr_{\sqcap}(p^{\mathcal{A}}) \subseteq Irr_{\sqcap}(x)$, hence the elements in $Irr_{\sqcup}(p^{\mathcal{A}})$. ⊣

Theorem 3.22. *A labelled twoset M is isomorphic to the labelled twoset of its algebra: $M \cong lset(palg(M))$.*

Proof. The major step of this proof follows from Theorems 3.8 and 3.9 in [Davey and Priestley 1990]. Let the labelled twoset be $M = (S, Q, \preccurlyeq, prop)$, let G_M be the Galois relation generated by M and let P_M be the set of principal Galois-stable subsets of S. The quoted theorem shows that every element of P_M is a join-irreducible of $\mathscr{G}(M)$ and the set P_M is join-dense in $\mathscr{G}(M)$. The same theorem also states that for each q in Q, the set $\ell(q)$ is meet-irreducible in $\mathscr{G}(M)$ and the set $R_M \triangleq \{\ell(q) \mid q \in Q\}$ is meet-dense in $\mathscr{G}(M)$. It follows that $N = (P_M, R_M, \subseteq)$ is a twoset. Finally, the maps above extend to an isomorphism between the posets in M and N.

We need to check the condition on labelling functions. Suppose p is in $prop(x)$ for an element x in S. The labelling function satisfies that if (y, x) is in the Galois relation G_M, then y is in $\ell(u(x))$, and moreover $\ell(u(y))$ is contained in $\ell(u(x))$. Thus, if p is in $prop(\ell(u(x)))$ it is also in the label of $\ell(u(y))$, as required. ⊣

The Abstraction Perspective

Boolean, monotone and conjunctive state structures are related by abstract interpretation. Every conjunctive propositional algebra is an abstraction of a distributive propositional algebra in the sense that every perfect lattice is the image of a perfect distributive lattice under a closure operator, and labels are preserved. Every distributive propositional algebra is an abstraction of a propositional algebra in the sense that every perfect distributive lattice is the image of a perfect Boolean algebra under a closure operator and labels are preserved. We make these statements precise below.

Consider a conjunctive propositional algebra $\mathcal{C} = (C, O_C)$. The algebra $\mathcal{C}$ defines the algebras $\mathcal{D} = (D, O_D)$ and $\mathcal{B} = (B, O_B)$ below.

$$
\begin{array}{ll}
D \mathrel{\hat{=}} \mathscr{D}(Irr_{\sqcup}(C), \preccurlyeq) & B \mathrel{\hat{=}} \mathscr{P}(Irr_{\sqcup}(C)) \\
\preccurlyeq \mathrel{\hat{=}} (Irr_{\sqcup}(C) \times Irr_{\sqcup}(C)) \cap \sqsubseteq & \\
p^{\mathcal{D}} \mathrel{\hat{=}} Irr_{\sqcup}(p^{\mathcal{C}}) & p^{\mathcal{B}} \mathrel{\hat{=}} Irr_{\sqcup}(p^{\mathcal{C}})
\end{array}
$$

The definitions above imply that $\mathcal{D}$ is a distributive propositional algebra and that $\mathcal{B}$ is a propositional algebra. To relate the algebras $\mathcal{B}$ and $\mathcal{D}$ to the algebra $\mathcal{C}$, we define the two functions below.

$$
\begin{array}{ll}
\textit{b-to-d} : B \to B & \textit{d-to-c} : D \to D \\
\textit{b-to-d} \mathrel{\hat{=}} X \mapsto X\downarrow & \textit{d-to-c} \mathrel{\hat{=}} Y \mapsto \ell(u(Y))
\end{array}
$$

The functions are closure operators and preserve the interpretation of propositional symbols. In the terminology of abstract interpretation, every conjunctive propositional algebra arises as a γ-complete abstraction of a distributive propositional algebra and every distributive propositional algebra arises as a γ-complete abstraction of a propositional algebra.

Theorem 3.23. *Let $\mathcal{C}$ be a conjunctive propositional algebra defining the distributive propositional algebras $\mathcal{D}$ and the propositional algebra $\mathcal{B}$ as above.*

1. *The function b-to-d is an upper closure satisfying that b-to-d(B) is isomorphic to D and that $p^{\mathcal{D}} =$ b-to-d$(p^{\mathcal{B}})$.*
2. *The function d-to-c is an upper closure satisfying that d-to-c(D) is isomorphic to C and that $Irr_{\sqcup}(p^{\mathcal{C}}) =$ d-to-c$(p^{\mathcal{D}})$.*

The proof follows from the representation theorems proved earlier.

SECTION SUMMARY This section recalled set-theoretic and algebraic structures for modelling fragments of propositional logic and made their relation precise using representation theorems and abstraction functions. We recalled the definitions of propositional algebras, which are defined over perfect Boolean algebras, distributive propositional algebras, which are defined over perfect distributive lattices, and conjunctive propositional algebras, which are defined over perfect lattices. We showed that these algebras have representations as labelled sets, labelled posets, and labelled twosets, respectively.

3.3 STATE STRUCTURES

We now study structures that model the states of a program may reside in and model how these states change over time. Such models are used in model checking and in static analysis. We introduce Boolean, monotone and conjunctive state structures, corresponding to logics that have state-based

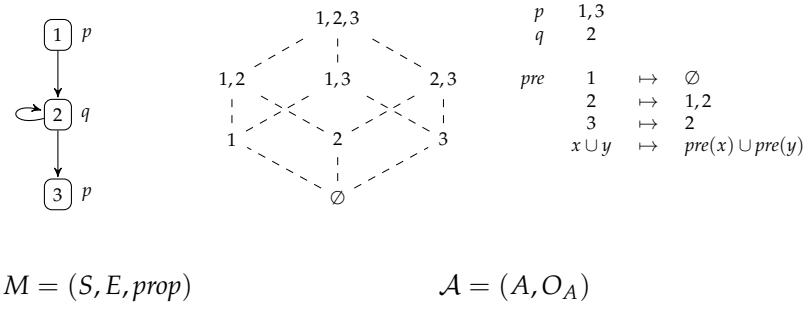


Figure 6: A transition system M and the algebra $\mathcal{A}$ that it generates

modalities and are closed under Boolean operations, monotone operations, and conjunction, respectively. The Boolean structures are well known in the literature under names such as collecting semantics and labelled transition systems. The monotone and conjunctive structures have recently been studied in algebraic modal logic and in model checking of infinite state systems. The contribution of this section is that existing results in algebraic logic provide relational representations of objects studied in lattice-based program verification. No such translations exist for non-Boolean lattices.

Boolean State Structures

We now study transition systems and state algebras. Transition systems are a standard, relational model of discrete, dynamic systems. A transition system consists of a set of states and a transition relation describing state changes. Labelling functions annotate states and transitions with the propositions they satisfy. In technical terms, the transition systems we consider provide semantics for logics with state and event-modalities.

The algebraic representation of a transition system is a perfect Boolean algebra with four functions called state transformers. Elements of the algebra represent sets of states. The existential successor transformer maps a set of states to those states the system may be in after a transition. The universal successor transformer maps a set of states to those states the system must be in after a transition. Symmetrically, the existential predecessor transformer maps a set of states to those states the system may have been in before making a transition. The universal predecessor transformer maps a set of states to those states the system must have been in before making a transition. The example below illustrates a transition system and a state algebra.

Example 3.24. A transition system M is shown in Figure 6. It contains a set of states S labelled with sets of propositions and a set of transitions E between states. An algebraic representation of M is the structure $\mathcal{A}$ in the figure, called a predecessor algebra. The states of M form the powerset lattice $\mathscr{P}(\{1,2,3\})$. The transition relation generates a predecessor transformer *pre*, that maps a set of states X to their preimage with respect to E. Propositions map to sets of states. Observe that if $\{s\} \subseteq p^{\mathcal{A}}$ for a state s, the label of s contains p. There is a transition from t to s if and only if t is in $pre(\{s\})$. ⌟

TRANSITION SYSTEMS Fix a set of proposition symbols *Prop* and a set of action symbols *Act* for the rest of this section.

Definition 3.25. A *transition system* $M = (S, E, prop, act)$ consists of states S, a transition relation $E \subseteq S \times S$, a state labelling function $prop : S \to \mathscr{P}(Prop)$ and a transition labelling function $act : E \to \mathscr{P}(Act)$.

In a transition (s, t), s is called the *predecessor* of t and t is the *successor* of s. The set of transitions whose label contains a is denoted E_a. Transitions are assumed to have non-empty labels: $act(s, t) \neq \emptyset$ for all transitions in E. This assumption is not a limitation but is useful for avoiding unwieldy case distinctions later. In illustrations all transitions are assumed to have the same label and none are shown. Fix a transition system $M = (S, E, prop, act)$ for this section. A transition systems homomorphism is a relational homomorphism (Section 2.5) that is also a labelled set homomorphism. Isomorphism of transition systems is defined and denoted $M \cong N$.

STATE ALGEBRAS The algebraic representation of transition systems follows directly from the work of Jónsson and Tarski [1952]. The terminology and classification we use is different from what exists in the literature. The distinctions we make are required later to identify notions that generalise to non-Boolean settings and those that don't. A transition relation generates four transformers from its image, pre-image and their De Morgan duals. Different signatures are obtained based on the choice of operators.

$Prop \cup \{pre_a \mid a \in Act\}$	Existential predecessor
$Prop \cup \{post_a \mid a \in Act\}$	Existential successor
$Prop \cup \{upre_a \mid a \in Act\}$	Universal predecessor
$Prop \cup \{upost_a \mid a \in Act\}$	Universal successor
$Prop \cup \{pre_a, upre_a \mid a \in Act\}$	Monotone predecessor
$Prop \cup \{post_a, upost_a \mid a \in Act\}$	Monotone successor
$Prop \cup \{post_a, pre_a, upost_a, upre_a \mid a \in Act\}$	State signature

The transformer pre_a is the *existential predecessor* with respect to a and $post_a$ is the *existential successor* with respect to a. We will drop the word existential when referring to these transformers. The transformer $upre_a$ is the *universal predecessor* with respect to a and $upost_a$ is the *universal successor* with respect to a. In the context of programs, $post_a$ is called the strongest postcondition of a and $upre_a$ is the weakest liberal precondition of a. A *predecessor signature* contains only propositions and predecessor symbols. The other signatures are named as expected. A *state signature* contains all four transformers.

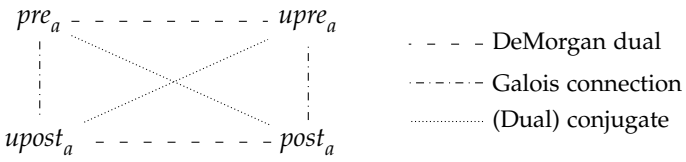
Example 3.26. We illustrate the different transformers on the lattice A in Figure 6. The successor transformer maps a lattice element X to its image under the transition relation. The universal predecessor transformer maps X to the set of states with all their successors in X. The universal successor transformer maps X to the set of states with all their predecessors in X.

$$
\begin{aligned}
pre &\triangleq \emptyset \mapsto \emptyset, \{1\} \mapsto \emptyset\, \{2\} \mapsto \{1,2\}\,, \{3\} \mapsto \{2\} \\
&\quad X \cup Y \mapsto post(X) \cup post(Y) \\
post &\triangleq \emptyset \mapsto \emptyset, \{1\} \mapsto \{2\}\,, \{2\} \mapsto \{2,3\}\,, \{3\} \mapsto \emptyset, \\
&\quad X \cup Y \mapsto post(X) \cup post(Y) \\
upre &\triangleq \{1,2,3\} \mapsto \{1,2,3\}\,, \{1,2\} \mapsto \{1,2,3\}\,, \{1,3\} \mapsto \{3\}\,, \\
&\quad \{2,3\} \mapsto \{1,2,3\}\,, X \cap Y \mapsto upre(X) \cap upre(Y) \\
upost &\triangleq \{1,2,3\} \mapsto \{1,2,3\}\,, \{1,2\} \mapsto \{1,2\}\,, \{1,3\} \mapsto \{1\}\,, \\
&\quad \{2,3\} \mapsto \{3\}\,, X \cap Y \mapsto upre(X) \cap upre(Y)
\end{aligned}
$$

Observe that the sets $upre(\emptyset)$ and $upost(\emptyset)$ need not be empty. The operators above satisfy various properties. First, the transformers *pre* and *post* are strict additive, while *upre* and *upost* are strict multiplicative. Second, for two elements X and Y in A, $pre(X) \cap Y$ is empty if and only if $X \cap post(Y)$ is empty. Third, for two elements X and Y in A, $pre(X) \subseteq Y$ if and only if $X \subseteq upost(Y)$ and the same property is true of *post* and *upre*. Finally, $upre(X)$ is equal to $\neg\, pre(\neg X)$, meaning *upre* is the De Morgan dual of *pre*. Similarly, *upost* is the De Morgan dual of *post*. A lattice A with four transformers satisfying the properties indicated above is called a state algebra. ⌟

Definition 3.27. A *predecessor algebra* $\mathcal{A} = (A, O_A)$ over a predecessor signature is a perfect Boolean algebra with strict additive transformers pre_a. A *successor algebra* is similarly defined with strict additive transformers $post_a$. Universal predecessor and universal successor algebras are defined similarly except that $upre_a$ and $upost_a$ are strict multiplicative.

In a perfect Boolean algebra, a strict additive transformer has a unique De Morgan dual, conjugate and right adjoint. As a result, each transformer above gives rise to three others.



The predecessor transformer pre_a has $upre_a$ as its De Morgan dual and $upost_a$ as its right adjoint. The transformer pre_a and $post_a$ are conjugate while $upre_a$ and $upost_a$ are dual conjugate.

Definition 3.28. A *state algebra* $\mathcal{A} = (A, O_A)$, over a state signature is a perfect Boolean algebra with a strict additive transformers pre_a having De Morgan dual $upre_a$, right adjoint $upost_a$ and conjugate $post_a$.

Since one transformer defines all the others, it suffices to consider only one when dealing with state algebras. The algebras introduced above are collectively called *Boolean state algebras*.

A REPRESENTATION THEOREM The translation from transition systems to predecessor and successor algebras is well known and is frequently used in the verification literature. Instances of this translation are discussed at the end of this chapter. A predecessor algebra defines a transition system in which atoms become states and there is a transition from an atom x to an atom y if x is below the predecessor of y. Equivalent translations may be obtained using universal transformers. These, being less known, are presented below. A transition system $M = (S, E, prop, act)$ defines an algebra $salg(M) = (\mathscr{P}(S), O_M)$ over a state signature, also denoted $\mathcal{M}$.

$$
\begin{aligned}
p^{\mathcal{M}} &\triangleq \{s \mid p \text{ is in } prop(s)\} \\
post_a^{\mathcal{M}}(X) &\triangleq \{t \mid \text{ for some } s \in X, (s,t) \in E \text{ and } a \in act(s,t)\} \\
pre_a^{\mathcal{M}}(X) &\triangleq \{s \mid \text{ for some } t \in X (s,t) \in E \text{ and } a \in act(s,t)\} \\
upost_a^{\mathcal{M}}(X) &\triangleq \{t \mid \text{ for every } s \in S, (s,t) \in E \text{ implies } t \in X\} \\
upre_a^{\mathcal{M}}(X) &\triangleq \{s \mid \text{ for every } t \in S, (s,t) \in E \text{ implies } t \in X\}
\end{aligned}
$$

This construction is standard. Cousot [1981] proves that the transformers obtained satisfy the additivity, multiplicity and adjointness conditions. The conjugacy condition is not standard in the literature, so we cover that case.

Lemma 3.29. *A transition system M generates a state algebra $salg(M)$.*

Proof. Consider a transition system $M = (S, E, prop, act)$, the algebra $salg(M)$, and two sets of states X and Y. We show that pre_a and $post_a$ are conjugate.

$$\begin{aligned}
& post_a(X) \cap Y \neq \emptyset \\
\iff & \text{there are } s \in X \text{ and } t \in Y \text{ such that } (s,t) \in T \text{ and } a \in act(s,t) \\
\iff & \text{there exists } s \in X \text{ such that } s \in pre_a(Y) \\
\iff & X \cap pre_a(Y) \neq \emptyset
\end{aligned}$$

It follows that pre_a and $post_a$ are conjugate. The proofs that the other requirements hold are covered in [Cousot 1981]. ⊣

A transition system generates a powerset lattice of states and four transformers. In the reverse direction, each transformer generates a relation. A state algebra defines a relation in four different ways. A state algebra $\mathcal{A} = (A, O_A)$ defines a transition system $srel(\mathcal{A}) = (S_\mathcal{A}, E_\mathcal{A}, prop_\mathcal{A}, act_\mathcal{A})$ as follows. The relation $E_\mathcal{A}$ is defined to be E_{pre} below.

$$\begin{aligned}
S_\mathcal{A} &\triangleq Atom(A) \\
E_{pre} &\triangleq \{(x,y) \mid x \sqsubseteq pre_a(y) \text{ for some action } a\} \\
E_{post} &\triangleq \{(x,y) \mid y \sqsubseteq post_a(x) \text{ for some action } a\} \\
E_{upre} &\triangleq \{(x,y) \mid upre_a(\neg x) \sqsubseteq \neg y \text{ for some action } a\} \\
E_{upost} &\triangleq \{(x,y) \mid upost_a(\neg x) \sqsubseteq \neg y \text{ for some action } a\} \\
prop_\mathcal{A}(x) &\triangleq \left\{p \in Prop \mid x \sqsubseteq p^\mathcal{A}\right\} \\
act_\mathcal{A}(x,y) &\triangleq \{a \in Act \mid x \sqsubseteq pre_a(y)\}
\end{aligned}$$

Although the predecessor transformer is used to generate the relation $E_\mathcal{A}$, Lemma 3.30 shows that all four transformers define the same relation.

Lemma 3.30. *The four relations defined by a state algebra are equivalent.*

Proof. Consider atoms x and y. There are three cases to consider.
($E_{pre} = E_{post}$)

$$\begin{aligned}
& (x,y) \text{ is in } E_{pre} \\
\iff\ & x \sqsubseteq pre_a(y) \iff x \sqcap pre_a(y) \neq \bot \\
\iff\ & post_a(x) \sqcap y \neq \bot \text{ because } post_a \text{ and } pre_a \text{ are conjugate} \\
\iff\ & y \sqsubseteq post_a(x) \text{ because } y \text{ is an atom} \\
\iff\ & (x,y) \text{ is in } E_{post}
\end{aligned}$$

($E_{pre} = E_{upre}$)

$$\begin{aligned}
& (x,y) \text{ is in } E_{pre} \\
\iff\ & x \sqsubseteq pre_a(y) \iff \neg\, pre_a(\neg\neg y) \sqsubseteq \neg x \\
\iff\ & upre_a(y) \sqsubseteq x \text{ because } upre_a \text{ is the De Morgan dual of } pre_a\,. \\
\iff\ & (x,y) \text{ is in } E_{post}
\end{aligned}$$

The case $E_{upost} = E_{post}$ is similar to the second case above. ⊣

Theorems 3.31 and 3.32, which follow, show that there are structure preserving translations between transition systems and state algebras. These results are not new but are included for completeness.

Theorem 3.31. *A transition system is isomorphic to the one generated by its state algebra:* $M \cong srel(salg(M))$.

Jónsson and Tarski's representation theorem shows that $s \mapsto \{s\}$ is an isomorphism of relational systems. It is straightforward that labels are preserved. All the four different kinds of algebras over states have representations as transition systems. It suffices to consider predecessor algebras.

Theorem 3.32. *A predecessor algebra is isomorphic to the algebra of its transition system:* $\mathcal{A} \cong salg(srel(\mathcal{A}))$.

The proof is a minor extension of Jónsson and Tarski's theorem.

Proof. The predecessor algebra $\mathcal{A} = (A, O_A)$ is a Boolean algebra with operators. Let $\mathcal{M}$ be $salg(srel(\mathcal{A}))$. By Jónsson and Tarski's theorem, the function $h : x \mapsto Atom(x)$ is an isomorphism of BAOs. A proposition p is in $prop(x)$ for an atom x exactly if $x \sqsubseteq p^{\mathcal{A}}$, so $h(p^{\mathcal{A}}) = Atom(p^{\mathcal{A}})$, and consequently $h(p^{\mathcal{A}})$ is $p^{\mathcal{M}}$. ⊣

When defined over perfect Boolean algebras, the transition system obtained from each transformer is equivalent. In the non-Boolean lattices considered next, this equivalence no longer holds.

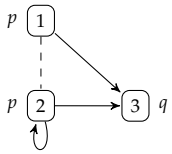
Monotone State Structures

A monotone logic is one that is not closed under negation. Monotone logics are interesting because there are families of programs that admit decidable verification in a monotone logic, but not in the extension of the same logic with negation. For example, checking negation-free μ-calculus properties of two-dimensional, rectangular, hybrid automata is decidable, but checking arbitrary μ-calculus properties is not [Henzinger et al. 1995]. We now study structures modelling monotone logics.

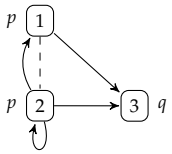
Monotone transition systems contain an order that constrains the transition relation. Such transition systems have gained significant attention in the last decade because they provide a uniform setting for proving decidability results in verification [Abdulla et al. 2000; Finkel and Schnoebelen 2001]. Monotone state algebras are distributive lattices with transformers. The relationship between the transformers cannot depend on complements. In particular, they cannot be De Morgan duals. Such algebras have been studied in the modal logic literature [Gehrke and Jónsson 1994; Dunn 1995] and in static analysis [Cousot and Cousot 1992a]. There are several ways in which an order can be defined on a transition system or an algebra. Consequently, the literature contains several, subtly different definitions.

The contribution of this section is to show that results from modal logic apply to translate between structures studied in transition system and lattice-based verification. A second contribution is to relate the different orders that can be defined on a transition system.

MONOTONE TRANSITION SYSTEMS Monotone transition systems extend labelled posets with a transition relation. Suppose s and t are states in a transition system and s is less than t. The order intuitively represents that any behaviour of t is also a behaviour of s. Different interpretations of the



$M = (S, \preccurlyeq, E, prop)$



$N = (S, \preccurlyeq, F, prop)$

Figure 7: Two predecessor ordered transition systems M and N. The system M is not saturated but N is. Both transition systems are also successor ordered

word behaviour yield different monotone transition systems. If a behaviour is the set of transitions from a state, every transition from t should have a corresponding transition from s such that the resulting states are ordered. If behaviour is defined to be incoming transitions, every transition to t should have a corresponding transition to s such that the source states are ordered. If transitions to and from a state are considered, both conditions must be satisfied. The next examples illustrates two different ways to define an order on a transition system.

Example 3.33. A transition system M is shown in Figure 7. The states of M are ordered by $\preccurlyeq$. Only the non-identity edges of $\preccurlyeq$ are shown.

$$\preccurlyeq \triangleq \{(1,1),(2,2),(3,3),(2,1)\}$$

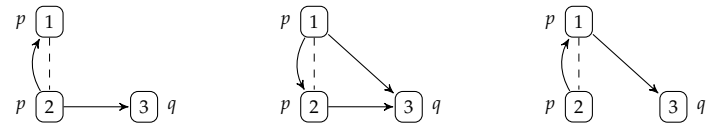
Observe that 2 is less than 1 in M. The order has two consequences. Every proposition labelling 1 also labels 2 and every transition from 1 can be mirrored by 2. We cannot make the converse observation because 2 can transition to itself, but 1 cannot transition to either 1 or 2. More generally, if $s \preccurlyeq t$, every label of t is a label of s. Furthermore, if there is a transition (t, t') there is also a transition (s, s') such that $s' \preccurlyeq t'$. The transition relation and the order satisfy the set inequality: $\preccurlyeq \circ E \subseteq E \circ \preccurlyeq$. A transition system in which this inequality holds is called a *pre-transition system* because an order on predecessors implies an order on successors.

Consider the self transition of state 2. The set of predecessor of 1, being empty, is contained in the set of predecessors of 2 and satisfies the inequality $\preccurlyeq \circ E^{-1} \subseteq E^{-1} \circ \preccurlyeq$. A transition system satisfying this condition is called a *post-transition system* . ⌟

The notions in the example are made formal below. In the definitions below, the set of transitions labelled a is denoted E_a.

Definition 3.34. A *monotone transition system* $M = (S, \preccurlyeq, E, prop, act)$, is one of the following extensions of a labelled poset $(S, \preccurlyeq, prop)$. The conditions must hold for every action a.

1. A *pre*-transition system if $\preccurlyeq \circ E_a \subseteq E_a \circ \preccurlyeq$, and is a saturated *pre*-transition system if $\preccurlyeq \circ E_a \circ \preccurlyeq \subseteq E_a$.
2. A *post*-transition system if $\preccurlyeq \circ E_a^{-1} \subseteq E_a^{-1} \circ \preccurlyeq$, and is a saturated *post*-transition system if $\preccurlyeq \circ E_a^{-1} \circ \preccurlyeq \subseteq E_a^{-1}$.
3. A *upre*-transition system if $\succcurlyeq \circ E_a \subseteq E_a \circ \succcurlyeq$, and is a saturated *upre*-transition system if $\succcurlyeq \circ E_a \circ \succcurlyeq \subseteq E_a$.
4. A *upost*-transition system if $\succcurlyeq \circ E_a^{-1} \subseteq E_a^{-1} \circ \succcurlyeq$, and is a saturated *upost*-transition system if $\succcurlyeq \circ E_a^{-1} \circ \succcurlyeq \subseteq E_a^{-1}$.



$M_1 = (S, \preccurlyeq, E_1, prop)$ $M_2 = (S, \preccurlyeq, E_2, prop)$ $M_3 = (S, \preccurlyeq, E_3, prop)$

Figure 8: Transition systems over posets. M_1 is *pre*- but not *post*-ordered, M_2 is *post*- but not *pre*-ordered and M_3 is neither *pre*- nor *post*-ordered. Note that M_1 and M_2 are saturated.

A *semi-commutation condition* is of the form $\preccurlyeq \circ E \subseteq E \circ \preccurlyeq$. The difference between a *pre*-transition system and a saturated *pre*-transition system is illustrated next. Similar examples exhibit the difference between the other transition systems and their saturated variants.

Example 3.35. The *pre*-transition system M in Figure 7 is not saturated. The state 2 is less than 2, $(2,2)$ is in E and 2 is less than 1, so $(2,1)$ is in $\preccurlyeq \circ E \circ \preccurlyeq$. This pair is not in E. The *pre*-transition system N in Figure 7 includes a transition from 2 to 1 and is saturated. ⌟

An example illustrating the difference between *pre*-transition systems and *post*-transition systems follows.

Example 3.36. Three transition systems over the same poset are shown in Figure 8. Consider the *pre*-transition system conditions. M_1 is a saturated *pre*-transition system. M_2 is not a *pre*-transition system because there is a transition from 1 to 2 but no transition from 2 to a state labelled p. M_3 is not a *pre*-transition system because 2 is less than 1 but 1 has a transition to 3 whereas 2 does not.

Now consider the *post*-transition system conditions. M_1 is not a *post*-transition system because 1 has an incoming transition from 2 but 2 has no incoming transitions. M_2 is a *post*-transition system because the incoming transitions to 1, being empty, are a subset of those incoming to 2. In fact, M_2 is a saturated *post*-transition system. M_3 is not a *post*-transition system because 2 is less than 1 but 1 has incoming transitions and 2 does not.

Consider the *upre*-transition system conditions. M_1 is not a *upre*-transition system because, from the order $1 \preccurlyeq 2$ and transition $(2,1)$, $(1,1)$ is in $\succcurlyeq \circ E_1$. This element cannot be in $E_1 \circ \succcurlyeq$ because 1 has no outgoing transitions. M_2 is a *upre*-transition system because every transition from 2 is a transition from 1. M_3 is not a *upre*-transition system because there is a transition from 2 to 1, so $(1,1)$ is in $\succcurlyeq \circ E_3$. However, $(1,1)$ cannot be in $E_3 \circ \succcurlyeq$ because E_1 contains only one transition from 1 and 3 is incomparable to 1 in the order. ⌟

The different types of monotone transition systems are closely related. If M is a *pre*-transition system, replacing the transition relation with its inverse yields a *post*-transition system. If M is a *pre*-transition system, replacing the order with its dual yields a *upre*-transition system. We have not seen these observations articulated in the literature.

Proposition 3.37. *Each type of monotone transition system satisfies the first equality below and satisfies the second equality if it is saturated.*

pre-transition system	$\preccurlyeq \circ E_a \circ \preccurlyeq = E_a \circ \preccurlyeq$	$\preccurlyeq \circ E_a \preccurlyeq = E_a$
post-transition system	$\preccurlyeq E_a^{-1} \circ \preccurlyeq = E_a^{-1} \circ \preccurlyeq$	$\preccurlyeq \circ E_a^{-1} \circ \preccurlyeq = E_a^{-1}$
upre-transition system	$\succcurlyeq \circ E_a \circ \succcurlyeq = E_a \circ \succcurlyeq$	$\succcurlyeq \circ E_a \circ \succcurlyeq = E_a$
upost-transition system	$\succcurlyeq \circ E_a^{-1} \circ \succcurlyeq = E_a^{-1} \circ \succcurlyeq$	$\succcurlyeq \circ E_a^{-1} \succcurlyeq = E_a^{-1}$

Proof. We prove the statement for *pre*-transition systems.

(*non-saturated*) The relation $\preccurlyeq$ is transitive, so $\preccurlyeq \circ E_a \subseteq E_a \circ \preccurlyeq$ implies that $\preccurlyeq \circ E_a \circ \preccurlyeq$ is contained in $E_a \circ \preccurlyeq$. The converse direction holds because $\preccurlyeq$ is reflexive.

(*saturated*) The reflexivity of $\preccurlyeq$ also entails that E_a is contained in $\preccurlyeq \circ E_a \circ \preccurlyeq$, and the equality for the saturated case follows.

The proof for *post*-transition systems is identical, with E^{-1} in place of E. The only properties of $\preccurlyeq$ required above are reflexivity and transitivity. The proof for universal variants is identical because $\succcurlyeq$, being a partial order, has these properties. ⊣

Corollary 3.38. *A saturated pre-transition system is a saturated upost-transition system and a saturated post-transition system is a saturated upre-transition system.*

For the proof, note that the inverse of the semi-commutation condition for a saturated *pre*-transition system is that of a saturated *upost*-transition system. The transition system M in Example 3.35 is a *pre*-transition system but is not saturated. It is not a *upost*-transition system, so the saturation condition in the corollary above is required.

The structure-preserving maps between ordered transition systems are homomorphisms. A homomorphism of *pre*-transition systems is a labelled poset homomorphism that is a transition system homomorphism. Isomorphism is defined and denoted as usual. Homomorphism and isomorphism for the other variants is as expected, with either order or transition system inversion.

MONOTONE STATE ALGEBRAS Monotone state algebras are perfect distributive lattices with state transformers. In this section, we prove that each type of monotone transition system has an algebraic representation. The representation of *pre*-transition systems exists in the literature but the other results in this section are, to the best of our knowledge, new. The construction used in the representation theorem is illustrated in the example below. Observe that, unlike the Boolean case, the predecessor and successor transformers generate distinct transition relations.

Example 3.39. The *pre*-transition system M in Figure 7 appears again in Figure 9. The poset $(S, \preccurlyeq)$ generates the lattice $\mathscr{D}(S)$ of downsets of states. The predecessor transformer maps a downset X to its preimage under E and the successor transformer maps X to its image under E. The definition of the transformers over join-irreducibles is shown.

The algebra $\mathcal{A}$ generates two transition systems. The join-irreducibles of A, shaded in the figure, are states of the transition system. A transformer defines relations over join-irreducibles. Let x and y be join-irreducibles. The relation E_{pre} contains (x, y) if $x \subseteq pre(y)$ holds and E_{post} contains (x, y) if $y \subseteq post(x)$ holds. The predecessor of $\{1, 2\}$ is $\{2\}$ because $(2, 2)$ is a transition in E and because 1 has no predecessor. Consequently, the relation E_{pre} contains a

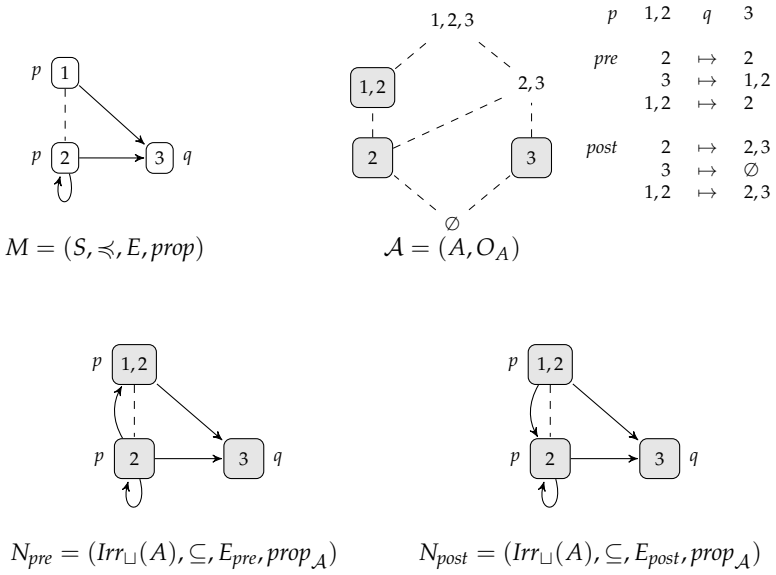
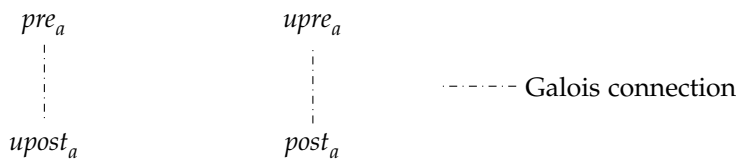


Figure 9: A transition system M that is *pre*- and *post*-ordered. M generates the algebra $\mathcal{A}$ of downsets with predecessor and successor transformers. The algebra generates two monotone transition systems: *pre* generates N_{pre} and *post* generates N_{post}

transition $(\{2\},\{1,2\})$ that is not in M. Similarly, E_{post} contains a transition $(\{1,2\},\{2\})$, which is not in M. Contrary to the situation with Boolean state algebras, *pre* and *post* generate different relations. Moreover, these relations are also different from the relation E used to generate the algebra. ⌟

In a perfect distributive lattice a strict additive function has a unique adjoint. In the absence of negation, De Morgan duals are not used. State transformers in monotone state algebras are related as follows.



Definition 3.40. A *distributive predecessor algebra* $\mathcal{A} = (A, O_A)$ over a predecessor signature is a perfect, distributive lattice A with strict additive transformers pre_a. Distributive successor algebras are similarly defined. Distributive, universal predecessor and successor algebras are defined likewise with the difference that the transformers are strict multiplicative.

The definition of a state algebra does not apply in the distributive setting because De Morgan duals need not exist. We now study algebras generated by monotone transition systems. Our main result is that the constraints on a monotone transition system are necessary and sufficient for the image of a set under a state transformer to be a downset. We also prove that distributive state algebras generate saturated, monotone transition systems.

A monotone transition system $M = (S, \preccurlyeq, E, prop, act)$ defines a lattice of downsets and the transformers below. The definition of transformers is identical to the Boolean transition system case.

$$\begin{aligned}
p^{\mathcal{M}} &\triangleq \{s \mid p \text{ is in } prop(s)\} \\
pre_a^{\mathcal{M}}(X) &\triangleq \{s \mid \text{ for some } t \in X, (s,t) \in E \text{ and } a \in act(s,t)\} \\
post_a^{\mathcal{M}}(X) &\triangleq \{t \mid \text{ for some } s \in X, (s,t) \in E \text{ and } a \in act(s,t)\} \\
upost_a^{\mathcal{M}}(X) &\triangleq \{t \mid \text{ for every } s \in S, (s,t) \in E \text{ implies } t \in X\} \\
upre_a^{\mathcal{M}}(X) &\triangleq \{s \mid \text{ for every } t \in S, (s,t) \in E \text{ implies } t \in X\}
\end{aligned}$$

A transformer is *downset closed* if it maps downsets to downsets. Lemma 3.41 shows that the semi-commutation conditions are necessary and sufficient for downset closure.

Lemma 3.41. *Let $M = (S, \preccurlyeq, E, prop, act)$ be a monotone transition system generating the transformers above.*

1. *M is a pre-transition system exactly if pre_a is downset closed.*
2. *M is a post-transition system exactly if $post_a$ is downset closed.*
3. *M is a upre-transition system exactly if $upre_a$ is downset closed.*
4. *M is a upost-transition system exactly if $upost_a$ is downset closed.*

Proof. We prove the lemma for *pre* and *upre*; the proofs for these two cases are not symmetric. Consider a downset X. For each transformer there are two directions to consider.

(*Existential predecessors*) The condition is $\preccurlyeq \circ E_a \subseteq E_a \circ \preccurlyeq$.

(*Semi-commutation to downsets*) Assume that $\preccurlyeq \circ E_a \subseteq E_a \circ \preccurlyeq$. If s is in $pre_a(X)$, there is a t in X such that (s,t) is in E_a. By the semi-commutation condition, every $s' \preccurlyeq s$ has a corresponding t' such that (s',t') is in E_a and $t' \preccurlyeq t$. Being a downset, X contains t', hence $pre_a(X)$ contains s' and is a downset.

(*Downsets to semi-commutation*) Assume that $pre_a(X)$ is a downset and that X is a principal downset $t{\downarrow}$. Consider (s,t) in E_a, $s' \preccurlyeq s$ and $t \preccurlyeq t'$. By definition, $pre_a(X)$ contains s and being a downset, contains s'. Moreover, by definition of predecessors, there must be a state t' and transition (s',t') with t' in X. However, X is a principal downset, so $t' \preccurlyeq t$. We have shown that $\preccurlyeq \circ E_a \subseteq E_a \circ \preccurlyeq$.

(*Universal predecessors*) The condition is $\succcurlyeq \circ E_a \subseteq E_a \circ \succcurlyeq$.

(*Semi-commutation to downsets*) Assume that $\succcurlyeq \circ E_a \subseteq E_a \circ \succcurlyeq$. If s is in $upre_a(X)$, $E_a(s)$ is contained in X, by definition of $upre_a$. Consider the inequality $s' \preccurlyeq s$. If s' has no successors labelled a, $E_a(s') \subseteq X$, so s' is in X. If s' has a successor t', the semi-commutation condition implies that s has a successor $t \succcurlyeq t'$. Since X is a downset, every such t' is in X, implying $E_a(s') \subseteq X$. Being a downset, X contains t', so $upre_a(X)$ contains s' and is a downset.

(*Downsets to semi-commutation*) Consider elements x, y and z, such that $x \succcurlyeq y$ and (y,z) is in E_a. Furthermore, let X be $E_a(x){\downarrow}$, the smallest downset containing successors of x. Every successor of x is in X, so x is in $upre_a(X)$. By assumption, $upre(X)$ is a downset, so it contains y. Moreover, X contains all successors of X, including z. Since X is the downset of successors of x, there is some successor z' of x such that $z \preccurlyeq z'$. We have shown that $\succcurlyeq \circ E_a$ is contained in $E_a \circ \succcurlyeq$.

The cases for $post_a$ and $upost_a$ are similar. ⊣

The previous lemma allows us to define distributive state algebras from monotone transition systems. The algebra $salg(M)$ defined by a *pre*-transition system is the lattice of downsets with the predecessor transformer. The definitions for the other algebras is similar.

Lemma 3.42. *Each type of monotone transition system defines the corresponding distributive state algebra.*

Proof. Lemma 3.12 shows that the lattice and propositions are a distributive propositional algebra. Lemma 3.41 shows that pre_a is downset closed. Since pre_a is defined elementwise, it is strict additive. ⊣

We shall now generate a monotone transition system from a predecessor algebra. The only difference from the Boolean case is that join-irreducibles are used instead of atoms. A distributive predecessor algebra $\mathcal{A} = (A, O_A)$ with order $\sqsubseteq$ defines a structure $srel(\mathcal{A}) = (S_\mathcal{A}, \preccurlyeq_\mathcal{A}, E_\mathcal{A}, prop_\mathcal{A}, act_\mathcal{A})$ as follows.

$$\begin{aligned} S_\mathcal{A} &\triangleq Irr_\sqcup(A) \\ \preccurlyeq_\mathcal{A} &\triangleq (Irr_\sqcup(A) \times Irr_\sqcup(A)) \cap \sqsubseteq \\ E_\mathcal{A} &\triangleq \{(x,y) \mid x \sqsubseteq pre_a(y) \text{ for some action } a\} \\ prop_\mathcal{A}(x) &\triangleq \left\{p \in Prop \mid x \sqsubseteq p^\mathcal{A}\right\} \\ act_\mathcal{A}(x,y) &\triangleq \{a \in Act \mid x \sqsubseteq pre_a(y)\} \end{aligned}$$

As one may expect, $srel(\mathcal{A})$ is a *pre*-transition system. What is less obvious is that $srel(\mathcal{A})$ is saturated.

Lemma 3.43. *A distributive predecessor algebra $\mathcal{A}$ defines a saturated pre-transition system $srel(\mathcal{A})$.*

Proof. Lemma 3.12 shows that $srel(\mathcal{A})$ contains a labelled poset. We show that $E_\mathcal{A}$ satisfies the saturation condition. Consider a transition (x,y) in $E_\mathcal{A}$ and elements $x' \preccurlyeq x$ and $y \preccurlyeq y'$ in $S_\mathcal{A}$. By definition of *srel*, $x \sqsubseteq pre_a(y)$ holds, from the order, $x' \sqsubseteq pre_a(y)$ and by monotonicity $x' \sqsubseteq pre_a(y')$ whereby (x', y') is in $E_\mathcal{A}$ and is labelled a. As required, E_a contains $\preccurlyeq \circ E_a \circ \preccurlyeq$. ⊣

Lemma 3.43 reveals an asymmetry between *pre*-transition systems and distributive predecessor algebras. Every *pre*-transition system generates a distributive predecessor algebra. However, distributive predecessor algebras only generate saturated *pre*-transition systems.

Theorem 3.44. *A saturated pre-transition system is isomorphic to the one generated by its algebra: $M \cong srel(salg(M))$.*

Proof. Theorem 3.13 shows that $h : s \mapsto s{\downarrow}$ is an isomorphism of labelled posets. We have to show that (s,t) is in E if and only if $(h(s), h(t))$ is in $E_\mathcal{A}$. If (s,t) is in E and has label a, $pre_a(t{\downarrow})$ contains $s{\downarrow}$, so $(s{\downarrow}, t{\downarrow})$, is in $E_\mathcal{A}$ and has label a. Conversely, if $(s{\downarrow}, t{\downarrow})$, is in $E_\mathcal{A}$ and has label a, there is a state s' such that $s \preccurlyeq s'$ and (s', t) is a transition in E labelled a. The saturation condition $E = \preccurlyeq \circ E \circ \preccurlyeq$, implies that (s,t) is a transition in E. ⊣

Theorem 3.45. *A distributive predecessor algebra is isomorphic to the one generated by its transition system: $\mathcal{A} \cong salg(srel(\mathcal{A}))$.*

Proof. Theorem 3.14 established that $h : x \mapsto Irr_\sqcup(x)$ is an isomorphism of distributive propositional algebras. We show that the isomorphism respects the predecessor transformers: $h(pre_a(y)) = pre'_a(h(y))$, where pre'_a is the

transformer in $salg(srel(\mathcal{A}))$. Consider an arbitrary element y and join-irreducible x satisfying $x \sqsubseteq pre_a(y)$. Since pre_a is strict additive, there exists a join-irreducible y' satisfying the condition $x \sqsubseteq pre_a(y')$. For every such y', (x, y') is in $E_{\mathcal{A}}$. Consequently, the isomorphism h maps $pre_a(y)$ to $Irr_{\sqcup}(pre_a(y))$, or equivalently to $pre'_a(Irr_{\sqcup}(y))$, which is the element $pre'_a(h(y))$ as required. ⊣

Conjunctive State Structures

Conjunctive state structures model properties that are independent of negation and disjunction. Conjunctive algebras include the lattices for constant propagation and interval-analysis, and have existed since the origin of lattice-based program analysis [Kildall 1973; Cousot and Cousot 1977]. Relational representations of perfect lattices have been introduced recently in the study of sub-structural logics [Dunn et al. 2005; Gehrke 2006]. Such structures are currently not used in the verification literature.

The technical contribution of this section is in showing that all results proved about monotone transition systems lift to conjunctive transition systems. The representation theorems proved here allow us to generalise notions such as counterexample traces (currently only defined for Boolean structures) to non-Boolean structures.

A conjunctive transition system contains states, a set of elements called *costates*, an order on states and a transition relation over states. In the Boolean case, costates are complements of singleton sets of states, justifying the term. Just as the transition relation in a monotone transition system is constrained by the order, the transition relation in a conjunctive transition system is constrained by the order and the costates. Conjunctive state algebras are defined as before. In the absence of distributivity, a transformer and its adjoint will not generate the same transition system.

CONJUNCTIVE TRANSITION SYSTEMS A conjunctive transition system extends a labelled twoset with a transition relation. Recall that a twoset $M = (S, Q, \preccurlyeq)$ defines a Galois relation G_M between elements of S that have the same overapproximations in Q. A conjunctive transition system differs from a monotone transition system in that all constraints use a Galois relation in place of the order.

Example 3.46. Three different conjunctive transition systems are shown in Figure 10. The set of states, costates and transition relation are identical in each case. The transition systems only differ in the order.

M_1 represents a Boolean transition system. Think of each state s as a set $\{s\}$ A state represents a minimal component of a transition system. A costate represents a maximal component; a set of states that cannot be enlarged without including all states. In the Boolean case, costates represent complements of singleton sets. The states 1 and 2 are both below the same costate, hence they have the same label.

M_2 is a monotone *pre*-transition system. There is an order on both states and costates. Observe that the order on states is isomorphic to the dual of the order on costates. The order on states generates a distributive lattice of downsets. The costates represent the poset of meet-irreducibles.

M_3 is a conjunctive *pre*-transition system. The order on states is not the dual of the order on costates.

Each conjunctive transition system generates a lattice of Galois-stable subsets and a predecessor transformer as shown in the figure. States become

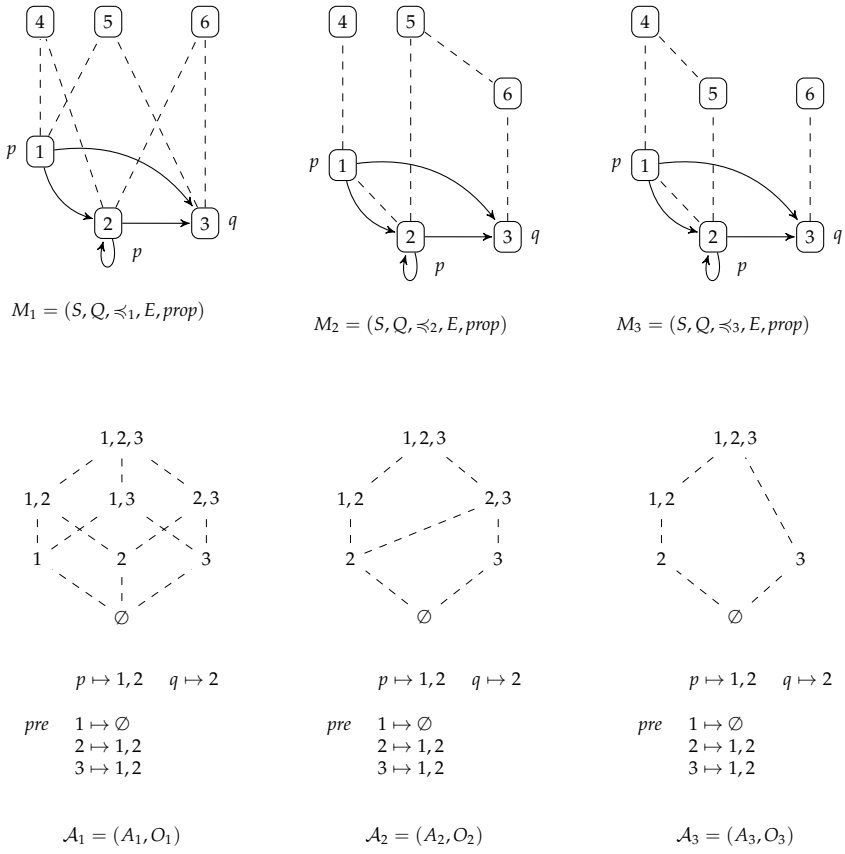


Figure 10: Three conjunctive transition systems and the predecessor algebras they generate. M_1 generates a Boolean predecessor algebra, M_2 a distributive predecessor algebra and M_3 a conjunctive predecessor algebra.

join-irreducibles and costates become meet-irreducibles of the lattice. M_1 generates a Boolean predecessor algebra, M_2 a distributive predecessor algebra and M_3 a conjunctive predecessor algebra. The transformer *pre* is strict additive in every case ⌟

The notions in the example are formalised in the remainder of the section. The definitions that follow are new.

Definition 3.47. A *conjunctive transition system* $M = (S, Q, \preccurlyeq, E, prop, act)$ is one of the following extensions of a labelled twoset $(S, Q, \preccurlyeq, prop)$. The conditions must hold for every action a. Let G be the Galois relation of M.

1. M is a *pre*-transition system if $G \circ E_a \subseteq E_a \circ G$, and is a saturated *pre*-transition system if $G \circ E_a \circ G \subseteq E_a$.
2. M is a *post*-transition system if $G \circ E_a^{-1} \subseteq E_a^{-1} \circ G$, and is a saturated *post*-transition system if $G \circ E_a^{-1} \circ G \subseteq E_a^{-1}$.
3. M is a *upre*-transition system if $G^{-1} \circ E_a \subseteq E_a \circ G^{-1}$, and is a saturated *upre*-transition system if $G^{-1} \circ E_a \circ G^{-1} \subseteq E_a$.
4. M is a *upost*-transition system if $G^{-1} \circ E_a^{-1} \subseteq E_a^{-1} \circ G^{-1}$, and is a saturated *upost*-transition system if $G^{-1} \circ E_a^{-1} \circ G^{-1} \subseteq E_a^{-1}$.

As in the monotone case, the semi-commutation conditions on different types of transition systems imply equalities.

Proposition 3.48. *Each type of conjunctive transition system below satisfies the first equality and satisfies the second equality if it is saturated.*

pre	$G \circ E_a \circ G = E_a \circ G$	$G \circ E_a G = E_a$
post	$GE_a^{-1} \circ G = E_a^{-1} \circ G$	$G \circ E_a^{-1} \circ G = E_a^{-1}$
upre	$G^{-1} \circ E_a \circ G^{-1} = E_a \circ G^{-1}$	$G^{-1} \circ E_a \circ G^{-1} = E_a$
upost	$G^{-1} \circ E_a^{-1} \circ G^{-1} = E_a^{-1} \circ G^{-1}$	$G^{-1} \circ E_a^{-1} G^{-1} = E_a^{-1}$

The proofs are similar to those for monotone transition systems. The only properties required for Proposition 3.37 were the semi-commutation condition and reflexivity and transitivity of the order. All these properties still apply. The difference between saturated and unsaturated conjunctive transition systems is similar to the monotone case.

CONJUNCTIVE STATE ALGEBRAS The algebraic structure corresponding to a conjunctive transition system is a conjunctive state algebra. Conjunctive state algebras are perfect lattices with transformers. Each transformer must be strict additive or strict multiplicative. There are four different types of conjunctive state algebras, each equipped with predecessor or successor algebras and their universal counterparts.

Definition 3.49. A *conjunctive predecessor algebra* $\mathcal{A} = (A, O_A)$ over a predecessor signature is a perfect lattice A with strict additive transformers pre_a. *Conjunctive successor algebras* are similarly defined. *Conjunctive universal predecessor* and *conjunctive universal successor* algebras are defined likewise with the difference that the transformers are strict multiplicative.

We now study the algebras generated by conjunctive transition systems. When translating a transition system to an algebra Galois stable subsets are used instead of powersets or downsets. When moving from algebras to transition systems both join and meet-irreducibles are used.

A conjunctive transition system $M = (S, Q, \preccurlyeq, E, prop, act)$ defines a lattice of Galois-closed subsets and the transformers below.

$$
\begin{aligned}
p^{\mathcal{M}} &\triangleq \{s \mid p \text{ is in } prop(s)\} \\
pre_a^{\mathcal{M}}(X) &\triangleq \{s \mid \text{ for some } t \in X, (s,t) \in E \text{ and } a \in act(s,t)\} \\
post_a^{\mathcal{M}}(X) &\triangleq \{t \mid \text{ for some } s \in X, (s,t) \in E \text{ and } a \in act(s,t)\} \\
upost_a^{\mathcal{M}}(X) &\triangleq \{t \mid \text{ for every } s \in S, (s,t) \in E \text{ implies } t \in X\} \\
upre_a^{\mathcal{M}}(X) &\triangleq \{s \mid \text{ for every } t \in S, (s,t) \in E \text{ implies } t \in X\}
\end{aligned}
$$

The definitions of the transformers are identical to the Boolean transition system case. A transformer is *Galois-stable* if it maps a Galois-stable set to a Galois-stable set. Lemma 3.50 shows that the semi-commutation conditions are necessary and sufficient for transformers to be Galois-stable.

Lemma 3.50. *Let $M = (S, Q, \preccurlyeq, E, prop, act)$ be a conjunctive transition system generating the transformers above.*

1. *M is a pre-transition system exactly if pre_a is Galois stable.*
2. *M is a post-transition system exactly if $post_a$ is Galois stable.*
3. *M is a upre-transition system exactly if $upre_a$ is Galois stable.*
4. *M is a upost-transition system exactly if $upost_a$ is Galois stable.*

Proof. Consider a Galois-stable set $X \subseteq S$. For each transformer there are two directions to consider.

(*Existential predecessors*) The condition is $G_M \circ E_a \subseteq E_a \circ G_M$.

(*Semi-commutation to Galois-stability*) Assume the semi-commutation condition. If s is in $pre_a(X)$, there is a state t in X and a transition (s,t) in E_a. Semi-commutation implies that for every (s',s) in G_M, there is a state t' and a transition (s',t') in E_a with (t',t) in G_M. The set X is Galois-stable and contains t'. It follows that $pre_a(X)$ contains s' and is Galois-stable.

(*Galois-stability to semi-commutation*) Assume that $pre_a(X)$ is Galois-stable and X is of the form $G_M^{-1}(t)$, where t is a state. Consider a transition (s,t) in E_a and a pair (s',s) in G_M. By definition, $pre_a(X)$ contains s and, being Galois-stable, contains s'. Moreover, by definition of predecessors, there must be a state t' and transition (s',t') with t' in X. However, X is the preimage of t under G_M, so t' must be in this preimage. We have shown that $G_M \circ E_a \subseteq E_a \circ G_M$.

(*Universal predecessors*) The condition is $G_M^{-1} \circ E_a \subseteq E_a \circ G_M^{-1}$.

(*Semi-commutation to Galois-stability*) Assume the semi-commutation condition. If s is in $upre_a(X)$, $E_a(s)$ is contained in X, by definition of $upre_a$. Consider a pair (s',s) in G_M^{-1}. If s' has no successors labelled a, $E_a(s') \subseteq X$, so s' is in X. If s' has a successor t', the semi-commutation condition implies that s has a successor t such that (t',t) is in G_M. Since X is Galois-stable, every such t' is in X, implying that $E_a(s')$ is contained in X. Thus, $upre_a(X)$ contains s' and is Galois-stable.

(*Galois-stability to semi-commutation*) Consider elements x, y and z, such that (y,x) is in G_M and (y,z) is a transition. Furthermore, let X be the preimage $G_M^{-1}(E_a(x))$ of the successors of x. Every successor of x is in X, so x is in $upre_a(X)$. By assumption, $upre(X)$ is Galois-stable, so it contains y. Moreover, X contains all successors of X, including z. Since X is Galois-stable there is some successor z' of x such that (z,z') is in G_M. We have shown that $G_M^{-1} \circ E_a$ is contained in $E_a \circ G_M^{-1}$.

The cases for $post_a$ and $upost_a$ are similar. ⊣

We shall now generate a conjunctive transition system from a conjunctive predecessor algebra. The difference with the monotone case is that both join- and meet-irreducibles are both used. A conjunctive predecessor algebra $\mathcal{A} = (A, O_A)$ with order $\sqsubseteq$ defines a structure $srel(\mathcal{A}) = (S_\mathcal{A}, Q_\mathcal{A}, \preccurlyeq_\mathcal{A}, E_\mathcal{A}, prop_\mathcal{A}, act_\mathcal{A})$ as follows.

$$
\begin{aligned}
S_\mathcal{A} &\triangleq Irr_\sqcup(A)\\
Q_\mathcal{A} &\triangleq Irr_\sqcap(A)\\
\preccurlyeq_\mathcal{A} &\triangleq (Irr_\sqcup(A) \cup Irr_\sqcap(A))^2 \cap \sqsubseteq\\
E_\mathcal{A} &\triangleq \{(x,y) \mid x \sqsubseteq pre_a(y) \text{ for some action } a\}\\
prop_\mathcal{A}(x) &\triangleq \left\{p \in Prop \mid x \sqsubseteq p^\mathcal{A}\right\}\\
act_\mathcal{A}(x,y) &\triangleq \{a \in Act \mid x \sqsubseteq pre_a(y)\}
\end{aligned}
$$

Similar to the construction in the monotone case, $srel(\mathcal{A})$ is a saturated conjunctive *pre*-transition system.

Lemma 3.51. *A conjunctive predecessor algebra $\mathcal{A}$ defines a saturated, conjunctive pre-transition system $srel(\mathcal{A})$.*

Proof. Lemma 3.20 shows that the structure $(S, Q, \preccurlyeq, prop)$ generated by $\mathcal{A}$ is a labelled twoset. It remains to be shown that the transition relation satisfies the semi-commutation conditions. Let G be the Galois relation of $srel(\mathcal{A})$.

Consider pairs (x', x) and (y, y') in G_M and a transition (x, y) in $E_{\mathcal{A}}$, where all elements are join-irreducibles. The pair (x', y') is in $G_M \circ E_{\mathcal{A}} \circ G_M$ and we have to show that it is in $E_{\mathcal{A}}$. The element y' being join-irreducible is the greatest element of $\ell(u(y'))$. The element y' being join-irreducible is the greatest element of $\ell(u(y'))$. Since y is in $G_M^{-1}(y')$, it must be that $y \sqsubseteq y'$. By the same argument, $x' \sqsubseteq x$ holds. By construction, $x \sqsubseteq pre(\ell(u(y)))$ and because *pre* is monotone, $x' \sqsubseteq pre(\ell(u(y')))$ also holds, showing that (x', y') is in $E_{\mathcal{A}}$. ⊣

Theorem 3.52. *A saturated, conjunctive pre-transition system is isomorphic to the one generated by its algebra: $M \cong srel(salg(M))$.*

Proof. Let M be a conjunctive *pre*-transition system with Galois relation G_M. The proof of Theorem 3.22 shows that the map $h : x \mapsto G_M^{-1}(x)$ is a labelled twoset isomorphism. The condition on transitions has to be verified. If (x, y) is a transition in M, the generated algebra satisfies the inequality $\ell(u(x)) \sqsubseteq pre(\ell(u(y)))$ and consequently satisfies that $(\ell(u(x)), \ell(u(y)))$ is a transition in $srel(salg(M))$. We have shown that $(h(x), h(y))$ is a transition. The converse direction is the same. ⊣

Theorem 3.53. *A conjunctive predecessor algebra is isomorphic to the one generated by its transition system: $\mathcal{A} \cong salg(srel(\mathcal{A}))$.*

Proof. The proof of Theorem 3.21 shows that $h : x \mapsto Irr_{\sqcup}(x)$ is an isomorphism of conjunctive propositional algebras. We show that this isomorphism commutes with the predecessor transformer. Consider x and y such that $x \sqsubseteq pre_a(y)$. The transformer pre_a is strict additive, so $pre_a(y) = pre_a(\bigsqcup Irr_{\sqcup}(y))$ and the equality $pre_a(y) = \bigsqcup pre_a(Irr_{\sqcup}(y))$ holds. For every element w in $Irr_{\sqcup}(x)$ there exists an element z in $Irr_{\sqcup}(y)$ such that $w \sqsubseteq pre_a(z)$. Thus, there is a transition (w, z) in $srel(\mathcal{A})$ and the generated transformer $pre_{\mathcal{M}}$ satisfies $w \sqsubseteq pre_{\mathcal{M}}(z)$. By taking the join of join-irreducible sets we obtain $h(x) \sqsubseteq pre_{\mathcal{M}}(h(y))$ and, by a similar argument, the converse inequality. Substituting x with $pre_a(y)$ yields the desired result. ⊣

The Abstraction Perspective

We extend the abstract interpretation characterisation of propositional, distributive and conjunctive algebras to the corresponding state algebras. Every conjunctive predecessor algebra arises as an abstraction of a distributive predecessor algebra and every distributive predecessor algebra arises as an abstraction of a state algebra. To make this correspondence precise, we have to extend Theorem 3.23 to show that predecessor transformers are also preserved by abstraction.

Consider a conjunctive predecessor algebra $\mathcal{C} = (C, O_C)$. The algebra $\mathcal{C}$ defines the algebras $\mathcal{D} = (D, O_D)$ and $\mathcal{B} = (B, O_B)$ below, which are over a predecessor signature.

$$
\begin{array}{ll}
D \mathrel{\hat{=}} \mathscr{D}(Irr_{\sqcup}(C), \preccurlyeq) & B \mathrel{\hat{=}} \mathscr{P}(Irr_{\sqcup}(C)) \\
\preccurlyeq \mathrel{\hat{=}} (Irr_{\sqcup}(C) \times Irr_{\sqcup}(C)) \cap \sqsubseteq & \\
p^{\mathcal{D}} \mathrel{\hat{=}} Irr_{\sqcup}(p^{\mathcal{C}}) & p^{\mathcal{B}} \mathrel{\hat{=}} Irr_{\sqcup}(p^{\mathcal{C}}) \\
pre_a^{\mathcal{D}} \mathrel{\hat{=}} Y \mapsto Irr_{\sqcup}(pre_a^{\mathcal{C}}(\bigsqcup Y)) & pre_a^{\mathcal{B}} \mathrel{\hat{=}} Y \mapsto Irr_{\sqcup}(pre_a^{\mathcal{C}}(\bigsqcup Y))
\end{array}
$$

Since the set of join irreducibles below an element is a downset, both $pre_a^{\mathcal{D}}$ and $pre_a^{\mathcal{B}}$ above are well defined. Recall the two closure operators below, which were used to relate the algebras above without predecessor transformers.

$$\textit{b-to-d} : B \to B \qquad\qquad \textit{d-to-c} : D \to D$$
$$\textit{b-to-d} \mathrel{\hat{=}} X \mapsto X{\downarrow} \qquad\qquad \textit{d-to-c} \mathrel{\hat{=}} Y \mapsto \ell(u(Y))$$

Theorem 3.54. *Let $\mathcal{C}$ be a conjunctive predecessor algebra defining the distributive predecessor algebras $\mathcal{D}$ and predecessor algebra $\mathcal{B}$ as above.*

1. *The upper closure b-to-d satisfies that $pre_a^{\mathcal{B}} \circ \textit{b-to-d} = \textit{b-to-d} \circ pre_a^{\mathcal{D}} \circ \textit{b-to-d}$.*
2. *The upper closure d-to-c satisfies that $pre_a^{\mathcal{D}} \circ \textit{d-to-c} = \textit{d-to-c} \circ pre_a^{\mathcal{C}} \circ \textit{d-to-c}$.*

The statements above show that the predecessor transformers in a distributive predecessor algebra are *γ-complete abstractions* of the corresponding transformers on predecessor algebras. Moreover, predecessor transformers on conjunctive predecessor algebras are *γ-complete abstractions* of the corresponding transformers on predecessor algebras. The equalities can be interpreted in terms of the semi-commutation conditions imposed on monotone and conjunctive transition systems.

The representation theorem for distributive predecessor algebras shows that the corresponding transition system must satisfy the condition on the left below, which when read from right to left can be rewritten as the algebraic condition on the right.

$$\preccurlyeq \circ E_a \subseteq E_a \circ \preccurlyeq \qquad\qquad \textit{b-to-d} \circ pre_a \subseteq pre_a \circ \textit{b-to-d}$$

The second condition asserts that the downwards closure of the predecessors of a set of states S should be contained in the predecessors of the downwards closure of S. Downwards closure is an upper closure on the powerset lattice B, so applying it on both sides of the set inequality above, combined with the extensivity of closure operators, yields the equality below.

$$pre_a^{\mathcal{B}} \circ \textit{b-to-d} = \textit{b-to-d} \circ pre_a^{\mathcal{B}} \circ \textit{b-to-d}$$

Combining this identity above with standard substitution arguments yields the equality in Theorem 3.54. Conjunctive predecessor algebras have representations that satisfy a similar semi-commutation condition to distributive predecessor algebras. The argument for showing that conjunctive predecessor algebras are γ-complete abstractions is similar.

SECTION SUMMARY This section recalled transition systems and their algebraic counterpart, state algebras. We recalled extensions of transition systems to monotone and conjunctive transition systems and presented algebraic representations of monotone transition systems as distributive predecessor algebras, and of conjunctive transition systems as conjunctive predecessor algebras. The contribution of this section was to distinguish between different kinds of monotone transition systems and to show that only saturated structures arise as representations of distributive predecessor and successor algebras.

3.4 TRACE STRUCTURES

State structures capture the behaviour of a system at the level of transitions. A state can have multiple successors, expressing that a system has distinct future behaviours. Consequently, state structures are *branching-time* models.

An alternative model is to view a system as a collection of executions, called traces. The trace-based model is called *linear-time* because a state on a trace has a unique successor. Applications that require both linear-time and branching-time information need models that represent both behaviours.

Emerson and Halpern [1985] combined branching-time and linear-time behaviour by augmenting a transition system with a set of traces. This model was used to give semantics to the logic CTL* and is standard in the literature. However, the distinction between states and traces complicates an algebraic treatment. Cousot and Cousot [2000] showed that, by lifting state transformers to sets of traces (as opposed to single traces), one can model linear-time and branching-time behaviour with a structure that contains only traces without explicitly referring to states. Their insight is the basis for the model presented in this section.

Algebras for transition systems have been researched extensively due to their connection to modal logic and programming language semantics. In contrast, algebras over traces have received less attention. One reason is that algebras are typically derived from the axioms of a logic. Boolean and monotone state algebras satisfy the axioms of modal logic. The axioms of logics over traces are intimidating beasts: Manna and Pnueli [1992] give seventeen axioms for LTL, Emerson and Halpern [1985] identify ten axioms for CTL and Reynolds [2001] suggested twenty three axioms for CTL*, all in addition to the axioms of propositional logic. It is unwieldy to work with an algebra whose definition involves so many properties.

Two issues complicate an algebraic treatment of logics over states and traces. First, the standard model of time has a finite past and infinite future. An algebra for such a model must distinguish between a time instant that has no history and all other time instants. The second complication arises because some transformers apply to states and others to traces. The corresponding algebras are two-sorted and conditions regulating the interaction of state and trace transformers. Möller et al. [2006] recently suggested a two-sorted, algebraic model based on quantales.

The contribution of this section is a new model, a *trace algebra*, that combines two existing insights to avoid these complications. The treatment of time is simplified if the future and past are both infinite, as suggested independently by Cousot and Cousot [2000] and von Karger [2002]. The distinction between states and traces is avoided by lifting state transformers to traces as shown by Cousot and Cousot [2000]. The main result of this section is that trace algebras have representations as sets of traces.

TRACES We define a new notion of a trace. Unlike the standard notion, our traces have an infinite past. The operators we define enable navigating along a trace just as predecessor and successor transformers enable exploring a transition system.

A transition system is *two-way total* if every state has at least one predecessor and successor. Fix a two-way total transition system $M = (S, E, prop, act)$. A sequence of states $\bar{s}$ *respects* E if consecutive elements of $\bar{s}$ are in E. A *trace* is a two-way sequence $\pi : \mathbb{Z} \to S$ that respects E and $traces(E)$ contains all traces that respect E. The set of all two-way sequences over S is $traces(S)$ (which abbreviates $traces(S \times S)$).

Fix a trace π. The *future* of π is the set of mappings $\{i \mapsto \pi_i \mid i \geq 0\}$ and the *history* of π is similarly defined for indices $i \leq 0$. The *origin* of a trace is the state $\pi(0)$, also written π_0 The set of traces with origin s is

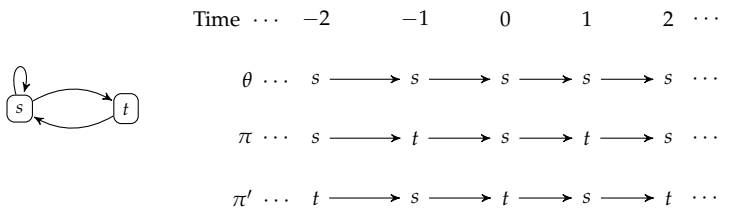


Figure 11: A trace system consisting of a transition system and a closed set of traces

traces(s). A step function advances a trace by k steps. Formally, the function *step* : *traces*$(S) \times \mathbb{Z} \to$ *traces*(S)

maps a trace π and index k to the trace $\{i \mapsto \pi(i+k) | i \in \mathbb{Z}\}$.

A set of traces T is *forward closed* if

step(π, k) is in T for every trace π in T and $k \geq 0$.

Backward closure is similarly defined for indices $k \leq 0$. A *closed* set of traces is forward and backward closed. Trace systems extend transition systems with sets of traces.

Example 3.55. A two-way total transition system and three traces θ, π and π' are shown in Figure 11. The set of traces $T = \{\pi, \pi', \theta\}$ is closed. The set T is a strict subset of the traces of M. A set of closed traces obtained from a transition system is called a trace system. ⌟

Definition 3.56. A *trace system* $N = (S, E, T, prop, act)$ is a transition system with a closed set of traces T.

TRACE ALGEBRAS The algebraic analogues of trace systems are introduced next. A *trace signature*

$$Prop \cup Act \cup \{next, prev, some\}$$

contains symbols for propositions, actions and three *trace transformers*. The transformers *next* and *prev* move a trace forward or backward, respectively. The transformer *some* maps a set of traces to all traces with the same origin. A reader familiar with the logic CTL* may think of *some* as modelling the existential modality E.

Example 3.57. A trace algebra $\mathcal{A} = (A, O_A)$ is shown in Figure 12 with the definition of *next* and *some*. Each element of the lattice represents a set of traces. The trace transformer *next* maps a set of traces to those obtained by advancing each trace by one step. This transformer neither adds to nor depletes a set of traces. Formally, it is strict additive and strict multiplicative. The transformer *some* maps a set of traces to those traces that have the same origin. In Figure 12, *some* maps 2 and 3 to 6, indicating that 2 and 3 represent sets of traces that have the same origin. The image of the lattice under *some* is exactly the powerset lattice of states.

The algebra $\mathcal{A}$ defines a trace system N. A state of N is an element *some*(x), where x is an atom of A. There is a transition from a state *some*(x) to a state *some*(y) if *some*$($*next*$(x)) =$ *some*(y). The traces of N can be constructed by iteratively applying *next* and *prev* to the states of N. The state at index i of a

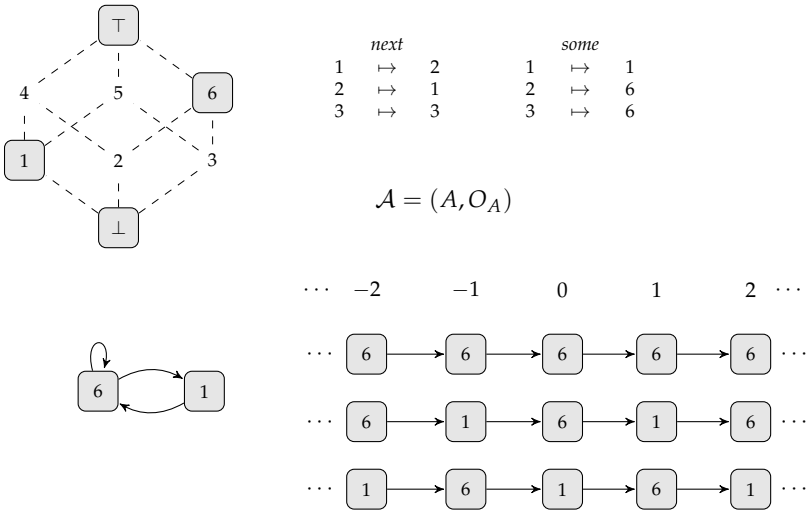


Figure 12: A trace algebra $\mathcal{A}$ with the definition of *next* and *some*. The image of A under *some* is shown by shaded elements. The algebra generates a transition system over the shaded elements and a closed set of traces

trace π with origin $some(x)$ is $some(next^i(x))$. Proceeding in this manner, we obtain a trace system identical to the one in Figure 11. ⌟

The formal definition of a trace algebra follows. This definition and all theorems that follow are new. The construction of a trace system from a trace algebra is straightforward, but proving a representation theorem is challenging because three trace transformers are required to define a single trace. We require several properties of these transformers to ensure that a trace algebra defines a trace system.

Definition 3.58. A *trace algebra* $\mathcal{A} = (A, O_A)$ over a trace signature satisfies the conditions below.

1. The domain A is a perfect Boolean algebra.
2. The transformer *some* is strict additive, satisfies $some(some(x)) \sqsubseteq some(x)$ for all x, and $x \sqsubseteq some(x)$ for all $x \sqsubseteq some(\top)$.
3. The transformers *next* and *prev* are conjugate and dual-conjugate.
4. For every proposition $p^{\mathcal{A}}$, $x \sqsubseteq p^{\mathcal{A}}$ if and only if $some(x) \sqsubseteq p^{\mathcal{A}}$.
5. For all atoms x and y satisfying the equalities $some(x) = some(y)$ and $some(next(x)) = some(next(y))$, $x \sqsubseteq a^{\mathcal{A}}$ if and only if, $y \sqsubseteq a^{\mathcal{A}}$, for every action a.
6. For every x, the transformers satisfy that, if $some(x)$ is not $\bot$, neither are $some(next^i(x))$ and $some(prev^i(x))$ for all i in $\mathbb{N}$.

The remainder of the section establishes a series of properties required for the representation theorem.

PROPERTIES OF *next* AND *prev* The conjugacy and dual-conjugacy conditions on *next* and *prev* imply that the transformers preserve all meets and joins. Furthermore, one transformer is the inverse of the other. Contrast this situation with the state algebra case, where *pre* and *post* are not inverses.

Lemma 3.59. *The transformer next is the inverse of prev.*

Proof. It suffices to show the equality $next(prev(x)) = x$. The inequality below is the Dedekind law for conjugate functions in Theorem 2.24.

$$\begin{aligned}
& next(x) \sqcap y \sqsubseteq next(x \sqcap prev(y)), \\
\Longrightarrow\ & next(\top) \sqcap y \sqsubseteq next(\top \sqcap prev(y)), \text{ by setting } x \text{ to } \top \\
\Longleftrightarrow\ & y \sqsubseteq next(prev(y)), \text{ because } next \text{ is } \top\text{-strict.}
\end{aligned}$$

Dual conjugate functions satisfy the dual Dedekind law, so the inequality $next(prev(y)) \sqsubseteq y$ follows by a dual sequence of arguments. ⊣

The inverse property leads to several others.

Corollary 3.60. *The transformer next is the left and right adjoint of prev.*

Proof. The inequality $next(x) \sqsubseteq y$ implies $prev(next(x)) \sqsubseteq prev(y)$, so $x \sqsubseteq prev(y)$. The other directions are similar. ⊣

Lemma 3.61. *The transformers next and prev are perfect Boolean algebra isomorphisms.*

Proof. A perfect Boolean algebra isomorphism is strict additive, strict multiplicative and commutes with complementation. We show that *next* has each of these properties. The same arguments apply to *prev*.

($\bigsqcup, \bigsqcap$) The transformer *next* and *prev* are conjugate, implying strict additivity and dual-conjugate, implying strict multiplicativity.

($\neg$) We prove that the equality $next(\neg x) = \neg\, next(x)$. The first inequality below is due to Jónsson and Tarski (Theorem 2.24).

$$\begin{aligned}
& prev(x \sqcap \neg\, next(y)) \sqsubseteq prev(x) \sqcap \neg y \\
\Longrightarrow\ & prev(\neg\, next(y)) \sqsubseteq prev(\top) \sqcap \neg y, \text{ by setting } x \text{ to } \top \\
\Longleftrightarrow\ & prev(\neg\, next(y)) \sqsubseteq \neg y, \text{ because } prev \text{ is } \top\text{-strict} \\
\Longleftrightarrow\ & next(prev(\neg\, next(y))) \sqsubseteq next(\neg y), \text{ because } next \text{ is monotone} \\
\Longleftrightarrow\ & \neg\, next(y) \sqsubseteq next(\neg y), \text{ as } prev \circ next \text{ is identity.}
\end{aligned}$$

The dual arguments yield that $\neg\, next(y) \sqsupseteq next(\neg y)$.
Lemma 3.59 shows that *next* has an inverse *prev*, so it is bijective, hence is an isomorphism. ⊣

The property $\neg\, next(x) = next(\neg x)$ shows that *next* is its own De Morgan dual. The corollary below is useful later.

Corollary 3.62. *The transformer next maps atoms to atoms.*

Proof. Consider an atom x. Since $next(x)$ is a Boolean algebra isomorphism, $next(x) \neq \bot$ because x is not $\bot$. We show that there is no element strictly between $next(x)$ and $\bot$. Consider an element y.

$$\begin{aligned}
& y \sqsubseteq next(x) \\
\Longleftrightarrow\ & y \sqcap \neg\, next(x) = \bot \\
\Longleftrightarrow\ & y \sqcap next(\neg x) = \bot, \text{ because } next \text{ is self-dual} \\
\Longleftrightarrow\ & prev(y) \sqcap \neg x = \bot, \text{ because } prev \text{ and } next \text{ are conjugate} \\
\Longleftrightarrow\ & prev(y) = \bot \text{ or } prev(y) = x, \text{ because } x \text{ is an atom} \\
\Longleftrightarrow\ & y = \bot \text{ or } y = next(x)
\end{aligned}$$

⊣

PROPERTIES OF *some* The transformer *some* differs from *next* and *prev* in its properties. The properties of $some(A)$ are required for the representation theorem of trace algebras.

Proposition 3.63. *The transformer some is idempotent.*

Proof. Strict additivity implies that *some* is monotone, so $some(x) \sqsubseteq some(\top)$ for all x. In turn, $some(x) \sqsubseteq some(some(x))$ by the definition of *some*. The reverse inequality holds by definition. ⊣

Lemma 3.64. *The image of the domain of a trace algebra under the transformer some is a perfect Boolean algebra.*

Proof. Consider a trace algebra $\mathcal{A} = (A, O_A)$ with a domain $(A, \sqsubseteq, \sqcap, \sqcup, \neg)$. Let the image domain be $(some(A), \preccurlyeq, \curlywedge, \curlyvee, \sim)$, where the order, meet and join are those of A and negation $\sim x$ is $\neg x \sqcap some(\top)$. We show that the definition above yields transformers on $some(A)$ and the properties of a perfect Boolean algebra hold. The top and bottom elements of $some(A)$ are $some(\top)$ and $\bot$.

($\curlyvee$) Since *some* is strict additive, $some(A)$ is closed under joins of A.

($\curlywedge$) By monotonicity, $some(some(x) \sqcap some(y)) \sqsubseteq some(x) \sqcap some(y)$ holds. Conversely, since *some* is extensive on elements below $some(\top)$, $some(x) \sqcap some(y) \sqsubseteq some(some(x) \sqcap some(y))$ holds. It follows that $some(A)$ is closed under $\sqcap$.

($\sim$) The complement properties have to be shown. By definition of $\sim$, is $\sim some(x) \sqcap some(x)$ is $\neg\, some(x) \sqcap some(\top) \sqcap some(x)$, which is $\bot$. Dually, $\sim some(x) \sqcup some(x)$ is $(\neg\, some(x) \sqcap some(\top)) \sqcup some(x)$, which, by distributivity in A, is $some(\top) \sqcup some(x)$, and by monotonicity, we have $some(\top)$.

(*Complete Boolean algebra*) The lattice is distributive because A is. The lattice is complemented, hence a Boolean algebra. Arbitrary joins exist because $\bigvee some(X) = some(\bigsqcup X)$, implying completeness.

(*Atomic*) The atoms of $some(A)$ are the elements $some(x)$, where x is an atom of A and $some(x)$ is not $\bot$. To see this, consider y such that $\bot \preccurlyeq some(y) \preccurlyeq some(x)$. We show that $some(y) = some(x)$. The inequality $some(y) \preccurlyeq some(x)$ is equivalent to $some(y) \sqsubseteq some(x)$ and furthermore to the equality $some(y) \sqcap \neg\, some(x) = \bot$. Constraining further, $some(y) \sqcap \neg\, some(x) \sqcap some(\top) = \bot$ which is equivalent to $some(y) \sqcap \sim some(x) = \bot$ by the definition of complement. The other direction is similar. ⊣

A study of the representation of trace algebras follows. Recall that $traces(S)$ is the set of all traces over a set S. A trace system $N = (S, E, T, prop, act)$ generates an algebra $talg(N) = (\mathscr{P}(\mathbb{Z} \to S), O_N)$, also denoted $\mathcal{N}$, with the transformers below.

$$
\begin{aligned}
p^{\mathcal{N}} &\triangleq \{\pi \mid p \text{ is in } prop(\pi_0)\} \\
a^{\mathcal{N}} &\triangleq \{\pi \mid a \text{ is in } act(\pi_0, \pi_1)\} \\
next(X) &\triangleq \{step(\pi, 1) \mid \pi \text{ is in } X\} \\
prev(X) &\triangleq \{step(\pi, -1) \mid \pi \text{ is in } X\} \\
some(X) &\triangleq \{\pi \in T \mid \text{ there exists } \theta \in X \text{ such that } \theta_0 = \pi_0\}
\end{aligned}
$$

The set $p^{\mathcal{N}}$ contains traces with an origin labelled p and $a^{\mathcal{N}}$ contains traces whose first transition is labelled a. The transformer *next* advances a trace by one step, *prev* rewinds a trace by one step and *some* maps a trace to all traces in T with the same origin.

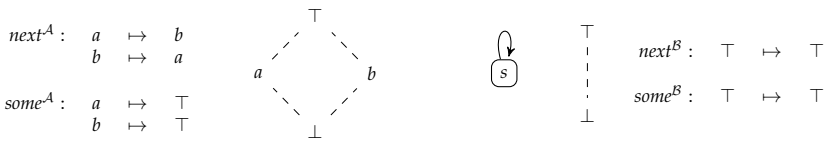


Figure 13: Two trace algebras $\mathcal{A}$ and $\mathcal{B}$ that generate isomorphic trace systems

Lemma 3.65. *The algebra generated by a trace system is a trace algebra.*

Proof. Consider a trace system $N = (S, E, T, prop, act)$ and the algebra $talg(N)$. We show that the conditions of a trace algebra are satisfied. Let X and Y be sets of traces.

(*some*) Consider a binary relation R over $traces(S)$ defined as follows:

$$\{(\theta, \pi) \mid \theta_0 = \pi_0, \theta \in traces(S), \pi \in T\}$$

The transformer *some* maps a set of traces X to $R(X)$. It follows from the Jónsson and Tarski theorem that *some* is strict additive. Since R is transitive, $R(X) = R(R(X))$, so *some* is idempotent. Observe that $R(traces(S))$ is the set of traces T of N and for $X \subseteq T$, every π in X is also in $R(X)$, so *some* is extensive on such T, as required.

(*next*, *prev*) We show that *next* and *prev* are conjugate and dual conjugate. Observe that $next(X) \cap Y \neq \emptyset$ exactly if $step(\pi, -1)$ is in X for some trace π in Y, which is, in turn, equivalent to the condition $X \cap prev(Y) \neq \emptyset$. The two transformers are dual conjugate because $next(X) \cup Y = \emptyset$ exactly if $step(\pi, -1)$ is not in X for every π in Y, exactly if $X \cup prev(Y) = \emptyset$.

(*Propositions*) If $X \subseteq p^{\mathcal{N}}$, p is in $prop(\pi_0)$ for every π in X. For every trace $\theta \in some(X)$ there is a trace $\pi \in X$ such that $\theta_0 = \pi_0$. Consequently, $prop(\theta_0) = prop(\pi_0)$ and $some(X) \subseteq p^{\mathcal{N}}$.

(*Actions*) An atom of $\mathscr{P}(T)$ is a single trace. The conditions $some(\theta) = some(\pi)$ and $some(next(\theta)) = some(next(\pi))$ for traces θ and π imply that $\theta_0 = \pi_0$ and $\theta_1 = \pi_1$, wherefore $act(\theta_0, \theta_1) = act(\pi_0, \pi_1)$. It follows that $\{\pi\} \subseteq a^{\mathcal{N}}$ if and only if $\{\theta\} \subseteq a^{\mathcal{N}}$. ⊣

Every trace system generates a trace algebra. However, all trace algebras cannot be generated by trace systems. In the remainder of the section, we identify the class of trace algebras with trace system representations. The construction of a trace system from a trace algebra is more involved than that of a transition system from a state algebra. This is because traces are defined as sequences of states, so a transition system has to be extracted from a trace algebra and traces must be defined over this transition system.

Example 3.66. Two trace algebras $\mathcal{A}$ and $\mathcal{B}$ are defined in Figure 13. The definition of the transformers is given only for atoms. The image of both algebras under *some* is the two element Boolean algebra. The corresponding transition system has one state and exactly one trace. The origin of the trace is $some(a)$ and the i-th state on the trace is $some(next^i(a))$. Observe that the generated trace systems $trel(\mathcal{A})$ and $trel(\mathcal{B})$ are isomorphic. ⌟

A trace algebra $\mathcal{A} = (A, O_A)$ defines a trace system given below. The construction uses a function $\Pi_{\mathcal{A}} : A \to (\mathbb{Z} \to \mathit{some}(A))$ that maps an element of A to a trace over $\mathit{some}(A)$. This function is used in proofs.

$$
\begin{aligned}
\mathit{trel}(\mathcal{A}) &= (S_{\mathcal{A}}, E_{\mathcal{A}}, T_{\mathcal{A}}, \mathit{prop}_{\mathcal{A}}, \mathit{act}_{\mathcal{A}}) \\
S_{\mathcal{A}} &\triangleq \{\mathit{some}(x) \mid x \in \mathit{Atom}(A), \mathit{some}(x) \neq \bot\} \\
E_{\mathcal{A}} &\triangleq \{(\mathit{some}(x), \mathit{some}(\mathit{next}(x))) \mid x \in \mathit{Atom}(A), \mathit{some}(x) \neq \bot\} \\
\Pi_{\mathcal{A}}(x) &\triangleq \left\{ i \mapsto \mathit{some}(\mathit{next}^i(x)), -i \mapsto \mathit{some}(\mathit{prev}^i(x)) \mid i \in \mathbb{N} \right\} \\
T_{\mathcal{A}} &\triangleq \{\Pi(x) \mid x \in \mathit{Atom}(A), \mathit{some}(x) \neq \bot\} \\
\mathit{prop}_{\mathcal{A}}(x) &\triangleq \left\{ p \mid x \sqsubseteq p^{\mathcal{A}} \right\}, \text{ where } x \in S_{\mathcal{A}} \\
\mathit{act}_{\mathcal{A}}(x, y) &\triangleq \left\{ a \mid x \sqsubseteq a^{\mathcal{A}} \right\}, \text{ where } (x, y) \in E_{\mathcal{A}}
\end{aligned}
$$

We show that $\mathit{trel}(\mathcal{A})$ is a transition system by showing that $S_{\mathcal{A}}$ is non-empty and that $E_{\mathcal{A}}$, $T_{\mathcal{A}}$, $\mathit{prop}_{\mathcal{A}}$ and $\mathit{act}_{\mathcal{A}}$ are, respectively, a transition relation, closed set of traces and propositional and action labelling functions over $S_{\mathcal{A}}$.

Lemma 3.67. *A trace algebra $\mathcal{A}$ generates a trace system $\mathit{trel}(\mathcal{A})$.*

Proof. The set of states is the set of atoms of $\mathit{some}(A)$. The transition relation is, by definition, over states. As *next* and *prev* are perfect Boolean algebra isomorphisms, when applied to elements of $S_{\mathcal{A}}$, we obtain elements of $S_{\mathcal{A}}$ which respect the transition relation. ⊣

Although we can map between trace systems and trace algebras, neither family of structures can generate the other. A trace system may contain states and transitions that do not appear on any trace and will consequently not appear in the trace system generated by an algebra. A trace algebra may contain redundant representations of a trace, which precludes a representation theorem. The solution is to identify minimal trace systems and algebras, which do not contain such redundancies.

MINIMALITY A trace system is *minimal* if every state appears as the origin of a trace and if, for every transition (s, t), there is a trace π with $\pi_0 = s$ and $\pi_1 = t$. A trace algebra $\mathcal{A}$ is *minimal* if, for all distinct elements x and y, there exists an $i \geq 0$ such that either $\mathit{some}(\mathit{next}^i(x)) \neq \mathit{some}(\mathit{next}^i(y))$ or $\mathit{some}(\mathit{prev}^i(x)) \neq \mathit{some}(\mathit{prev}^i(y))$. Recall that next^0 is the identity function. Minimal trace algebras have unique representations as trace systems.

Theorem 3.68. *A minimal trace system M is isomorphic to the trace system of its algebra: $M \cong \mathit{trel}(\mathit{talg}(M))$.*

Proof. Let the trace systems in consideration be $M = (S, E, T, \mathit{prop}, \mathit{act})$ and $\mathit{trel}(\mathit{talg}(M)) = (S_{\mathcal{A}}, E_{\mathcal{A}}, T_{\mathcal{A}}, \mathit{prop}_{\mathcal{A}}, \mathit{act}_{\mathcal{A}})$, with $\mathit{talg}(M)$ being $\mathcal{A}$. Note that the elements of $S_{\mathcal{A}}$ are sets of traces. The map $h : s \mapsto \{\pi \in T \mid s = \pi_0\}$ is a trace system isomorphism.

(*States*) The function h is injective because sets of traces with different origins are distinct and because M is minimal, so every state occurs in a distinct set of traces.

(*Transitions*) Consider a transition (s, t) in E. Since M is minimal and T a closed set of traces, there is a trace π with $\pi_0 = s$ and $\pi_1 = t$. Moreover, there is a transition $(\mathit{some}(\pi), \mathit{some}(\mathit{next}(\pi)))$ in $E_{\mathcal{A}}$. Observe that this transition is $(h(s), h(t))$. Conversely, if $(h(s), h(t))$ is a transition, there must exist a trace π with $\pi_0 = s$ and $\pi_1 = t$, in turn implying that there is a transition (s, t) in E.

(*Traces*) A trace is a sequence of transitions. The reasoning follows that for transitions.

(*Propositions*) The label $prop(s)$ of a state contains a proposition p exactly if $p^{\mathcal{A}}$ contains traces with origin s, implying, by definition of *trel*, that $prop(h(s))$ contains p. Thus, $prop(s) = prop(h(s))$.

(*Actions*) Consider a transition (s, t) labelled a. The trace system is minimal, so this transition is traversed on a trace, and the set of traces is closed, so there is a trace π with origin s and with $\pi_1 = t$. From the definition of the trace algebra $\mathcal{A}$, the trace π is contained in $a^{\mathcal{A}}$. It follows that a is in the label of $(h(s), h(t))$. ⊣

The previous statements show that minimal trace systems and trace algebras enjoy a tight correspondence. What of non-minimal trace systems and trace algebras? Every trace system can be minimised by omitting states and transitions that do not occur on traces. Similarly, every trace algebra can be minimised to one that generates the same trace system. With respect to generation of trace systems and trace algebras, minimality is not a restriction. Minimisation of a trace algebra is an abstraction.

The Abstraction Perspective

Trace algebras are related to state algebras by abstraction. The image of a trace algebra under *some* represents a lattice of states that occur as the origins of traces. The successor transformer *post* is a γ-complete abstraction of *next* and the predecessor transformer *pre* is a γ-complete abstraction of *prev*. Recall that every conjunctive and distributive predecessor algebra can be derived as an abstraction of a state algebra. There exist state algebras that do not arise as abstractions of trace algebras. This is because trace algebras are defined by two-way total transition systems but state algebras represent arbitrary transition systems.

Lemma 3.69. *Let $\mathcal{A} = (A, O_A)$ be a state algebra. The transition system represented by $\mathcal{A}$ is two-way total exactly if for every non-$\bot$ x in A, there exists a predecessor transformer satisfying $pre_a(x) \neq \bot$ and a successor transformer satisfying $post_b(x) \neq \bot$.*

Proof. The condition on transformers is referred to as the 'algebraic condition' below. There are two directions.

(*Two-way totality implies algebraic condition*) If a transition relation E is two-way total, the image $E(s)$ and the pre-image $E^{-1}(s)$ are non-empty for every state s. Consequently, there exists a symbol a such that $E_a(s)$ is non-empty and a symbol b such that $E_b^{-1}(s)$ is non-empty. From the representation of state algebras it follows that $post_a(s)$ and $pre_b(s)$ are non-empty. Additionally, by strict-additivity of the transformers, we have that for every non-empty X, there exist a and b such that $post_a(X)$ and $pre_b(X)$ are non-empty.

(*Algebraic condition implies two-way totality*) Consider a non-$\bot$ element x. If $post_a(x)$ is not bottom, there exists an atom y in $Atom(x)$ for which $Atom(post_a(x))$ is not empty. The state corresponding to y has a successor in the transition relation E_a. This property holds for all atoms, so the transition relation generated by $\mathcal{A}$ is total. By a symmetric argument, the transition relation is two-way total. ⊣

We conclude the study of trace algebras by formalising the abstraction between from trace algebras to state algebras. Let $\mathcal{A} = (A, O_A)$ be a state

algebra that defines a two-way total transition relation. The algebra $\mathcal{A}$ defines a transition system $srel(\mathcal{A}) = (Atom(A), E_{\mathcal{A}}, prop_{\mathcal{A}}, act_{\mathcal{A}})$, which in turn defines a trace system consisting of all traces of $srel(\mathcal{A})$. This trace system defines a trace algebra $\mathcal{B} = (B, O_{\mathcal{B}})$, that we denote $srel(\mathcal{A})$ by abuse of notation.

$$\begin{aligned} B &\triangleq \mathscr{P}(traces(srel(\mathcal{A}))) \\ some(x) &\triangleq \{\tau \mid \pi_0 = \tau_0 \text{ for some } x \text{ in } x\} \\ next(x) &\triangleq \{step(\pi, 1) \mid \pi \text{ is in } x\} \\ prev(x) &\triangleq \{step(\pi, -1) \mid \pi \text{ is in } x\} \end{aligned}$$

The algebra $srel(\mathcal{A})$ is a trace algebra. We show that *some* is an upper closure on $\mathcal{A}$, so projecting out origin states can be viewed as an abstraction function.

Lemma 3.70. *The operator some is an upper closure on the lattice in $srel(\mathcal{A})$, where $\mathcal{A}$ is a state algebra.*

Proof. The lattice B contains only traces that exist in $srel(\mathcal{A})$, so $some(\top) = \top$. From the trace algebra axiom that $x \sqsubseteq some(x)$ for $x \sqsubseteq some(\top)$, we have that $x \sqsubseteq some(x)$ holds for all x in the lattice. ⊣

Using the operator *some* we can define the predecessor and successor transformers as given below. State algebras are complete abstractions in the sense given below. Note that the meet with the symbol a occurs before *next* is applied for the successor equality below and after *prev* is applied for the predecessor equality.

Theorem 3.71. *Let $\mathcal{A}$ be a state algebra defining the trace algebra $srel(\mathcal{A})$ as above.*

1. *The lattice A is isomorphic to the image $some(B)$ of the lattice of traces under the closure operator.*
2. *The successor transformer satisfies the equality*

   $$post_a(some(x)) = some(next(some(x) \sqcap a))$$

   for all elements x.
3. *The predecessor transformer satisfies the equality*

   $$pre_a(some(x)) = some(prev(some(x)) \sqcap a)$$

   for all elements x.

Proof. We consider each case separately.

(*Lattice Isomorphism*) We show that there is a bijection between $Atom(A$ and $Atom(some(B))$. First observe that each atom of $some(B)$ is a set of traces whose origin is the same and is an atom of A. Define a function f from an atom x of A to the set of traces $\{\pi \mid \pi \in traces(srel(\mathcal{A})), \pi_0 = x\}$. The function is injective because atoms are distinct. The function is surjective on $Atom(some(B))$ because the transition relation is two-way total, so every state occurs on a trace. The isomorphism of lattices follows because A and B are powersets over bijective sets.

(*Successor*) The set $post_a(some(x))$ contains all traces whose origin is a state reachable by a transition labelled a from a state occurring as the origin of a trace in x. The set of origins of traces reachable from x by one transition is given by $next(some(x))$ and those by transitions labelled a are given by $next(some(x) \sqcap a)$.

(*Predecessor*) The argument is similar to the successor case. The construction is different because labels of traces are defined with respect to outgoing traces but are symmetrically defined in a state algebra.

⊣

SECTION SUMMARY This section introduced an algebraic model of linear- and branching-time behaviour and introduced transformers for quantifying over traces and for navigating forwards and backwards through time. We used a non-standard, two-way infinite model of time. Although not as common as one-way infinite models, this model of time leads to a simpler algebraic treatment because well-foundedness issues do not arise. A second novelty of our model is in lifting existential trace quantification to sets of traces, to allow for a single-sorted model, unlike some existing two-sorted proposals. Our main result is the representation theorem showing that every trace system can be derived from a minimal trace algebra.

3.5 BIBLIOGRAPHIC NOTES

The literature on logic usually defines the syntax of a logic and then introduces structures of the form appearing in this section to give semantics to the logic. We have taken a cue from Poizat [2000] and chosen to study structures independent of syntactic concerns. The algebraic and relational structures introduced in this section reflect the distinct algebraic and relational traditions that have existed since the beginning of modern logic. We discuss related work from this perspective.

ALGEBRAIC AND RELATIONAL STRUCTURES IN LOGIC These two traditions exist and appear repeatedly in the study of propositional logic, modal logic and program verification. Boole's conception of logic in was algebraic and algebraic models were used in the work of his contemporaries. Frege's *Begriffsschrift* (1879) introduced the predicate calculus, which was used in the *Principia Mathematica* (1910 – 1913) and largely defines the current presentation of classical logics.

The two threads of algebraic and relational semantics emerge early in the foundations of modal logic. Jónsson and Tarski [1952] introduced Boolean algebras with operators, which give algebraic semantics to modal logic. Their work was largely unnoticed by modal logicians. For instance, some of their results are rediscovered by Lemmon [1966a; 1966b]. The reception of algebraic semantics is reflected in the introduction of [Lemmon 1966b].

> *I am very greatly indebted to the ideas and stimulus of Dana Scott, though I alone am responsible for the ugly algebraic form into which I have cast some of his elegant semantics.*

The social reception of algebraic semantics contrasts greatly with that of the relational semantics introduced by Kripke [1959] (who was 17 years old at the time). A scholarly survey of the origins of modal semantic structures appears in the book by Blackburn et al. [2006] which has this to say.

> *'Revolutionary' is an overused word, but no other word adequately describes the impact relational semantics [...] had on the study of modal logic.*

Consult Goldblatt's survey [2003] for a detailed discussion of mathematical perspectives from which to approach modal logic.

Trace algebras are comparable to temporal algebras of von Karger [2002] but differ by including branching-time modalities. The work of Möller et al. [2006] uses the language of quantales and includes trace concatenation as an operation in the algebra. Concatenation does not appear in trace algebras. Instead, we have considered two-way infinite traces and used operators to

navigate on these traces. We have derived the notion of two-way infinite traces from Cousot and the idea of lifting branching-time modalities and the semantics of state-formulae to traces from Cousot and Cousot [2000]. Unlike their work, our work has an axiomatic flavour, and is simpler, using only three operators.

ALGEBRAIC AND RELATIONAL STRUCTURES IN VERIFICATION In the computer science literature relational structures abound in various forms, ranging from automata used in language theory to graph-based data structures. Almost every relational structure used in computer science has an algebraic counterpart. For example, the algebraic approach to languages is based on monoids rather than automata. In all applications we are aware of, relational structures are used more frequently than algebraic structures. The semantics of a program can be defined using a transition system, as suggested by Keller [1976] or using lattices, following Scott [1969].

The relational and algebraic views lead down very different paths in program verification. Relational structures are graphs and are most easily analysed by graph-based algorithms. This is the trend in the early model checking literature [Clarke and Emerson 1981; Queille and Sifakis 1982]. Lattices lend themselves naturally to fixed point computations, as witnessed by the flow analysis literature [Kildall 1973] and notably by abstract interpretation [Cousot and Cousot 1977]. Cousot [1981] develops algebraic analysis starting from transition systems proving one direction of a representation theorem. The semantics of Kozen's μ-calculus [1982] uses the same construction. McMillan [1993] translated temporal logic properties into the μ-calculus and showed how to directly evaluate such an expression, developing an approach now known as *symbolic model checking* [Burch et al. 1992]. Semantically speaking, a symbolic model checking algorithm takes as input a representation of a transition system but operates on algebraic rather than relational semantics.

The verification literature is rarely concerned with questions of whether different semantic models are equivalent. The best known results in this direction are due to Steffen [1991; 1993], who proved that certain data-flow analysis problems formulated over lattices can equivalently be stated as model checking problems in terms of control flow graphs. The consequences of these results have been incorrectly interpreted. For instance, Beyer et al. [2007] begin their paper on combining static analysis and model checking techniques with the following statement.

> *Automatic program verification requires a choice between precision and efficiency. [..] Historically, this trade-off was reflected in two major approaches to static verification: program analysis and model checking. While in principle, each of the two approaches can be (and has been) viewed as a subcase of the other ...*

This claim is justified by referring to the papers of Steffen and Schmidt [1991; 1998; 1998]. The conclusion that model checking and abstract interpretation techniques can be seen as instances one another is incorrect. Steffen's results only apply to bit-vector data-flow analyses. The results of this chapter show that static analyses which require non-Boolean lattices cannot be translated into problems involving transition systems without first enriching the notion of a transition system. Prominent examples of such analyses include constant propagation, and analysis using the domains of intervals, octagons or polyhedra. In general, a lattice-based analysis is strictly more general than one defined over transition systems.

Weaker forms of the representation theorems proved in this chapter have appeared in the literature, often in disconnected contexts. Cousot [1981] derives an algebraic representation, called the *collecting semantics*, of a transition system but does not prove the equivalence of the two. Weaker forms of representation theorems have also been rediscovered by Loiseaux et al. [1995] and by Ranzato and Tapparo [2007]. Schmidt [2008] observed that non-Boolean abstract domains used in static analysis can be viewed as transition systems. His results show how one can derive logics from abstract domains and explore a complementary direction to the one studied in this chapter, where we extracted transition systems corresponding to abstract domains. Schmidt [2012] studies topological representations of abstract domains, but once again, does not use the framework of representation theorems. Topological representations for the algebras considered in this chapter can be synthesised using well known topological representations of lattices. We have consciously avoided topological constructions by working with perfect lattices. Our choice is justified by the prevalence of abstract domains based on perfect lattices.

The framework of Stone duality has been applied to rigorously show the equivalence of structures used in closely related area of programming language semantics. Abramsky [1987; 1991b] extended classical Stone duality to relate domain theory to modal logic and to topological spaces. In a strict mathematical sense, the work in this chapter overlaps with the work of Abramsky but does not generalise that work. Our work is simpler because we use discrete duality theory and restrict ourselves to lattices rather than categories and topological spaces. However, the ordered structures considered by Abramsky are distributive, while we consider non-distributive structures. Stone duality has also been applied to derive a logical characterisation of strictness analysis by Jensen [1991]. We believe that our work is the first application of Stone duality to structures at the intersection of model checking and program analysis.

4

LANGUAGES

> [..] the birth of modern temporal logic is unquestionably credited to Arthur Norman Prior, 1914-1969. Prior was a philosopher, who was interested in theological and ethical issues. His own religious path was somewhat convoluted; he was born a Methodist, converted to Presbytarianism, became an atheist, and ended up an agnostic.
>
> ...
>
> His wife, Mary Prior, recalled after his death:
>
> > "I remember his waking me one night [in 1953], coming and sitting on my bed, ..., and saying he thought one could make a formalised tense logic."
>
> ...
>
> It is interesting to note that the linear vs. branching time dichotomy, which has been a subject of some controversy in the computer science literature since 1980, has been present from the very beginning of temporal-logic development. In Prior's early work on temporal logic, he assumed that time was linear. In 1958, he received a letter from Saul Kripke, who wrote
>
> > "In an indetermined system, we perhaps should not regard time as a linear series, as you have done. Given the present moment, there are several possibilities for what the next moment may be like and for each possible next moment, there are several possibilities for the moment after that. Thus the situation takes the form, not of a linear sequence, but of a tree."
>
> – Moshe Vardi, *From Church and Prior to PSL*, 2007

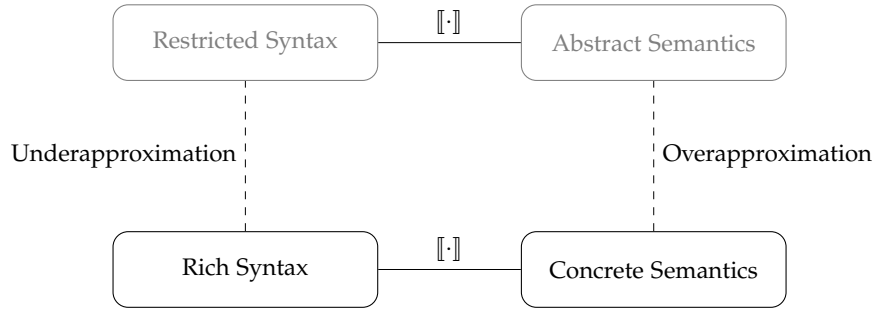
This chapter introduces a framework called *abstraction of syntax* for manipulating the syntax of grammars. The contribution of this chapter is to show that standard manipulation of syntactic objects, such as defining languages, logics, and representations of abstract domains, can all be formalised by abstract interpretation of syntax. The chapter presents a new perspective on the design of abstractions by showing that the problem of designing semantic overapproximations can be approached by designing syntactic underapproximations. This perspective is demonstrated via two case studies. The first case study shows that several numerical abstract domains used in practice can be derived from a variant of Presburger arithmetic by purely syntactic means. The second case study shows that several temporal logics used in practice can also be derived in such a manner.

4.1 OVERVIEW

The fields of automated reasoning and program analysis deal with an intricate interplay between syntax and semantics. The goal of program analysis is to make deductions about semantic properties of a program. These deductions must be expressed syntactically, either as human-readable formulae, or by data-structures stored in memory. This chapter formalises this observation that manipulation of semantics amounts to manipulation of syntax. A consequence of this observation, also explored in this chapter, is that approximation of semantics can be approached from a different perspective by approximating syntax.

Consider the process of designing an abstract interpreter. The first step is to characterise the behaviour of a system by fixed point in a lattice of semantic objects. The second step is to approximate this fixed point using a lattice and transformers, together called an abstract domain. The third step is to compute this fixed point approximation using iteration algorithms and operators to ensure termination and improve precision. The designer of an abstract domain usually has to answer three questions when executing these steps. What are the elements of the domain? How are abstract transformers implemented? Does analysis with the domain produce information for solving the problem? The first is a specification question, the second, an algorithmic question, and the third, an empirical evaluation question. The work in this chapter provides a syntactic view of abstract domain design.

The schema below expresses the relationships between syntactic and semantic objects studied in this chapter. Syntactic and semantic objects are related by an interpretation function $[\![\cdot]\!]$. Overapproximation of semantics can be viewed in syntactic terms as limiting what can be expressed. Limiting a language is underapproximation of syntax.



This chapter formalises the relationships in the figure above by introducing a meta-language for expressing syntax and by formalising *abstraction of syntax* using abstract interpretation.

Abstraction of syntax is inspired by proof theory. Proof theory provides a mathematical basis for studying the structure and properties of formal proofs, and procedures for deriving proofs. Though proof theory does not formalise, or even automate, the exact process by which mathematicians construct proofs, it has led to new insights concerning the structure of proofs, and to automated deduction techniques. Analogously, abstraction of syntax provides a mathematical formalisation of how abstract domains can be derived in purely syntactic, mechanical terms.

One may ask if the syntactic, logical approach we take has advantages over the semantic approach usually adopted in the literature. In the Galois insertion setting, a result of Cousot and Cousot [1979] shows that the space of domains is a complete lattice. Domains in this lattice can be combined

using reduced sum, reduced product, reduced power, and various other operations on domains [Cousot and Cousot 1979; Filé et al. 1996; Giacobazzi et al. 2000]. This lattice of domains is a rich mathematical object that cannot be searched manually. In the absence of a computational representation for arbitrary domains, it not known how to automate the search either.

In contrast, the syntax of a logic representing an abstract domain can usually be described by a short BNF grammar. Committing to a syntactic representation may restrict the kinds of domains that can be expressed, but also makes it easier to enumerate syntactic descriptions of domains. There are many parameters such as the number of operators, variables and nesting depth, that can be manipulated to generate logics from a grammar. We show with two case studies that a wide range domain specifications can be derived with our approach, and that the syntactic descriptions are succinct.

The first case study shows that semantic overapproximations used in practice can be derived by underapproximation of syntax. We show that formulae representing numeric abstract domains can be derived by restricting the formulae of an extension of Presburger arithmetic. Our case study shows that abstraction of syntax can be used as an organisational principle for consolidating information regarding semantic domains.

The second case study introduces a temporal logic with linear- and branching-time modalities. The logic is interpreted using trace algebras. We show that several well-known logics can be derived from this logic by abstraction of syntax. The question of which logical model of time is best suited to reasoning about programs has been the subject of significant debate in the verification literature. The abstract interpretation perspective on this debate is that most logics used in practice can be derived as abstractions of a rich, expressive logic with forward, backward and time symmetric modalities. Our second case study shows that logics used in practice arise as syntactic abstractions of a rich logic.

SUMMARY OF CONCEPTS The following new concepts are introduced in this chapter.

1. Meta-syntax is a variant of BNF introduced in Definition 4.2. Meta-syntax includes uniform composition and substitution to make it more amenable to abstract interpretation than BNF.
2. Abstraction of syntax, formalised in Definition 4.8, allows for inductively defined sub-languages of the language of a meta-syntax grammar to be derived by abstract interpretation.
3. The structure defined by a language, and the special case of a completely conjunctive languages introduced in Definition 4.15, formalise the connection between syntactic underapproximation and semantic overapproximation.

SUMMARY OF RESULTS This chapter contains the following new results.

1. Theorem 4.16 shows that the structures generated by completely conjunctive sub-languages of a given language define semantic overapproximations.
2. Section 4.3 shows that several numeric abstract domains can be derived as sub-languages of a variant of Presburger arithmetic by abstraction of syntax. The sub-languages have simple specifications indicating that abstraction of syntax is a convenient way to navigate the landscape of numeric abstract domains.

3. Section 4.4 introduces a new logic that subsumes most propositional temporal logics existing in the literature. The logic contains past, future, linear- and branching time modalities and has a lattice-based semantics in terms of trace algebras. We show that several logics used in practice can be derived as syntactic abstractions of this logic.

CHAPTER ORGANISATION Meta-syntax in introduced in Section 4.2 and is used to formalise the idea that overapproximation of semantics can be viewed as underapproximation of syntax. The first case study showing that several numeric abstract domains can be derived from a variant of Presburger arithmetic is presented in Section 4.3. The second case study, which shows that temporal logics used in practice can be derived by abstraction of syntax, is presented in Section 4.4.

4.2 A META-LANGUAGE FOR SYNTAX

Meta-syntax is a system for making inductive definitions, similar to BNF. In contrast to BNF, we can collate symbols that have different arity. The purpose is to use the same grammar to define different languages and to apply abstract interpretation to define sub-languages.

Example 4.1. The difference between BNF and meta-syntax is illustrated using propositional logic formulae. Let *Prop* be a set of propositions and p range over *Prop*. BNF definitions for propositional logic and two sub-logics are given below below.

$$\varphi ::= p \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \neg\varphi \qquad \text{Propositional logic}$$
$$\varphi ::= p \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \qquad \text{Monotone propositional logic}$$
$$\varphi ::= p \mid \varphi \wedge \varphi \qquad \text{Conjunctive propositional logic}$$

Each grammar is an inductive definition: a propositional formula φ is a proposition, or is the composition of formulae using the Boolean connectives $\wedge$, $\vee$ and $\neg$. The grammars above are similar but are not identical. Meta-syntax makes such similarities precise. Observe that each set of formulae contains the one below it. Using meta-syntax we can show that each definition is an underapproximating abstraction of the one preceding it.

Meta-syntax definitions have the same form as BNF but use macros instead of actual symbols. These macros are instantiated to obtain languages. The expression $\widehat{\mathsf{f}}$ indicates a sequence of symbols corresponding to f but also matching the arity of op.

$$\mathsf{f} ::= \mathsf{prop} \mid \mathsf{op}(\widehat{\mathsf{f}}) \qquad\qquad \mathsf{L}$$

The definition above inductively defines a language L that contains constant symbols prop and is closed under a set of operators. The symbols prop, op and f are macros for propositions, operators and formulae, respectively. Three different instantiations of the macros are given below.

$$\mathsf{inst}_1 \mathrel{\hat{=}} \{\mathsf{prop} \mapsto \mathit{Prop}, \mathsf{op} \mapsto \{\neg, \vee, \wedge\}\}$$
$$\mathsf{inst}_2 \mathrel{\hat{=}} \{\mathsf{prop} \mapsto \mathit{Prop}, \mathsf{op} \mapsto \{\vee, \wedge\}\}$$
$$\mathsf{inst}_3 \mathrel{\hat{=}} \{\mathsf{prop} \mapsto \mathit{Prop}, \mathsf{op} \mapsto \{\wedge\}\}$$

The instantiation of f is not specified above but is derived by from a fixed point equation specified by the grammar. The three instantiations yield the

three languages shown above. The difference is that we *explicitly* use the same grammar. Another difference is that the instantiations can be ordered by pointwise inclusion. This order implies language inclusion. ⌟

META-SYNTAX The system above is presented formally below.

Definition 4.2. Let Var be a set of meta-variables and Sym be a set of meta-symbols. A *syntax term* is one of the following.

1. A *meta-variable* x or a *meta-symbol* s.
2. The *composition* $\mathsf{s}(t_0, \ldots, t_{n-1})$ of a *meta-symbol* with terms.
3. The *uniform composition* $\mathsf{s}(\widehat{t})$ of a meta-symbol s with a term t.
4. The *substitution* $\mathsf{t}_1[\mathsf{x} \mapsto \mathsf{t}_2]$ of a meta-symbol x in a syntax term t_1 with a syntax term t_2.

A *grammar rule* over a meta-variable x and terms t_i has the form below.

$$\mathsf{x} ::= t_0 \mid \cdots \mid t_{n-1}$$

A *grammar* is a finite set of grammar rules.

The set of free meta-variables in a term t is written meta-var(t) and defined in the standard manner below.

$$\begin{aligned}
\text{meta-var}(\mathsf{x}) &\triangleq \{\mathsf{x}\} \\
\text{meta-var}(\mathsf{s}) &\triangleq \emptyset \\
\text{meta-var}(\mathsf{s}(t_0, \ldots, t_{n-1})) &\triangleq \bigcup_{i \leq n} \text{meta-var}(t_i) \\
\text{meta-var}(\mathsf{s}(\widehat{t})) &\triangleq \text{meta-var}(t) \\
\text{meta-var}(t_1 \mid t_2) &\triangleq \text{meta-var}(t_1) \cup \text{meta-var}(t_2) \\
\text{meta-var}\begin{pmatrix} \mathsf{x}_0 ::= d_0 \\ \vdots \\ \mathsf{x}_{n-1} ::= d_{n-1} \end{pmatrix} &\triangleq \left(\bigcup_i \text{meta-var}(d_i) \right) \setminus \{\mathsf{x}_0, \ldots, \mathsf{x}_{n-1}\}
\end{aligned}$$

The grammar below is used frequently in this dissertation and is called the *closed grammar*. Languages generated by this grammar are closed under concatenation with certain symbols.

$$\mathsf{x} ::= \mathsf{const} \mid \mathsf{op}(\widehat{\mathsf{x}})$$

The language defined by a grammar is defined next. A signature, as in Section 2.5, is a set of symbols with an arity function. A signature is *finite* if it contains finitely many symbols, *finitary* if it only contains symbols of finite arity and *completely finite* if it is finite and finitary. A signature that is not finite is infinite and one that is not finitary is infinitary. Fix a signature *Sig* for the rest of the section. The language defined by a meta-syntax term is expressed using labelled trees.

Definition 4.3. A *Sig*-labelled *syntax tree* $\tau = (V, E, sym)$ is a finite-height, ordered tree (V, E) with a function $sym : V \to Sig$ labelling vertices with symbols. The children of a vertex form a possibly infinite sequence $\bar{v} = v_0, v_1, \ldots$. The root of a tree is $root(\tau)$. The set of syntax trees over *Sig* is $\mathsf{Syn}(Sig)$, written Syn if *Sig* is clear.

The prefix *Sig*-labelled is dropped when clear. A *well formed tree* satisfies that every vertex v has $ar(sym(v))$ children. The tree below on the left is not well-formed but the tree on the right is well-formed. An *expression* is a well-formed syntax tree and a *language* is a set of expressions.

Symbols in a language are assigned to meta-symbols by an instantiation function.

Definition 4.4. An *instantiation* $\mathsf{In} = (\mathsf{Syn}, \mathsf{inst})$ is a set of syntax trees Syn with an instantiation function $\mathsf{inst} : \mathsf{Sym} \to \mathscr{P}(\mathsf{Syn})$. The instantiation function maps each meta-symbol to a set of trees with a single vertex. A *syntax environment* (or just environment) $\mathsf{env} : \mathsf{Var} \to \mathscr{P}(\mathsf{Syn})$ is a map from meta-variables to sets of syntax trees.

Intuitively, a single vertex is a symbol, but in strict terms, the operations on trees are different from operations on symbols. The *meaning* of a term t given an instantiation $\mathsf{In} = (\mathsf{Syn}, \mathsf{inst})$ and environment env, is a set of syntax trees, denoted $\|t\|_{\mathsf{In},\mathsf{env}}$, and defined inductively below. The arity of the symbol at the root of a tree τ is abbreviated to $ars(\tau)$.

$$
\begin{aligned}
\|\mathsf{x}\|_{\mathsf{In},\mathsf{env}} &\triangleq \mathsf{env}(\mathsf{x}) \\
\|\mathsf{s}(t_0,\ldots,t_{n-1})\|_{\mathsf{In},\mathsf{env}} &\triangleq \{\tau \mid \mathit{root}(\tau) \text{ is in } \mathsf{inst}(\mathsf{s}), ars(\tau) = n \\
&\qquad \text{and child } \tau_i \text{ of } \mathit{root}(\tau) \text{ is in } \|t_i\|_{\mathsf{In},\mathsf{env}}\} \\
\|\mathsf{s}(\widehat{t})\|_{\mathsf{In},\mathsf{env}} &\triangleq \{\tau \mid \mathit{root}(\tau) \text{ is in } \mathsf{inst}(\mathsf{s}) \text{ and for } i < ars(\tau) \\
&\qquad \tau \text{ has children } \tau_i \in \|t\|_{\mathsf{In},\mathsf{env}}\} \\
\|t_1[\mathsf{x}/t_2]\|_{\mathsf{In},\mathsf{env}} &\triangleq \{\tau \mid \tau \text{ is in } \|t_1\|_{\mathsf{In},\mathsf{env}'}, \text{ where } \mathsf{env}' \text{ maps} \\
&\qquad \mathsf{x} \text{ to } \|t_2\|_{\mathsf{In},\mathsf{env}}, \text{ and } \mathsf{y} \neq \mathsf{x} \text{ to } \|t_1\|_{\mathsf{In},\mathsf{env}}\} \\
\|t_1 \mid t_2\|_{\mathsf{In},\mathsf{env}} &\triangleq \|t_1\|_{\mathsf{In},\mathsf{env}} \cup \|t_2\|_{\mathsf{In},\mathsf{env}}
\end{aligned}
$$

The terms above may contain free variables. The *language of a grammar* containing the rules $\mathsf{x}_1 ::= t_1, \ldots, \mathsf{x}_n ::= t_n$ and no free variables is

$$\text{the least environment satisfying } \mathsf{env}(\mathsf{x}_i) = \|t_i\|_{\mathsf{In},\mathsf{env}} \text{ for every } i.$$

Note that the language of a grammar is not a single set of syntax trees but a tuple containing a set of syntax trees for each rule. A language Lang is *closed* under a symbol f if there is a syntax tree for $f(\bar{\varphi})$ for every $ar(f)$-sequence $\bar{\varphi}$ of syntax trees in Lang. Observe that languages generated by the closed grammar are closed under the symbols in the instantiation of op.

We now give a generic definition of the depth of a symbol in a syntax tree. This definition is required in Section 5.2. The supremum of a set of ordinals X is written $\sup X$.

$$
\begin{aligned}
dep(S,p) &= 0 && \text{if } ar(p) = 0 \text{ and } p \notin S \\
&= 1 && \text{if } ar(p) = 0 \text{ and } p \in S \\
dep(S, f(\bar{\varphi})) &= \sup\{dep(\varphi_i) \mid \varphi_i \text{ is in } \bar{\varphi}\} && \text{if } f \notin S \\
&= \sup\{dep(\varphi_i) \mid \varphi_i \text{ is in } \bar{\varphi}\} + 1 && \text{if } f \in S
\end{aligned}
$$

Collecting Interpretation of Meta-Syntax

It is known, since the work of Chomsky and Schutzenberger [1963] that languages defined by grammars have fixed point characterisations. Ginsburg and Rice [1962] extended this characterisation to recursively enumerable languages. Istrail [1982] showed that concatenation, union, intersection and substitution on languages suffice for the fixed point characterisation. It

follows that the languages defined by meta-syntax grammars have fixed point characterisations. There has, however, been very little work combining such fixed point characterisations with abstract interpretation. We facilitate this combination by defining the semantics of meta-syntax by transformers.

The *domain of syntax trees* is the powerset of syntax trees

$$(\mathscr{P}(\mathsf{Syn}), \subseteq, \cap, \cup)$$

and the *domain of instantiation functions* is

$$(\mathsf{Inst}, \sqsubseteq, \sqcap, \sqcup),$$

where Inst is the set of functions $\mathsf{Sym} \to \mathscr{P}(\mathsf{Syn})$ with pointwise order and operations. The *domain of syntax environments* is

$$(\mathsf{Env}, \sqsubseteq, \sqcap, \sqcup),$$

where Env is the set of functions $\mathsf{Var} \to \mathscr{P}(\mathsf{Syn})$ with pointwise order and operations. Note that these domains are complete lattices. The bottom element among syntax environments, denoted $\mathsf{env}_\bot$, maps all meta-variables to the emptyset.

The meaning of terms is lifted to the domains above using syntax transformers. Fix an instantiation $\mathsf{In} = (\mathsf{Syn}, \mathsf{inst})$. A *syntax transformer* for a term t is a function $syn_t : \mathsf{Env} \to \mathscr{P}(\mathsf{Syn})$ that maps environments to syntax trees. The syntax transformer for a grammar rule $\mathsf{x} ::= r$ is a function $syn_{\mathsf{x}::=r} : \mathsf{Env} \to \mathsf{Env}$ that maps between environments. Both transformers are defined below.

$$\begin{aligned} syn_t &\mathrel{\hat{=}} \mathsf{env} \to \|t\|_{\mathsf{In},\mathsf{env}} \\ syn_{x::=r} &\mathrel{\hat{=}} \mathsf{env} \to \mathsf{env}[\mathsf{x}/\, syn_r(\mathsf{env})] \end{aligned}$$

Since the function $\|t\|_{\mathsf{In},\mathsf{env}}$ is inductively defined, so is syn_t. The transformer for the separator $\mid$ is the join operator. The language of a grammar can be characterised as a fixed point over syntax transformers. This characterisation requires a function null to filter out nullary symbols.

$$\begin{aligned} \mathsf{null}(\mathsf{s}) &\mathrel{\hat{=}} \{\tau \in \mathsf{inst}(\mathsf{s}) \mid ars(\tau) = 0\} \\ \mathsf{null}(\mathsf{x}) &\mathrel{\hat{=}} \emptyset \\ \mathsf{null}(\mathsf{s}(\widehat{t})) &\mathrel{\hat{=}} \mathsf{null}(\mathsf{s}) \\ \mathsf{null}(t_1 \mid t_2) &\mathrel{\hat{=}} \mathsf{null}(t_1) \cup \mathsf{null}(t_1) \end{aligned}$$

The meaning of a grammar is defined next. Each grammar rule $\mathsf{x}_i ::= r_i$ generates an equation

$$\mathsf{x}_i = \mathsf{null}(r_i) \sqcup syn_{r_i}(\mathsf{env})$$

and the meaning of a grammar is the least solution to these equations. Recall that, in a transition system, the states reachable from a set *Init* are characterised as the fixed point of the equation $X = Init \cup post(X)$, where *post* is the successor transformer. Observe that the equations generated by a grammar have the same form. The example below illustrates a fixed point computation with syntax transformers.

Example 4.5. Consider the grammar and instantiation below. For convenience, symbols are treated as single-vertex syntax trees.

$$\mathsf{f} ::= \mathsf{prop} \mid \mathsf{op}(\widehat{\mathsf{f}}) \qquad \mathsf{inst} \mathrel{\hat{=}} \{\mathsf{prop} \mapsto \mathit{Prop}, \mathsf{op} \mapsto \{\neg, \vee, \wedge\}\}$$

The syntax transformers generated by this grammar are given below.

$$\begin{aligned} syn_{\mathsf{prop}} &\mathrel{\hat{=}} \mathsf{env} \mapsto \mathit{Prop} \\ syn_{\mathsf{op}(\widehat{\mathsf{f}})} &\mathrel{\hat{=}} \mathsf{env} \mapsto \{\neg\varphi, \varphi \wedge \psi, \varphi \vee \psi \mid \varphi, \psi \in \mathsf{env}(\mathsf{f})\} \end{aligned}$$

Iterating through the grammar to compute a fixed point produces the sequence of meta-syntax environments below.

$$\mathsf{env}_0 = \{\mathsf{f} \mapsto \emptyset\} \qquad \mathsf{env}_1 = \{\mathsf{f} \mapsto \mathit{Prop}\} \quad \cdots \quad \mathsf{env}_n = \{\mathsf{f} \mapsto T_n\}$$

The initial environment is empty. After one iteration, the value of f is the set of propositions. After n iterations, T_n contains expressions with at most $n-1$ operators. The fixed point of this sequence maps the meta-symbol f to the set of propositional formulae. ⌟

We will combine the fixed point semantics, of the form in Example 4.5, with abstract interpretation to derive inductively defined sub-languages of a meta-syntax grammar. We call this combination of abstract interpretation and meta-syntax *abstraction of syntax*. The examples below illustrate abstraction of syntax.

Example 4.6. This example shows that the monotone fragment of propositional logic can be derived from the full logic using an abstraction. Consider the meta-syntax grammar and the two signatures below.

$$\mathsf{f} ::= \mathsf{prop} \mid \mathsf{op}(\widehat{\mathsf{f}}) \qquad \mathit{Sig}_1 \mathrel{\hat{=}} \mathit{Prop} \cup \{\wedge, \vee, \neg\} \qquad \mathit{Sig}_2 \mathrel{\hat{=}} \mathit{Prop} \cup \{\wedge, \vee\}$$

The two signatures define domains of syntax trees that are related by a Galois connection. This Galois connection lifts in a standard manner to instantiation functions and syntax environments. Let $\mathsf{Inst}(\mathit{Sig}_1)$ denote the set of instantiation functions over $\mathsf{Syn}(\mathit{Sig}_1)$.

$$\begin{aligned} (\mathscr{P}(\mathsf{Syn}(\mathit{Sig}_1)), \supseteq) &\underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (\mathscr{P}(\mathsf{Syn}(\mathit{Sig}_2)), \supseteq) \\ (\mathsf{Inst}(\mathit{Sig}_1), \sqsupseteq) &\underset{\alpha_{\mathsf{In}}}{\overset{\gamma_{\mathsf{In}}}{\leftrightarrows}} (\mathsf{Inst}(\mathit{Sig}_2), \sqsupseteq) \end{aligned}$$

Note the superset order, indicating that the abstraction is underapproximating. The abstraction and concretisation functions between domains of syntax trees and their pointwise lifting to instantiation functions is given below.

$$\begin{aligned} \alpha &\mathrel{\hat{=}} T \mapsto T \cap \mathsf{Syn}(\mathit{Sig}_2) & \gamma &\mathrel{\hat{=}} S \mapsto S \\ \alpha_{\mathsf{In}} &\mathrel{\hat{=}} \mathsf{inst} \mapsto \{\mathsf{s} \mapsto \alpha(\mathsf{inst}(\mathsf{s}))\} & \gamma_{\mathsf{In}} &\mathrel{\hat{=}} \mathsf{inst}' \mapsto \{\mathsf{s} \mapsto \mathsf{inst}'(\mathsf{s})\} \end{aligned}$$

The concretisation function is the identity function on the subset of elements that is in both domains.

Consider the instantiation $(\mathsf{Syn}(\mathit{Sig}_1), \mathsf{inst})$ with the instantiation function

$$\mathsf{inst} \mathrel{\hat{=}} \{\mathsf{prop} \mapsto \mathit{Prop}, \mathsf{op} \mapsto \{\neg, \wedge, \vee\}\}$$

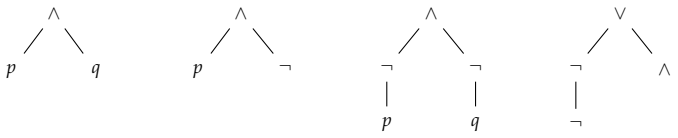
that we used for propositional logic. Using the Galois connection, we can replace inst by its best abstract transformer to obtain a tuple

$$(\mathscr{P}(\mathsf{Syn}(\mathit{Sig}_2)), \alpha_{\mathsf{In}} \circ \mathsf{inst} \circ \gamma_{\mathsf{In}})$$

called an *abstract instantiation* Evaluating the fixed point of the grammar with this instantiation produces the monotone fragment of propositional logic. ⌟

A second example, below, illustrates that more complex domains can be used to derive sub-languages.

Example 4.7. Consider the same grammar and instantiation inst as in Example 4.6. Let ATree be the set of syntax trees with at most one negation symbol. The trees need not be well formed. The first two trees below are in ATree but the next two are not.



Observe that ATree cannot be of the form $\mathsf{Syn}(Sig)$ for any Sig contained in Sig_1 because Sig must contain negation and the resulting trees, multiple negations. Nonetheless, ATree is a subset of $\mathsf{Syn}(Sig_1)$, so again, we have a Galois connection as in Example 4.6. To approximate the fixed point of the grammar, we use *abstract syntax transformers* on the lattice $\mathsf{Var} \to \mathscr{P}(\mathsf{ATree})$. The fixed point contains formulae with at most one negation. ⌟

We formalise the notions used in Example 4.7 using the concepts of abstract interpretation. A subset of symbols of a signature is a *sub-signature*. A subset of expressions in a language is a *sub-language*.

Definition 4.8. An instantiation $\mathsf{AIn} = (\mathsf{ATree}, \mathsf{ainst})$ is a *syntactic abstraction* of $\mathsf{In} = (\mathsf{Syn}, \mathsf{inst})$ if ATree is a subset of Syn and for every meta-symbol and meta-variable u, $\mathsf{ainst}(\mathsf{u}) \subseteq \mathsf{inst}(\mathsf{u})$.

Syntactic abstractions are underapproximating abstractions in the sense of abstract interpretation.

Proposition 4.9. *A syntactic abstraction* $(\mathsf{ATree}, \mathsf{ainst})$ *of* $(\mathsf{Syn}, \mathsf{inst})$ *defines a Galois connection between* $(\mathscr{P}(\mathsf{Syn}), \supseteq)$ *and* $(\mathscr{P}(\mathsf{ATree}), \supseteq)$.

Proof. Define an abstraction function $\alpha : \mathscr{P}(\mathsf{Syn}) \to \mathscr{P}(\mathsf{ATree})$ to be $\alpha(X) \mathrel{\hat{=}} X \cap \mathsf{ATree}$ and the concretisation function $\gamma : \mathscr{P}(\mathsf{ATree}) \to \mathscr{P}(\mathsf{Syn})$ to be the identity function. Consider a set $X \subseteq \mathsf{Syn}$ and $Y \subseteq \mathsf{ATree}$. The two functions are a Galois connection because $\alpha(X) = X \cap \mathsf{ATree}$ and $\alpha(X) \supseteq Y$ if and only if $X \supseteq Y$. ⊣

A syntactic abstraction AIn defines an *inductively defined* sub-language $\|\mathsf{Gram}\|_{\mathsf{AIn}}$ of $\|\mathsf{Gram}\|_{\mathsf{In}}$. Signature abstractions are a special class of syntactic abstractions obtained by considering sub-signatures.

Definition 4.10. A *signature abstraction* of an instantiation $(\mathsf{Syn}, \mathsf{inst})$ over a signature Sig, is a tuple $(\mathsf{ATree}, \mathsf{ainst})$, where ATree is the set of syntax trees over a sub-signature $Sig' \subseteq Sig$ and ainst sends a meta-symbol or variable s to $\mathsf{inst}(\mathsf{s}) \cap \mathsf{ATree}$.

The abstraction in Example 4.7 was not a signature abstraction.

Semantic Abstractions from Syntactic Abstractions

A language is typically used to describe elements of some mathematical structure. The mapping between a language, which consists of syntactic objects, and a mathematical structure which gives that language semantics, is called an interpretation function. We define interpretations of languages using domains and transformers. Our definition of an interpretation function strictly generalises the standard notion of an interpretation used in classical

logic. The standard notions are defined in terms of sets and relations. The representation theorems from the previous section show that the sets and relations can be encoded by powerset domains with completely additive transformers.

Definition 4.11. A *semantic structure* $\mathcal{A} = (\mathcal{A}, Int_{\mathcal{A}})$ for a signature *Sig* is a family of lattices $\mathcal{A}$ called *domains* and an family of functions $Int_{\mathcal{A}}$ called the *interpretation*. There are $ar(f) + 1$ domains in $\mathcal{A}$ and a function $f^{\mathcal{A}}$ of the form below for each $f \in Sig$.

$$\mathcal{A} = \left\{ A_{f,i} \mid f \in Sig \text{ and } 0 \leq i \leq ar(f) \right\}$$
$$Int_{\mathcal{A}} = \left\{ f^{\mathcal{A}} : A_{f,0} \times \cdots \times A_{f,n-1} \to A_{f,n} \mid f \in Sig \text{ and } n = ar(f) \right\}$$

Fix a semantic structure $\mathcal{A} = (\mathcal{A}, Int_{\mathcal{A}})$. If all domains in $\mathcal{A}$ are identical, the semantic structure is an algebra and the semantic structure is written (A, O_A) as usual. A *semantics function* $[\![\cdot]\!]_{\mathcal{A}}$ maps expressions in a language to domain elements. A semantics function is *partially* defined as shown below. This definition is partial because the function compositions below may not be well defined.

$$[\![c]\!]_{\mathcal{S}} \triangleq c^{\mathcal{S}} \text{ where } c, \text{ is a symbol with arity } 0$$
$$[\![f(\bar{a})]\!]_{\mathcal{S}} \triangleq f^{\mathcal{S}}([\![a_0]\!]_{\mathcal{S}}, \ldots, [\![a_{n-1}]\!]_{\mathcal{S}}), \text{ where } f(\bar{a}) \in \|\mathsf{s}(t_0, \ldots, t_{n-1})\|_{\Lambda}$$
$$[\![f(\bar{a})]\!]_{\mathcal{S}} \triangleq f^{\mathcal{S}}([\![a_0]\!]_{\mathcal{S}}, \ldots, [\![a_i]\!]_{\mathcal{S}}, \ldots), \text{ where } f(\bar{a}) \in \|\mathsf{s}(\widehat{t})\|_{\Lambda}$$

An expression has well defined semantics if the arguments of a function symbol have the same domain as their subexpressions. This notion is made precise using types. The semantic structure $\mathcal{A}$ defines

$$\text{a set of types } \mathsf{Type} = \mathcal{A} \cup \{\delta\}, \text{ with } \delta \text{ being a symbol not in } \mathcal{A}.$$

That is, every domain is treated as a type. The type of an expression is defined below.

$$\begin{aligned} \mathsf{type}(c) &\triangleq A_{c,0} && \text{if } c \text{ is a constant} \\ \mathsf{type}(f(\bar{a})) &\triangleq A_{f,ar(f)} && \text{if for all } i \in dom(\bar{a}), \mathsf{type}(a_i) = A_{f,i} \\ &\triangleq \delta && \text{otherwise} \end{aligned}$$

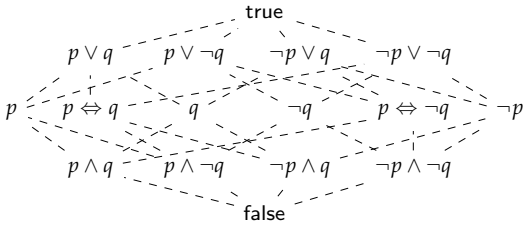
An expression e is *well defined* with respect to $\mathcal{A}$ if $\mathsf{type}(e)$ is not δ. All expressions are assumed to have well defined semantics. The syntactic perspective on abstraction is illustrated in the next example.

Example 4.12. This example illustrates syntactic descriptions of semantic abstractions. Imagine a program with a single variable x that ranges over the mathematical integers $\mathbb{Z}$. The concrete domain containing possible values of x is the lattice $\mathscr{P}(\mathbb{Z})$ shown in Figure 14. In the figure, each lattice in the left column is an abstraction of the semantics of the program. The semantic abstractions have representations as formulae on the right. For instance, $x \leq 1$ represents the interval $[-\infty, 1]$. The lattice of intervals can be viewed as a logic containing $x \leq k$ and $x \geq k$, closed under conjunction but not under disjunction. Implication defines the lattice order. ⌟

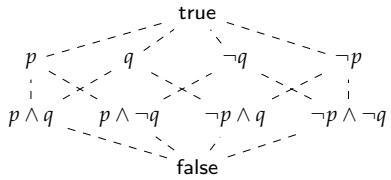
The utility of view above is an alternative approach to the challenging problem of designing abstract domains. Cousot and Cousot [1979] showed that the space of domains is a complete lattice. The lattice of domains is a

rich mathematical object that cannot be manually searched. In contrast, the syntax of a logic representing an abstract domain is usually at most two lines of BNF. Parameters such as the number of operators, variables and nesting depth can be manipulated to generate sub-languages. The example below shows that arbitrary sub-languages do not generate abstract domains. In general, syntactic abstraction and semantic abstraction are distinct.

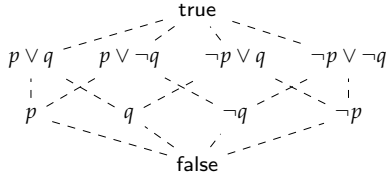
Example 4.13. All languages in this example can be derived by abstraction of syntax, but are presented informally to save space. Let Lang_1 be propositional logic over two variables p and q. Propositional formulae are interpreted over the lattice $\mathscr{P}(Prop \to \mathbb{B})$ containing truth assignments. The structure $struct(\mathsf{Lang}_1)$ obtained by interpreting all formulae over this lattice is below.



Consider the sub-language Lang_2, which contains propositions, negations of propositions and is closed under conjunction. The language Lang_2 does not contain disjunction. The lattice $struct(\mathsf{Lang}_2)$ obtained by evaluating such formulae is shown below. There is a Galois connection between $struct(\mathsf{Lang}_1)$ and $struct(\mathsf{Lang}_2)$.



A third language Lang_3 contains propositions, negation of propositions, and is closed under disjunction. The interpretation of formulae in Lang_3 yields the lattice $struct(\mathsf{Lang}_3)$ below.



For the lattice order in the figure, there is no Galois connection from $struct(\mathsf{Lang}_1)$ to $struct(\mathsf{Lang}_3)$ because $struct(\mathsf{Lang}_3)$ contains p and q, but not $p \wedge q$. However, Lang_3 and Lang_2 are both sub-language of Lang_1 and can both be derived by abstraction of syntax. ⌟

The abstraction of semantic structures is defined next. Recall from Section 2.1 that $\bar{\alpha}[\bar{x}]$ is the application of a sequence of functions to a sequence of arguments.

Definition 4.14. A semantic structure $\mathcal{A} = (\mathcal{A}, Int_{\mathcal{A}})$ is an *abstraction* of $\mathcal{C} = (\mathcal{C}, Int_{\mathcal{C}})$ if every f in Sig satisfies the conditions below.

1. There is a Galois connection $(C_{f,i}, \alpha_{f,i}, \gamma_{f,i}, A_{f,i})$, for every $0 \leq i \leq ar(f)$, between the domains for f in $\mathcal{A}$ and $\mathcal{C}$.

2. Let $\bar{\alpha}$ be the $ar(f)$-termed sequence of abstraction functions $(\alpha_{f,0}, \ldots \alpha_{f,i}, \ldots)$ each of which maps from the domains for f in $\mathcal{C}$ to their abstractions in $\mathcal{A}$. The interpretations of f satisfy the pointwise constraint $\alpha_{f,ar(f)} \circ f^{\mathcal{C}}(\bar{x}) \sqsubseteq f^{\mathcal{A}} \circ \bar{\alpha}[\bar{x}]$.

In loose terms, an abstraction is a collection of abstract domains and sound abstract transformers. If the Galois connection is provided, an abstraction consisting of best transformers can be generated. Let $\mathcal{C}$ and $\mathcal{A}$ be sets of domains and Γ be a family of Galois connections as defined below.

$$\Gamma \triangleq \left\{ (C_{f,i}, \alpha_{f,i}, \gamma_{f,i}, A_{f,i}) \mid f \in Sig \text{ and } 0 \leq i \leq ar(f) \right\}$$

between the concrete and abstract domains. The abstraction *induced by* Γ is the semantic structure $\mathcal{A} = (\mathcal{A}, Int_{\Gamma})$ containing a function

$$f^{\mathcal{A}} \triangleq \bar{x} \mapsto \alpha_{f,ar(f)} \circ f^{\mathcal{S}} \circ \bar{\gamma}[\bar{x}]$$

representing the best abstract transformer for each f in *Sig*. The sequence $\bar{\gamma} = (\gamma_{f,0}, \ldots)$ contains concretisation functions.

Fix a structure $(\mathsf{Gram}, \mathsf{In}, \mathcal{S})$ defining a language Lang. We can consider the fragment of $\mathcal{S}$ obtained by interpreting expressions in the language and obtain a poset. The *structure defined by* Lang, denoted *struct*(Lang), is the family of posets that together contain the elements $\{[\![e]\!]_{\mathcal{S}} \in \ \mid e \in \mathsf{Lang}\}$, and are ordered as before.

Definition 4.15. Let Lang be a language interpreted over a semantic structure $\mathcal{S} = (\mathcal{A}, \mathcal{F})$, with a single domain $\mathcal{A}$. The language Lang is *completely conjunctive* if for every set of expressions $E \subseteq \mathsf{Lang}$, there exists an expression C_E in Lang satisfying the equality $\bigsqcap \{[\![e]\!]_{\mathcal{S}} \mid e \in E\} = [\![C_E]\!]_{\mathcal{S}}$.

The extension of the definition above (and the theorem below) to structures involving multiple domains is not mathematically more complicated but is notationally cumbersome, so we skip it. The theorem below is new, and shows that completely conjunctive languages define abstract domains.

Theorem 4.16. *Let* Lang *be a language interpreted over the semantic structure* $\mathcal{S} = (\mathcal{A}, \mathcal{F})$*, where* $\mathcal{A}$ *is the domain* $(A, \sqsubseteq, \sqcap, \sqcup)$*. If* Lang *is completely conjunctive, the poset* *struct*(Lang) *is in a Galois insertion with* C*.*

Proof. We prove the statement in two steps, first showing that *struct*(Lang) is a complete lattice, and then that *struct*(Lang) is the image of A under a closure operator.

(*struct*(Lang) *is a complete lattice*) Consider two expressions e_1 and e_2 and the element $[\![e_1]\!]_{\mathcal{S}} \sqcap [\![e_2]\!]_{\mathcal{S}}$. Since Lang is completely conjunctive, we have that there exists an expression e satisfying $[\![e]\!]_{\mathcal{S}} = [\![e_1]\!]_{\mathcal{S}} \sqcap [\![e_2]\!]_{\mathcal{S}}$. Thus *struct*(Lang) is a meet-semi-lattice. Moreover, since Lang is completely conjunctive, the meet of every subset of expressions is definable, so *struct*(Lang) is a complete lattice.

(*struct*(Lang) *is an abstraction*) Define the function below.

$$\rho \triangleq \{x \mapsto [\![C_E]\!]_{\mathcal{S}} \mid e \in E \text{ if } x \sqsubseteq [\![e]\!]_{\mathcal{S}}\}$$

The function ρ is, by definition, monotone, idempotent and extensive, hence is an upper closure. It follows that $\rho(A)$ is an abstraction of A. The semantics of every expression in Lang is also present in $\rho(A)$.

By Ward's theorem, every Galois insertion can be represented as an upper closure on a lattice, so *struct*(Lang) is an abstraction of A, with respect to a Galois insertion. ⊣

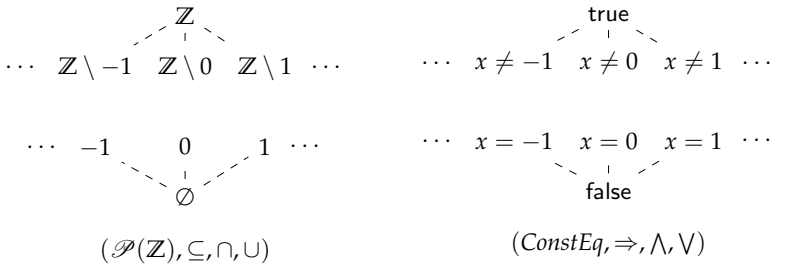


Figure 14: Two equivalent descriptions of the same mathematical objects. The left column shows the powerset lattice of integers on top, the lattice of integer intervals and the lattice of integer constants. These domains are represented by formulae in the column on the right. The lattice order is implication and the meet operator is conjunction. The join is not disjunction for the intervals and constants.

SECTION SUMMARY This section introduced a variant of BNF called meta-syntax. By combining abstract interpretation with the fixed point semantics of a meta-syntax grammar, we obtain a method to derive inductively defined sub-languages of the language of the grammar. For formal languages interpreted over lattices, elements of a sub-language can be viewed as defining elements of an abstract domain. We have shown that sub-languages that are closed under meet operations in a lattice define abstract domains in the Galois connection sense.

4.3 SYNTACTIC CONSTRUCTION OF ABSTRACT DOMAINS

In this section, we show that abstraction of syntax provides a systematic framework for specifying abstract domains. By combining abstract interpretation with meta-syntax, the search space for the intuitive art of abstract domain design can be make completely formal. In our opinion, the value of this view is comparable to the view that mathematical theorems are logical formulae and proofs are sequences of formulae satisfying certain consistency conditions. A purely syntactic view does not capture the intent and utility of theorems or abstract domains, but it makes explicit that there is a strict sense in which abstract domain design can be viewed as a mechanical process.

The design of an abstract domain can be approached syntactically by following the steps below. These steps are not the only way to approach abstract domain design and, in fact, are non-standard.

1. Define a structure $(\mathsf{Gram}, \mathsf{In}, \mathcal{S})$ for a logic expressing all properties of interest for a family of programs. Let $\mathcal{S}$ be the semantic structure for this logic.
2. Fix a set of syntactic parameters such that each set of parameter values generates a logic L of only the program properties we consider.
3. If L is closed under infinitary conjunction, the semantic structure $\mathit{struct}(\mathsf{L})$ is an abstract domain. If L contains modal operators, the structure $\mathit{struct}(\mathsf{L})$ contains the definitions of transformers.

In the rest of this section, we present a case study by applying the steps above to derive abstract domains of numeric properties.

A Logic of Numerical Properties

The logic we use to specify the semantics of programs is an extension of Presburger arithmetic with infinitary conjunction and disjunction, and with the next-state modality. This logic can encode properties (such as reachability or termination) of programs written in Turing complete languages. The grammar below defines a language with terms and formulae.

$$\mathsf{t} ::= \mathsf{var} \mid \mathsf{fun}(\widehat{\mathsf{t}}), \quad \mathsf{f} ::= \mathsf{pred}(\widehat{\mathsf{t}}) \mid \mathsf{bool}(\widehat{\mathsf{f}}) \mid \mathsf{mod}(\widehat{\mathsf{f}})$$

The signature *Sig* contains variables *Var* and the sets below (k is a positive integer). The binary predicate $\equiv_k$ denotes congruence modulo k. Other symbols have their standard arity.

$$\mathit{Fun} \triangleq \{+\} \cup \mathbb{N} \quad \mathit{Pred} \triangleq \{<, \leq, =, >, \geq, \equiv_k\} \quad \mathit{Bool} \triangleq \left\{\bigvee, \bigwedge, \wedge, \vee, \neg\right\}$$

The instantiation of meta-symbols is below.

$$\mathsf{var} \mapsto \mathit{Var} \quad \mathsf{fun} \mapsto \mathit{Fun} \quad \mathsf{pred} \mapsto \mathit{Pred} \quad \mathsf{bool} \mapsto \mathit{Bool} \quad \mathsf{mod} \mapsto \{\mathsf{EX}\}$$

An example formula is $\mathsf{EX}(x = y + y)$, stating that there is a successor state in which x equals $2y$. Familiar modalities can be recovered using infinitary

operators: $\mathsf{AX}\varphi$ is the formula $\neg\mathsf{EX}\neg\varphi$, stating that all successors satisfy φ, and $\mathsf{AG}\varphi$, defined as $\bigwedge_i \mathsf{AX}^i\varphi$, asserts that φ is true on every path. Infinitary operators can also be used to express multiplication.

We interpret this logic over transition systems. Let *Val* be a set of values and $Env \triangleq Var \to Val$ be the set of states (program environments). A *transition system* $M = (Env, E)$ contains a relation $E \subseteq Env \times Env$. Recall that the transition relation E generates a *predecessor transformer* $pre : \mathscr{P}(Env) \to \mathscr{P}(Env)$ defined below.

$$pre(X) \triangleq \{s \in Env \mid \text{ there exists } t \text{ in } X \text{ and } (s,t) \text{ is in } E\}$$

The semantic structure we need contains the powerset lattice of natural numbers $\mathscr{P}(\mathbb{N})$ and the powerset lattice of environments $\mathscr{P}(Env)$. The semantics of a term t is a set of values given by the function $[\![t]\!]_{\mathcal{S}} : \mathscr{P}(Env) \to \mathscr{P}(\mathbb{N})$ defined below.

$$\begin{aligned} [\![x]\!]_{\mathcal{S}} &\triangleq X \mapsto \{\varepsilon(x) \mid \varepsilon \in X\} \\ [\![t_1 + t_2]\!]_{\mathcal{S}} &\triangleq X \mapsto \{m + n \mid m \in [\![t_1]\!]_{\mathcal{S}}, n \in [\![t_2]\!]_{\mathcal{S}}\} \end{aligned}$$

The semantics of a formula φ is a set of environments defined by the function $[\![\varphi]\!]_{\mathcal{S}} : \mathscr{P}(Env) \to \mathscr{P}(Env)$ defined inductively below.

$$\begin{aligned} [\![\mathsf{EX}\varphi]\!]_{\mathcal{S}} &\triangleq pre([\![\varphi]\!]_{\mathcal{S}}) \\ [\![P(\bar{t})]\!]_{\mathcal{S}} &\triangleq \{\varepsilon \in Env \mid \varepsilon \text{ satisfies } P(\bar{t})\} \\ [\![\varphi \wedge \psi]\!]_{\mathcal{S}} &\triangleq [\![\varphi]\!]_{\mathcal{S}} \cap [\![\varphi]\!]_{\mathcal{S}} \\ [\![\varphi \vee \psi]\!]_{\mathcal{S}} &\triangleq [\![\varphi]\!]_{\mathcal{S}} \cup [\![\varphi]\!]_{\mathcal{S}} \\ [\![\neg\varphi]\!]_{\mathcal{S}} &\triangleq Env \setminus [\![\varphi]\!]_{\mathcal{S}} \end{aligned}$$

Infinitary conjunction and disjunction are interpreted as arbitrary union and intersection, respectively. We now derive abstractions of $\mathscr{P}(Env)$ using abstraction of syntax.

Parameters for Defining Sub-Languages

The parameters introduced in this section enable compact specification of the restrictions made on a meta-syntax grammar. A *k-variable predicate* is

an expression $P(\bar{t})$ that contains at most k variables.

Note that having k variables is a property of the terms appearing in the predicate, unlike the arity, which is a property of the predicate symbol. We introduce the notion of syntax parameters as a succinct way to specify families of syntax trees.

Definition 4.17. Let *Sig* be a signature. A *syntax parameter* is a tuple $(S, k) \in \mathscr{P}(Sig) \times \mathbb{N}_\infty$ consisting of a signature $S \subseteq Sig$ and a *depth bound* k. The relation $(Q, m) \sqsubseteq (S, n)$ between syntax parameters holds if $Q \subseteq S$ and $m \leq n$ both hold.

A syntax parameter (S, k) defines a set of syntax trees ATree over the symbols $S \cup Var \cup \{\bigwedge, \wedge\}$ such that every element of ATree is a k-variable predicate. The set of trees defined by a syntax parameter always includes variables and finite and infinitary conjunction.

Proposition 4.18. *The set of syntax parameters with the order $\sqsubseteq$ is a complete lattice.*

The results of the previous section imply that evaluating the grammar Gram over ATree defines a sub-language Lang for every parameter value, and that $struct(\mathsf{Lang})$ defines an abstract domain over $\mathscr{P}(Env)$. Let $struct(P)$ denote the substructure for the language generated by a parameter P. We will illustrate the passage from parameters via sub-languages to abstract domains. To reduce clutter, we write the parameter $(\{\leq, \equiv_2\}, 3)$ as $(\leq, \equiv_2; 3)$ in figures.

Example 4.19. Consider the transition system M, depicted on the left below. States represent the values of a variable x. A standard method of abstraction is to partition states using predicates. The result N of partitioning states of M using the predicates $x < 0$ and $x = 0$ is depicted on the right below.

We will see how this abstraction, and others, can be derived syntactically. Three parameters are given below and satisfy the order $P_3 \sqsubseteq P_2 \sqsubseteq P_1$.

$$P_1 \mathrel{\hat{=}} (0, \leq, =, >, \vee, \neg, \mathsf{EX}; 1)$$
$$P_2 \mathrel{\hat{=}} (0, \leq, =, >, \vee, \mathsf{EX}; 1)$$
$$P_3 \mathrel{\hat{=}} (0, \leq, =, >, \mathsf{EX}; 1)$$

The language generated by P_1 contains the constant 0, is closed under Boolean operations and has predicates with at most one variable. The predicate $x < 0$ is expressible in this language but $x < y$ is not. The language generated by P_2 is only closed under conjunction and disjunction but not negation, while the language generated by P_3 is only closed under conjunction. All the languages are also closed under the modal operator EX.

The structures $struct(P_i)$ for each parameter contain a lattice representing the semantics of predicates and a transformer for the semantics of EX. These substructures are shown in Figure 15. Every predicate in the first lattice represents a set of states of N above and *pre* is the predecessor function for N. Note that this abstraction *was not* derived from N but by evaluating formulae. The other two lattices are not closed under Boolean operations, hence they cannot be represented as transition systems. Working with parameters allows us to generate abstractions that cannot be generated by partitioning states. ⌟

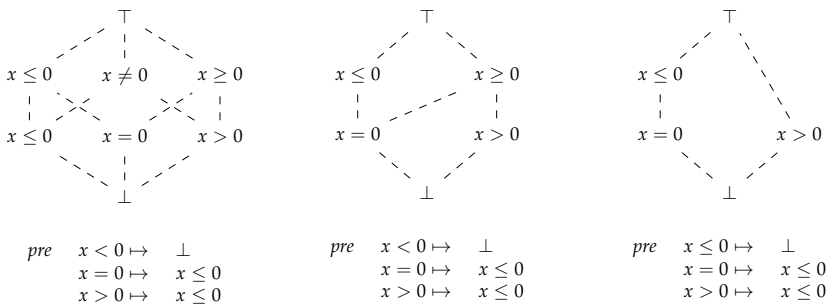


Figure 15: Abstract domains generated signature abstraction.

Generating Abstract Domains

We now show that appropriate parameters generate abstract domains that are used in practice. We emphasise again that the approach we suggest does not solve all the issues that arise in designing a domain. Nonetheless, it provides a designer with one approach to thinking of abstract domains and focusing on the properties of interest. We now show that a small range of parameter values generates the specifications of many abstract domains used in practice.

RELATIONAL AND NON-RELATIONAL ABSTRACTIONS A standard form of abstraction is to decouple the relationship between variables. Non-relational domains cannot express facts like $x = y$, while relational abstractions can. The terms weakly-relational refers to domains that express a limited range of relational information. The octagon domain is weakly relational because one octagon can express $x + y \leq c$, but not $x + y + z \leq c$, because at most two variables can occur in a constraint. A parameter satisfying $(S, m) \sqsubseteq (Sig, 1)$ contains at most one variable per predicate and cannot express relational information. A parameter satisfying $(S, m) \sqsubseteq (Sig, n)$ for finite n generates a weakly relational domain, because some, but not all relational information can be expressed. A parameter (S, ∞) contains predicates with no bound on the number of variables. Such parameters generate domains encoding relational information.

TEMPORAL ABSTRACTIONS We use the term temporal abstractions for those based in bisimulation, simulation and similar preorders defined on transition systems. The facts we use about such equivalences are summarised in [Blackburn et al. 2001]. Recall that a relation $R \subseteq Env \times Env$ is a simulation if whenever (s, t) is in R and (s, s') is a transition, there must be a transition (t, t') such that (s', t') is in R. It is known that bisimulation can be characterised by a logic that is closed under Boolean operations, infinitary conjunction, and EX, and that simulation is characterised by a logic closed under infinitary conjunction, disjunction and EX. We can use these results to specify abstract domains based on bisimulation and simulation.

Every parameter greater than $(\{\bigvee, \neg, \mathsf{EX}\}, 0)$ represents a bisimulation quotient with respect to a given set of predicates. The first abstraction in Figure 15 represents the bisimulation quotient of M using the predicates $x < 0$, $x = 0$ and $x > 0$. Every parameter greater than $(\{\bigvee, \mathsf{EX}\}, 0)$, containing infinitary disjunction but not necessarily negation, represents a simulation preserving domain. Ranzato and Tapparo [2007] study such domains in greater detail.

NUMERIC ABSTRACTIONS We now discuss parameter settings that generate standard abstract domains. For complete rigour, each claim that follows should be accompanied with a proof that the generated domain represents the claimed existing domain. Such proofs are not difficult, but are skipped here to focus on the conceptual message.

- *Affine Equalities* $(\{+, \mathbb{N}, =\}, \infty)$. This abstraction contains conjunctions of constraints of the form $a_1 x_1 + \cdots a_n x_n = k$.

- *Affine Congruences* $(\{+, \mathbb{N}, \equiv_k\}, \infty)$. This abstraction contains conjunctions of constraints of the form $a_1 x_1 + \cdots a_n x_n \equiv_k m$, stating that the constraint on the left is congruent to m, modulo k.

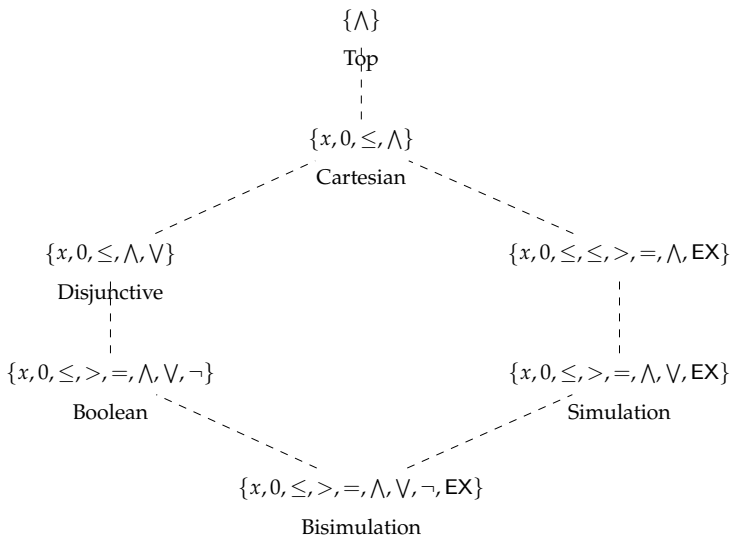


Figure 16: A lattice of signatures that generates the abstract domains shown.

- *Intervals* $(\{\mathbb{N}, \leq\}, 1)$. If at most one variable per predicate is allowed when using inequalities, we have constraints of the form $0 \leq x \wedge x \leq 3$, which encode that x is in the interval $[0, 3]$.
- *Constants* $(\{\mathbb{N}, =\}, 1)$. If at most one variable per predicate is allowed when using equalities, we have constraints of the form $x = 3$. Since no two distinct equalities are satisfiable, all conjunctions of formulae become false. This domain is used in constant propagation.
- *Parity* $(\{0, 1, \equiv_2\}, 1)$. If the domain is non-relational and the only predicate is $\equiv_2$, we obtain the parity domain.
- *Signs* $(\{0, <, =, >\}, 1)$. A non-relational domain in which every variable can only be compared with 0 is the signs domain. The parameter above only generates the 5 element signs domain containing the predicates $x < 0$, $x = 0$ and $x > 0$. If extended with disjunction, we obtain the 8 element signs domain closed under Boolean operations.

We emphasise that a signature can contain infinitely many symbols and the resulting domain can contain infinitely many elements. Several numeric domains mentioned above have infinitely many elements. Note also that the domains of constants, affine equalities and congruences, all contain infinitely many inequalities but have finite height. Logically, such domains correspond to logics in which there are no infinite sequences of strict implications. For these reasons, generating domains by abstraction of syntax is difficult to automate. Predicate abstraction [Ball et al. 2001] applies only to finite sets of predicates, but is automatic. The parameters for some domains we discussed are shown in Figure 16.

SECTION SUMMARY This section demonstrated that abstraction of syntax can be applied as a principle for organising abstract domains. Several abstract domains used in practice can be derived from a common, syntactic sublanguage by abstraction of syntax with simple restrictions on the language of the grammar. These results suggest an approach to design abstract domains

by first deriving a syntactic representation of the domain and then deriving algorithms to manipulate that representation.

4.4 TEMPORAL LOGICS

This section introduces the syntax of temporal logics. A temporal logic is a set of formulae constructed from propositional and action symbols, modalities and Boolean operators. A modality is basic or derived. The *basic modalities* are next operator $\mathsf{X}_{\rightarrow}$, a previous operator $\mathsf{X}_{\leftarrow}$ and existential trace projection E, all having unit arity. A *derived modality* is obtained by composing basic modalities, propositions, actions and Boolean operators. The Boolean operators are negation, conjunction and disjunction. Conjunction and disjunction may have finite or infinite arity. The condition for a trace π to satisfy a formula is described informally below. In the description, p is a proposition and a is an action.

p	The origin of π satisfies p.
a	The first transition in π satisfies a.
$\mathsf{X}_{\rightarrow}\varphi$	When shifted by one step, π satisfies φ.
$\mathsf{X}_{\leftarrow}\varphi$	When rewound by one step, π satisfies φ.
$\mathsf{E}\varphi$	A trace with the origin of π satisfies φ.
$\neg\varphi$	π does not satisfy φ.
$\varphi \wedge \psi$	π satisfies φ and satisfies ψ.
$\varphi \vee \psi$	π satisfies at least one of φ and ψ.
$\bigwedge \bar{\varphi}$	π satisfies every formula in $\bar{\varphi}$.
$\bigvee \bar{\varphi}$	π satisfies at least one formula in $\bar{\varphi}$.

The arities of $\bigwedge$ and $\bigvee$ are both ω unless otherwise specified. A pronunciation guide for basic and derived modalities appears below. The derived modalities are defined in Table 1. Defining derived modalities in this manner allows us to define semantics of a broad family of temporal logics by defining the semantics of basic modalities. The *bounded until* formula $\varphi \mathsf{BU}_k \psi$ in Table 1 is satisfied by a trace π if the first $k-1$ steps of π satisfy φ and π satisfies ψ after exactly k steps. The until formula $\varphi \mathsf{U}_{\rightarrow} \psi$ is satisfied by π if there exists a k for which the bounded until formula holds. A trace π eventually satisfies φ, written $\mathsf{F}_{\rightarrow}\varphi$, if a future instant on π satisfies φ The trace π always satisfies φ, written $\mathsf{G}_{\rightarrow}\varphi$, if every future instant satisfies φ. Past analogues of future modalities are derived by replacing $\mathsf{X}_{\rightarrow}$ by $\mathsf{X}_{\leftarrow}$ in the definition of a modality. The formula $\mathsf{G}_{\rightleftharpoons}\varphi$ combines future and past modalities and is satisfied by π if φ holds at every time instant.

Next	$\mathsf{X}_{\rightarrow}$	Previous	$\mathsf{X}_{\leftarrow}$
Until	$\mathsf{U}_{\rightarrow}$	Since	$\mathsf{U}_{\leftarrow}$
Eventually	$\mathsf{F}_{\rightarrow}$	Previously	$\mathsf{F}_{\leftarrow}$
Will always be	$\mathsf{G}_{\rightarrow}$	Was always	$\mathsf{G}_{\leftarrow}$
Forever	$\mathsf{G}_{\rightleftharpoons}$		
Some trace	E	All traces	A

A *temporal signature* is a set of proposition and action symbols, modalities and Boolean operators. A *temporal logic* is a language over a temporal signature. A temporal logic is a *state logic* if every basic modality is preceded by E or A and is a *trace logic* otherwise. Temporal formulae are called *properties*. The semantic structures for temporal logics are *temporal algebras*.

Notation	Definition	Explanation
$\mathsf{BU}_0(\varphi,\psi)$	ψ	
$\mathsf{BU}_{k+1}(\varphi,\psi)$	$\varphi \wedge \mathsf{X}_{\rightarrow}\mathsf{BU}_k(\varphi,\psi)$	π satisfies φ for k steps and then satisfies ψ.
$\varphi\mathsf{U}_{\rightarrow}\psi$	$\bigvee_{i\in\mathbb{N}} \mathsf{BU}_i(\varphi,\psi)$	π satisfies ψ in the future and until then satisfies φ.
$\mathsf{F}_{\rightarrow}\varphi$	$\mathsf{trueU}_{\rightarrow}\varphi$	π eventually satisfies φ.
$\mathsf{G}_{\rightarrow}\varphi$	$\neg\mathsf{F}_{\rightarrow}\neg\varphi$	Future instants satisfy φ.
$\mathsf{BS}_0(\varphi,\psi)$	ψ	
$\mathsf{BS}_{k+1}(\varphi,\psi)$	$\varphi \wedge \mathsf{X}_{\leftarrow}\mathsf{BS}_k(\varphi,\psi)$	π satisfied φ in the k steps satisfied ψ $k+1$ steps ago.
$\varphi\mathsf{U}_{\leftarrow}\psi$	$\bigvee_{i\in\mathbb{N}} \mathsf{BS}_i(\varphi,\psi)$	π satisfied ψ in the past and since then has satisfied φ.
$\mathsf{F}_{\leftarrow}\varphi$	$\mathsf{trueU}_{\leftarrow}\varphi$	φ held at a past instant.
$\mathsf{G}_{\leftarrow}\varphi$	$\neg\mathsf{F}_{\leftarrow}\varphi$	All past instants satisfied φ.
$\mathsf{G}_{\rightleftharpoons}\varphi$	$\mathsf{G}_{\rightarrow}\varphi \wedge \mathsf{G}_{\leftarrow}\varphi$	Every instant satisfies φ.
$\mathsf{A}\varphi$	$\neg\mathsf{E}\neg\varphi$	Every trace with origin π_0 satisfies φ

Table 1: Temporal Modalities

Temporal algebras are derived from trace algebras. The interpretation of logical operators over a trace algebra $\mathcal{A}$ is given below.

$$
\begin{aligned}
p^{\mathcal{A}} &\triangleq p^{\mathcal{A}} & a^{\mathcal{A}} &\triangleq a^{\mathcal{A}} \\
\mathsf{X}^{\mathcal{A}}_{\rightarrow}(Y) &\triangleq \mathit{prev}^{\mathcal{A}}(Y) & \mathsf{X}^{\mathcal{A}}_{\leftarrow}(Y) &\triangleq \mathit{next}^{\mathcal{A}}(Y) \\
\mathsf{E}^{\mathcal{A}}(Y) &\triangleq \mathit{some}^{\mathcal{A}}(X) & \neg^{\mathcal{A}} Y &\triangleq \neg Y \\
X \wedge^{\mathcal{A}} Y &\triangleq X \sqcap Y & X \vee^{\mathcal{A}} Y &\triangleq X \sqcup Y \\
\bigwedge \bar{X} &\triangleq \bigsqcap \{X_i \mid i \in \mathit{dom}(\bar{X})\} & \bigvee \bar{X} &\triangleq \bigsqcup \{X_i \mid i \in \mathit{dom}(\bar{X})\}
\end{aligned}
$$

Derived modalities are interpreted by composing the interpretation of basic modalities. For example, the interpretation of $\mathsf{U}_{\rightarrow}$ is the composition of *next*, finite meets and an infinitary join following the definition in Table 1. The semantics of formulae is given by a function $[\![\cdot]\!]_{\mathcal{A}}$ defined as shown in Section 4.2. The interpretations of formulae in a state logic are elements of the state algebra generated by a trace algebra. As a result, we interpret state logics over state algebras.

An element of the domain of a trace algebra is a *point*. Traces from an initial state or to an error state are examples of points. A *pointed temporal algebra* $(\mathcal{A}, a)$ is a temporal algebra $\mathcal{A}$ with a point a. When no ambiguity arises, pointed temporal algebras and temporal algebras are called algebras. An algebra $(\mathcal{A}, a)$ *satisfies a formula* φ, written $\mathcal{A}, a \models \varphi$, if $a \sqsubseteq [\![\varphi]\!]_{\mathcal{A}}$. An algebra that satisfies a formula is a *model* of a formula. The set of formulae in a temporal logic L satisfied by a pointed algebra $(\mathcal{A}, a)$ is denoted $\mathsf{L}(\mathcal{A}, a)$.

We introduce a temporal logic called TL that unifies several existing logics. Fix the sets *Prop*, *Act* and *Bool* of propositions, actions and Boolean operators,

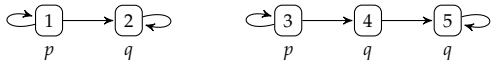
respectively. The set *Quant* of trace quantifiers is $\{\mathsf{A}, \mathsf{E}\}$ and *Tem* contains the modalities $\{\mathsf{X}_{\leftarrow}, \mathsf{X}_{\rightarrow}, \mathsf{U}_{\leftarrow}, \mathsf{U}_{\rightarrow}\}$. The signature of TL is

$$Sig_{\mathsf{TL}} \triangleq Prop \cup Act \cup Bool \cup Quant \cup Tem$$

and the grammar $\mathsf{Gram}_{\mathsf{TL}}$ and instantiation for the logic are given below.

$$\mathsf{x} ::= \mathsf{const} \mid \mathsf{op}(\widehat{\mathsf{x}})$$
$$\mathsf{inst} \triangleq \{\mathsf{const} \mapsto Prop \cup Act, \mathsf{op} \mapsto Bool \cup Quant \cup Tem\}$$

Example 4.20. Let us consider a few TL formulae interpreted over the transition systems below.

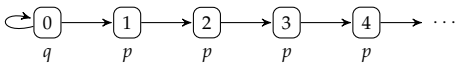


We denote the set of traces with origin s as $traces(s)$. The set $traces(1)$ satisfies the formulae $\mathsf{EG}_{\rightarrow}p$, $\mathsf{EF}_{\rightarrow}\mathsf{G}_{\rightarrow}q$ and $\mathsf{G}_{\leftarrow}p$. The set $traces(4)$ satisfies the formulae $\mathsf{X}_{\leftarrow}\mathsf{G}_{\leftarrow}p$, $\mathsf{G}_{\rightarrow}q$ and $\mathsf{AF}_{\rightarrow}(q \wedge \mathsf{EG}_{\leftarrow}q)$. The formula $p \wedge \mathsf{EX}_{\rightarrow}(q \wedge \mathsf{X}_{\rightarrow}(q \wedge \mathsf{EX}_{\leftarrow}p))$ asserts that the origin of a trace satisfies p, that there is a trace that satisfies q for two consecutive steps and that the state in the second step has a predecessor satisfying p. This formula is satisfied by $traces(1)$ but not by $traces(3)$. Let $\mathsf{X}^{n}_{\rightarrow}$ denote a sequence of n next operators. The property $\bigvee_{i\in\mathbb{N}} \mathsf{X}^{2i}_{\rightarrow}q$ states that q is true at even instants. This property cannot be expressed in LTL. ⌟

Temporal logics such as CTL, $\mathsf{CTL}^{\star}$ and the μ-calculus have the finite-model property, the tree-model property and are invariant under bisimulation. We shall see that none of these properties hold for TL or even the fragment of TL with finitary operators. A temporal logic has the *finite-model property* if every satisfiable formula has a model generated by a finite-state transition system.

Proposition 4.21. TL *does not have the finite model property.*

Proof. Let φ be the formula $\mathsf{EG}_{\rightarrow}(p \wedge \mathsf{AF}_{\leftarrow}\neg p)$. The set of traces $traces(1)$ from the transition system below satisfies φ.



We argue that this formula has no finite-state model. Assume that M is a finite-state transition system generating a temporal algebra $\mathcal{A}$ and let a be a point in $\mathcal{A}$ such that $(\mathcal{A}, a)$ satisfies φ. The algebra also satisfies the formula $\mathsf{EG}_{\rightarrow}p$, so for every trace π in a, $p \in prop(\pi_i)$ for $i \geq 0$. As M is finite, some state in M visited twice: there are indices $i < j$ with $\pi_i = \pi_j$. The trace π satisfies $\mathsf{EF}_{\rightarrow}\mathsf{EG}_{\leftarrow}p$ contradicting the assumption that π satisfies $\mathsf{EG}_{\rightarrow}\mathsf{AF}_{\leftarrow}\neg p$. ⊣

A temporal logic has the *tree-model property* if every satisfiable formula has a transition system model that is tree-shaped. The root of a tree has no predecessors. Two-way transition systems cannot be trees because every state must have a predecessor. A natural, relevant extension of trees is two-way trees. A two-way tree can be viewed as two trees adjoined by a common root. Paths flow into the root vertex in one of the trees and out of the root vertex of the other tree. Formally, a transition system $M = (S, E, prop, act)$ is a *two-way tree* if S is the union of two sets S_1 and S_2 such that there is a unique root state $\{s\} = S_1 \cap S_2$ with an edge relation E that is the disjoint

union $E_1 \cup E_2$ and such that the tuples (S_1, E_1) and (S_2, E_2^{-1}) are trees with a root s. A logic has the *two-way tree model property* if every satisfiable formula has a transition system model that is a two-way tree. The model need not be finite.

Proposition 4.22. TL *does not have the two-way tree model property.*

Proof. Let φ be the formula $p \wedge \mathsf{EF}_{\rightarrow}\mathsf{EG}_{\leftarrow}\neg p$. The transition systems and points $\mathit{traces}(1)$ and $\mathit{traces}(3)$ in Example 4.20 satisfy φ. Suppose π is a trace of a two-way tree that satisfies φ. There is a non-negative index i such that π_i is not labelled π. Moreover, there exists a trace θ such that $\pi_i = \theta_0$ and θ_j is not labelled p for every $j \leq 0$. If θ is generated by a two-way tree some past state is the root and is labelled p, so θ satisfies $\mathsf{AF}_{\leftarrow} p$ and a contradiction ensues. ⊣

Bisimulation is a relation between the states of transition systems. The reader familiar with bisimulation is referred to Example 4.20 for bisimilar transition systems distinguishable by TL formulae. The proofs that TL lacks properties common to standard logics do not use infinitary operators. The finitary fragment of TL also lacks the finite-model and two-way tree model properties and is not invariant under bisimulation. The remainder of this section relates TL to standard logics.

Deriving sub-logics by signature abstraction

We now apply the content of Section 4.2 to derive standard temporal logics by abstraction of syntax. The signature abstraction of a logic L is the language generated by interpreting the grammar of L over a restricted signature. The sub-signatures of TL given below define sub-logics by signature abstraction.

$\mathit{Finite} \triangleq \mathit{Sig}_{\mathsf{TL}} \setminus \left\{\bigwedge, \bigvee\right\}$	Finitary
$\mathit{Future} \triangleq \mathit{Finite} \setminus \{\mathsf{X}_{\leftarrow}, \mathsf{U}_{\leftarrow}\}$	Future-time
$\mathit{Past} \triangleq \mathit{Finite} \setminus \{\mathsf{X}_{\rightarrow}, \mathsf{U}_{\rightarrow}\}$	Past-time
$\mathit{Step} \triangleq \mathit{Finite} \setminus \{\mathsf{U}_{\leftarrow}, \mathsf{U}_{\rightarrow}\}$	Step
$\mathit{Until} \triangleq \mathit{Finite} \setminus \{\mathsf{X}_{\leftarrow}, \mathsf{X}_{\rightarrow}\}$	Until
$\mathit{Exist} \triangleq \mathit{Finite} \setminus \{\mathsf{A}, \neg\}$	Existential
$\mathit{Univ} \triangleq \mathit{Finite} \setminus \{\mathsf{E}, \neg\}$	Universal
$\mathit{Linear} \triangleq \mathit{Finite} \setminus \{\mathsf{A}, \mathsf{E}\}$	Linear-time

The signature *Finite* excludes infinitary operators. The logic generated by *Finite* extends $\mathsf{CTL}^\star$ with past modalities. The signature *Future* contains only future modalities and the resulting logic is similar, but not identical, to $\mathsf{CTL}^\star$. Unlike $\mathsf{CTL}^\star$, a distinction between state and trace formulae is not used to define semantics and formulae such as $\mathsf{AE}\varphi$ are permitted. The signature *Linear* excludes trace quantifiers. The resulting logic is a time-symmetric extension of LTL. Table 4.4 contains other examples of logics defined by signature abstraction. CTL is notable for its absence from Table 4.4. In fact, CTL cannot be obtained from $\mathsf{Gram}_{\mathsf{TL}}$ by signature abstraction. We give a proof to provide greater intuition about signature abstraction. A grammar $\mathsf{Gram}_{\mathsf{CTL}}$ and instantiation for CTL are given below.

$$\mathsf{x} ::= \mathsf{prop} \mid \mathsf{state}(\mathsf{tem}(\widehat{\mathsf{x}})) \mid \mathsf{bool}(\widehat{\mathsf{x}})$$

$$\mathsf{inst}_{\mathsf{CTL}} \triangleq \{\mathsf{const} \mapsto \mathit{Prop} \cup \mathit{Act}, \mathsf{state} \mapsto \mathit{Quant}, \mathsf{tem} \mapsto \mathit{Tem}, \mathsf{bool} \mapsto \mathit{Bool}\}$$

STL	*Finite*	Symmetric temporal logic
CTL*	*Future*	STL with only future modalities
PCTL*	*Past*	STL with only past modalities.
SLTL	*Linear*	Time symmetric LTL
LTL	*Linear* $\cap$ *Future*	Standard LTL
PLTL	*Linear* $\cap$ *Past*	Past-time LTL
SLTL$_\mathsf{X}$	*Linear* $\cap$ *Step*	Next fragment of SLTL
SLTL$_\mathsf{U}$	*Linear* $\cap$ *Until*	Until fragment of SLTL

Table 2: Sub-logics derived by signature abstraction.

Proposition 4.23. *The language* CTL *cannot be derived from the grammar and instantiation of* TL *by signature abstraction.*

Proof. Suppose a signature $S \subseteq Sig_{\mathsf{TL}}$ exists, giving rise to CTL by signature abstraction. The instantiation of $\mathsf{Gram}_{\mathsf{TL}}$ contains the formula $\mathsf{AX}_\rightarrow\varphi$. It will also include $\mathsf{AX}_\rightarrow\mathsf{X}_\rightarrow\varphi$, which is not in CTL. ⊣

Proposition 4.23 illustrates a limit of signature abstraction. It is not possible to distinguish between $\mathsf{AX}_\rightarrow$ and $\mathsf{AX}_\rightarrow\mathsf{X}_\rightarrow$ because symbols are treated in isolation. Signature abstraction is not context sensitive. However, context sensitive syntactic abstractions exist, as shown next.

Two families of syntax trees, $\mathsf{Quant}_{k,Sig}$ and $\mathsf{Tem}_{k,Sig}$ over a temporal signature *Sig* are defined below. The subscript *Sig* is dropped in the definition. Trees in Quant_k represent expressions with a bounded number of trace operators in the scope of a trace quantifier. Trees in Tem_k represent expressions in which the leading symbols are Boolean combinations of temporal operators.

1. A tree τ is in Quant_0 if temporal operators occur under trace quantifiers. That is, one of the conditions below should hold.
   a) The children of τ are in Quant_0 and $sym(root(\tau)) \notin Tem$.
   b) τ has a unique child $\tau' \in \mathsf{Tem}_0$ and $sym(root(\tau)) \in Quant$.
2. A tree τ is in Tem_0 if it represents an expression with temporal operator preceding an expression in Quant_0. That is, if $sym(root(\tau))$ is in *Tem* and all children of $root(\tau)$ are in Quant_0.

For $k \geq 0$ the sets are defined below.

1. A tree τ is in Quant_{k+1} if one of the conditions below holds.
   a) τ is in Quant_k.
   b) The children of τ are in Quant_{k+1} and $sym(root(\tau)) \notin Tem$.
   c) τ has a unique child $\tau' \in \mathsf{Tem}_{k+1}$ and $sym(root(\tau)) \in Quant$.
2. For $k > 0$, a tree τ is in Tem_{2k+1} if at most k temporal operators occur before a path operator. That is, either τ is in Tem_{2k} or its children are in Quant_{2k} and $sym(root(\tau))$ is in *Bool*.
3. A tree τ is in Tem_{2k} if k temporal operators and their Boolean combination can occur without a path operator *and* the k-th operator is not in the scope of a Boolean operator. That is, τ is in Tem_{2k-1} or, its children are in Quant_{2k-1} and $sym(root(\tau)) \in Tem$.

Acronym	Syntax Abstraction	Logic
SCTL	$\mathsf{QNest}_{0,\mathit{Finite}}$	Symmetric CTL
SCTL_U	$\mathsf{QNest}_{0,\mathit{Until}}$	Stuttering SCTL
CTL	$\mathsf{QNest}_{0,\mathit{Future}}$	Standard CTL
SCTL^k	$\mathsf{QNest}_{2k,\mathit{Finite}}$	$\mathsf{SCTL}^\star$ with temporal depth k
ACTL	$\mathsf{QNest}_{0,\mathit{Univ}\cap\mathit{Mono}}$	All-paths CTL
ML	$\mathsf{QNest}_{0,\mathit{Future}\cap\mathit{Step}}$	Modal logic

Table 3: Sub-logics obtained by syntactic abstraction.

The family of *nesting trees* $\mathsf{QNest}_k \mathrel{\hat{=}} \mathsf{Quant}_k \cup \mathsf{Tem}_{k,S}$ contains trees with a bound on the number of temporal operators in the scope of a trace quantifier. Observe that $\mathsf{QNest}_k \subseteq \mathsf{Syn}_{\mathsf{TL}}$. This set induces a syntactic abstraction, as shown in Section 4.2.

Proposition 4.24. *The syntactic abstraction induced by* $\mathsf{QNest}_{0,\mathit{Future}}$ *is* CTL.

Varying the parameters to $\mathsf{QNest}_{k,\mathit{Sig}}$ yields different logics, such as those in Table 4.4. If past-operators are allowed, the logic obtained is a time-symmetric extension of CTL. If the nesting depth is strictly positive, formulae in the logic have bounded sequences of modalities. Such logics lie strictly between CTL and $\mathsf{CTL}^\star$. Modal logic is the restriction of CTL with $\mathsf{EX}_\rightarrow$ as the only modality. Observe that $\mathsf{SCTL}^\star$ is the language $\bigcup_{k\in\mathbb{N}} \mathsf{SCTL}^k$.

SECTION SUMMARY This section introduced a temporal logic containing past and future linear-time modalities and a branching-time modality. The semantics of the logic was defined over trace algebras. We showed that several standard temporal logics could be derived from the new temporal logic by abstraction of syntax.

4.5 BIBLIOGRAPHIC NOTES

GRAMMARS AND FIXED POINTS This chapter used results from language theory, abstract interpretation, and lattice theory. The language-theoretic basis of the chapter is the Chomsky-Schützenberger theorem [1963], which characterises context free languages as fixed points. Ginsburg and Rice [1962] extended this characterisation to recursively enumerable languages and Istrail [1982] showed that concatenation, union, intersection and substitution on languages suffice for the fixed point characterisation. It follows from these results that the language of a BNF grammar has a fixed point characterisation. The design of meta-syntax is based on Istrail's results. See Cousot and Cousot [1992c; 1995a] or Paulson [1994] for more on inductive definitions and fixed points.

We applied abstract interpretation to approximate the fixed point characterisation of a BNF grammar. Cousot and Cousot's bi-inductive semantics [2009] has similar motivations with important differences. If we only consider the languages that can be derived, bi-inductive semantics uses posets and possibly non-monotone functions, hence is a strictly more general framework than meta-syntax. Meta-syntax resembles BNF grammars and, in our opinion, more accurately models how logics are specified in practice.

There is much work on using grammars to abstract the values of string variables for program analysis [Christensen et al. 2003; Cousot and Cousot

1995b; Okhotin and Reitwießner 2010; Wassermann et al. 2007]. Our focus is language generation from BNF, akin to invariant generation from a program, while the methods cited above are similar to analysis of a transition system. Abstract interpretation has been combined with parsing algorithms [Cousot and Cousot 2007; Schmitz 2007]. Parsing is a language recognition, not a language generation task.

TEMPORAL LOGICS Verification techniques based on temporal logic have consistently used models based on transition systems. Pnueli [1977], among several others, used Keller's model [1976] to give semantics to programs. The propositional μ-calculus [Kozen 1982] takes the middle ground, because its semantics is defined over a lattice generated by a transition system. This chapter has argued that one can generate the structural notions required to derive an abstract domain starting from the grammar of a logic. The work of Ranzato and Tapparo [2007] is closely related, allowing one to construct abstractions from the grammar of a logic.

5

PROPERTIES

> Today, some of the most interesting results in the expressiveness of modal logics rely on the notion of bisimulation. Bisimulation is indeed discovered in Modal Logic when researchers begin to investigate seriously issues of expressiveness for the logics, in the 1970s.
>
> ...
>
> Here is the discovery of bisimulation, and the choice of the name for it, in Milner's own words:
>
> ...
>
> > That same day we went for a walk in the hills around Edinburgh, and the express purpose was to agree what the pre-fixed points and the maximal fixed point should be called. We thought of a lot of words; David at one point liked "mimicry" which I vetoed. I think "bisimulation" was my suggestion; in any case, we both liked it, partly because we could use that word for the pre-fixed points and "bisimilarity" for the maximal fixed point itself. I think David demurred because there are five syllables; but we then thought that they were a lot easier to pronounce than the three syllables of "mimicry"!
>
> – Davide Sangiorgi, *On the Origins of Bisimulation and Coinduction*, 2009

This chapter presents a new notion called subsumption that characterises when two algebras satisfy the same logical properties. Subsumptions strictly generalise the notion of simulation between transition systems to various kinds of algebras. A second contribution of the chapter is a new algebraic generalisation of bisimulation called bisubsumption. The chapter presents characterisations of subsumption and bisubsumption in terms of fixed points and in terms of logics. These characterisations simplify proofs and generalise theorems concerning simulation and bisimulation.

5.1 OVERVIEW

A problem that frequently arises when reasoning about programs is whether two algebras generated by programs satisfy the same properties. For example, abstractions of transition systems replace a complex transition system by a simpler transition system. Soundness concerns mandate that every conclusion drawn by analysing the abstraction can also be drawn by analysing the concrete system. Due to imprecision, a concrete system may have properties that cannot be proved by examining the abstraction. A second example is in proving the correctness of compiler optimisation. Such proofs show that the semantics of a program is the same before and after optimisation and can be viewed as showing that two algebras are equivalent with respect to certain properties. A third example is reasoning about the semantics of concurrent processes, as studied in process algebra. Preorders such as simulation and bisimulation are used to prove that two concurrent processes (in the process algebra sense) are equivalent.

Different techniques are used to show equivalence of systems with respect to linear-time, branching-time, past-time and future-time properties. From a human perspective, the number of notions for proving that transition systems are equivalent is large and the body of results surrounding them even larger. For instance, van Glabbeek [2001] identified a lattice of 12 preorders, with 45 more possible if silent transitions are permitted [1993] and over 155 *distinct variants* if finite and infinite behaviours are differentiated. These articles do not even include the backward preorders of Lynch and Vaandrager [1995].

The question we ask in this chapter is whether there is a generalisation of notions such as simulation and bisimulation to lattices with transformers. Simulation and bisimulation provide structural characterisations of semantic notions such as property preservation. We introduce a new notion called *subsumption* that characterises when every property in a logic closed under conjunction satisfied by one algebra is satisfied by another algebra. *Additive subsumptions* are subsumptions that are additive functions. Given two algebras defined over perfect distributive lattices, additive subsumption can be used to prove that one algebra satisfies every property satisfied by the other algebra. *Bisubsumption* is an extension of subsumption that can be used to prove equivalence of algebras with respect to properties expressed in logics that contain negation. The notions of subsumption and bisubsumption are algebraic analogues of simulation and bisimulation, respectively.

The contents of this chapter are summarised in Figure 17. The dashed edges represent abstraction functions. Each box in the middle column represents a family of monotone functions. For each notion we introduce we prove a characterisation theorem showing that the notion in the middle of the figure characterises property preservation in the logic on the right of the figure. Unlike many results which depend on a fixed logic with a fixed set of modalities, our results are parameterised by the modalities in the logic.

We relate subsumption and bisubsumption to simulation and bisimulation by proving representation theorems. We show that a specific class of subsumptions provides algebraic representations of simulation relations and a specific class of bisubsumptions provides algebraic representations of bisimulation relations. We use examples to demonstrate that there are subsumptions and bisubsumptions that do not correspond to simulations and bisimulations, respectively, justifying our claim of strict generalisation.

SUMMARY OF CONCEPTS This chapter introduces the following concepts.

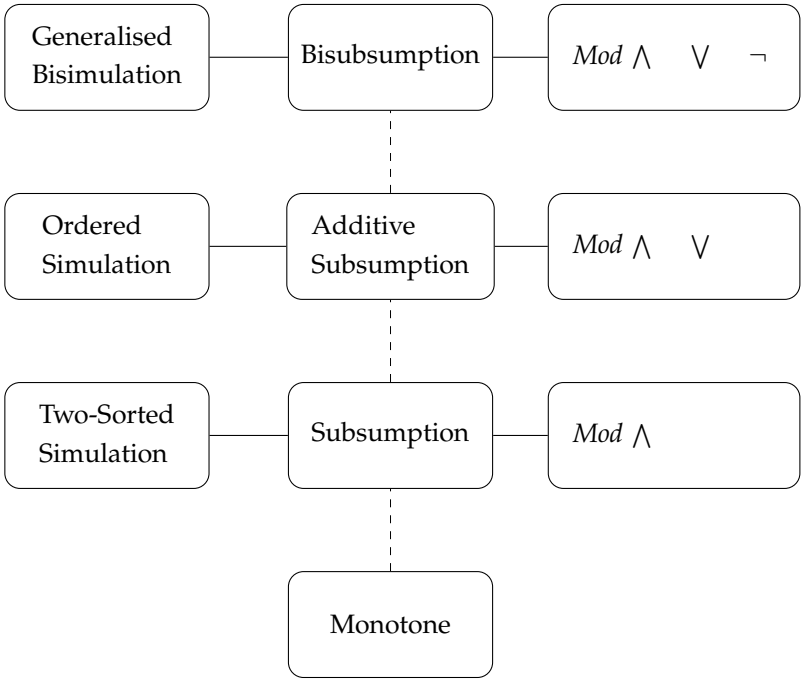


Figure 17: Chapter overview. The column in the middle represents families of monotone functions. The dashed line represents abstraction. Every bisubsumption is an additive subsumption and every additive subsumption is a subsumption. Each notion in the middle can be represented by the relation on the left and characterised by a logic on the right.

1. *Subsumption*, an algebraic generalisation of simulation, is introduced in Definition 5.3. *Stratified subsumption*, a weaker notion than subsumption, which is required to characterise property preservation in logics with finitary operators, is introduced in Definition 5.8.
2. *Bisubsumption*, an algebraic generalisation of bisimulation, is introduced in Definition 5.18. *Stratified bisubsumption*, a weakening of bisubsumption required to characterise logics with finitary, Boolean operators, is introduced in Definition 5.21.
3. *Ordered simulation*, and *two-sorted simulation*, which are generalisations of simulation to monotone and conjunctive *pre*-transition systems, respectively, are introduced in Definitions 5.36 and 5.40.

SUMMARY OF RESULTS The main results of this chapter are characterisations of subsumption and bisubsumption in terms of logics, closure operators, fixed points and relational representations.

1. Theorem 5.5 shows that subsumption characterises property preservation in logics closed under infinitary conjunction. Theorem 5.11 provides a characterisation for logics with finitary operators.
2. Theorem 5.20 is an algebraic generalisation of the Hennessy-Milner theorem and shows that bisubsumption characterises property preservation in logics closed under infinitary conjunction and negation. Theorem 5.22 provides the finitary analogue.
3. Theorems 5.16 and 5.25 show that subsumption and bisubsumption have fixed point characterisations.

4. Theorems 5.35, 5.39 and 5.43 show that additive subsumptions have representations as simulation relations, ordered simulations and two-sorted simulations, depending on the lattice involved.

CHAPTER ORGANISATION The definitions and results concerning subsumption are covered in Section 5.2 and those concerning bisubsumption are covered in Section 5.3. Section 5.4 introduces a sequence of representations for subsumptions between Boolean, distributive and conjunctive predecessor algebras. Section 5.5 applies subsumptions to prove properties of abstractions and to synthesise abstractions that preserve properties expressed in a logic.

5.2 SUBSUMPTION

This section introduces and studies subsumptions. Subsumptions strictly generalise the notion of simulation between transition systems. If there is a subsumption between the elements of two algebras, those elements satisfy the same properties in a logic closed under infinitary conjunction and monotone modalities. We give logical and fixed point characterisations of subsumptions.

Example 5.1. This example illustrates how a simulation between transition systems can be expressed as a function between lattices. Two transition systems M and N are shown in Figure 18 along with their algebraic representations $\mathcal{A}$ and $\mathcal{B}$. The arrows between lattice elements depict the operator *pre*. The relation R is a simulation. The states in every pair (s,t) in R have the same labels. Additionally, for every successor s' of s, there exists a successor t' of t such that (s',t') is in R. It is known that two states in a simulation satisfy the same ACTL* properties. The simulation R has a representation as a function $f : A \to B$ between the domains of $\mathcal{A}$ and $\mathcal{B}$. Observe that propositions map to propositions.

$$f(q^{\mathcal{A}}) = f(\{2\}) = \{4,5\} = q^{\mathcal{B}}$$

The same applies to $p^{\mathcal{A}}$. The successor condition of simulation is expressed functionally as follows. The successor of 1 is 2: $\{1\} \subseteq pre^{\mathcal{A}}(\{2\})$. Any state related to 1 has a successor related to 2.

$$f(\{1\}) = \{3\} \subseteq \{3,4\} = pre^{\mathcal{B}}(\{4,5\}) = pre^{\mathcal{B}}(f(\{2\}))$$

More generally, the successor condition of a simulation translates into the requirement that $f(x) \subseteq pre^{\mathcal{B}}(f(y))$ whenever $x \subseteq pre^{\mathcal{A}}(y)$. ⌟

Simulation is one of many preorders that admits a functional representation. For the remainder of the chapter, fix a logic L over a signature

$$Sig = Const \cup Mod \cup Bool$$

containing constant symbols, modalities and Boolean operators. Details of the operators in *Sig* will be clarified as required. Recall that a pointed algebra $(\mathcal{A},a)$ satisfies a formula φ in L, written $(\mathcal{A},a) \models \varphi$ if the inequality $a \sqsubseteq [\![\varphi]\!]_{\mathcal{A}}$ holds. Recall also that $\mathsf{L}(\mathcal{A},a)$ is the set of L-formulae satisfied by $(\mathcal{A},a)$.

Definition 5.2. A pointed algebra $(\mathcal{C},c)$ *preserves* L with respect to an algebra $(\mathcal{A},a)$ if every formula satisfied by $(\mathcal{A},a)$ is satisfied by $(\mathcal{C},c)$. That is, if $\mathsf{L}(\mathcal{A},a) \subseteq \mathsf{L}(\mathcal{C},c)$. The two algebras are L-*equivalent* if they satisfy the same formulae.

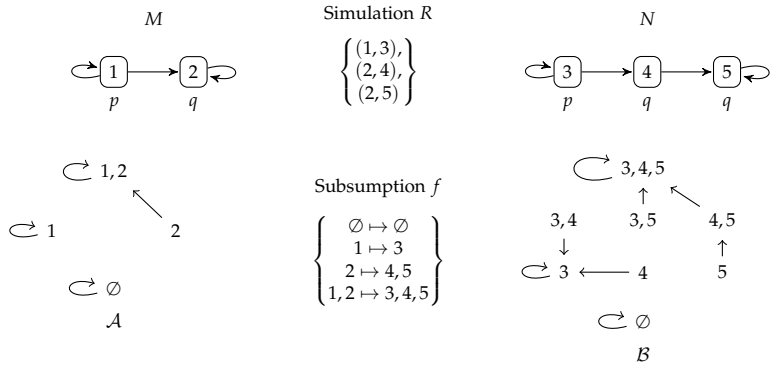


Figure 18: A simulation R between two transition systems M and N. The state algebras $\mathcal{A}$ for M and $\mathcal{B}$ for N are shown with arrows depicting *pre*. The bisimulation is represented by a function between algebras called a subsumption.

The definition of preservation above ranges over all formulae in L and is difficult to check. We say that preservation is a semantic notion because it is defined in terms of $[\![\cdot]\!]$. Subsumption provides structural characterisation of property preservation. Recall that $f\langle\bar{a}\rangle$ is the sequence obtained by applying f to each element of $\bar{a}$.

Definition 5.3. A function $f : A \to C$ is an L-*subsumption* if the following conditions hold.

1. For every constant $p \in \mathit{Const}$, $a \sqsubseteq p^{\mathcal{A}}$ implies that $f(a) \sqsubseteq p^{\mathcal{C}}$.
2. For every modality $\mathit{op} \in \mathit{Mod}$ and $\mathit{ar}(\mathit{op})$-sequence $\bar{a}$, if $a \sqsubseteq \mathit{op}^{\mathcal{A}}(\bar{a})$, it holds that $f(a) \sqsubseteq \mathit{op}^{\mathcal{C}}(f\langle\bar{a}\rangle)$.

A pointed algebra $(\mathcal{C}, c)$ *subsumes* $(\mathcal{A}, a)$ if there is a subsumption $f : A \to C$ such that $c \sqsubseteq f(a)$.

If the algebras in question are clear, we say that c *subsumes* a. The next example shows that subsumptions are generic and are easy to instantiate given the signature of a logic.

Example 5.4. We give examples of subsumptions for two state logics. The logics are interpreted over the transition systems M and N in Figure 18. Let L_1 be the logic over the signature below.

$$\mathit{Const} = \{p, q\} \quad \mathit{Mod} = \{\mathsf{EX}_{\rightarrow}\} \quad \mathit{Bool} = \{\wedge, \vee\}$$

The interpretation of $\mathsf{EX}_{\rightarrow}$ is the predecessor operator. The function f in Figure 18 was shown to satisfy the conditions of an L_1-subsumption in Example 5.1. Consider the logic L_2 over the signature below.

$$\mathit{Const} = \{p, q\} \quad \mathit{Mod} = \{\mathsf{EX}_{\leftarrow}\} \quad \mathit{Bool} = \{\wedge, \vee\}$$

The interpretation of $\mathsf{EX}_{\leftarrow}$ is the successor function. The state 5 satisfies the formulae q, $\mathsf{EX}_{\leftarrow}q$ and $\mathsf{EX}_{\leftarrow}\mathsf{EX}_{\leftarrow}p$. The state 2 satisfies these formulae as well as $\mathsf{EX}_{\leftarrow}p$, which is not satisfied by 5. Define a function $g : B \to A$ that additively extends the mappings below. The outer braces are omitted to reduce clutter.

$$\emptyset \mapsto \emptyset, \{3\} \mapsto \{1\}, \{4\} \mapsto \{2\}, \{5\} \mapsto \{2\}$$

Let us verify that g is an L_2-subsumption. We must check that constants map to constants.

$$g(q^{\mathcal{B}}) = g(\{4,5\}) = \{2\} = q^{\mathcal{A}}$$
$$g(p^{\mathcal{B}}) = g(\{3\}) = \{1\} = p^{\mathcal{A}}$$

Observe that 5 is a successor of 4: $\{5\} \subseteq post^{\mathcal{B}}(\{4\})$. The condition on operators is verified for 5 and 4 below.

$$g(\{4\}) = \{2\} \subseteq \{2\}\, post^{\mathcal{A}}(\{2\}) = post^{\mathcal{A}}(g(\{5\}))$$

The results that follow show that subsumption implies property preservation for logics without negation. ⌟

Theorem 5.5 gives a logical characterisation of subsumption. Observe that the logic is only closed under conjunction. We are not aware of a characterisation in the literature of logics closed under modalities and conjunction.

Theorem 5.5. *Let* L *be a logic closed under monotone modalities and infinitary conjunction. An element* c *subsumes* a *if and only if* $\mathsf{L}(\mathcal{A}, a) \subseteq \mathsf{L}(\mathcal{C}, c)$.

We prove each implication of the theorem separately. The proof that subsumption implies property preservation is by induction on formula structure. To show that property preservation implies subsumption, we define a function that maps an element of $\mathcal{A}$ to the greatest element of $\mathcal{C}$ satisfying the same properties. We show that this function is a subsumption.

Proof. We prove by induction that subsumption implies property preservation. Assume that there is a subsumption $f : A \to C$ with respect to which c subsumes a.

(*Base case*) Suppose that $\mathcal{A}, a \models p$ for a constant p. Satisfaction is defined by the inequality $a \sqsubseteq p^{\mathcal{A}}$. A subsumption must satisfy that $f(a) \sqsubseteq p^{\mathcal{C}}$ The assumption that c subsumes a entails that $\mathcal{C}, c \models p$.

(*Induction hypothesis*) Assume that $\mathcal{C}, f(a) \models \varphi$ follows if $\mathcal{A}, a \models \varphi$.

(*Induction step*) There are two cases to consider: modalities and infinitary conjunction. For notational simplicity, we consider unary modalities. The case for higher arities is identical.

- (*op*) Consider a formula $op(\varphi)$ such that $\mathcal{A}, a \models op(\varphi)$, where op is a unary modality. From the definition of $\models$, we know that $a \sqsubseteq op^{\mathcal{A}}([\![\varphi]\!]_{\mathcal{A}})$. For any formula $\mathcal{A}, [\![\varphi]\!]_{\mathcal{A}} \models \varphi$, so the induction hypothesis implies that $f([\![\varphi]\!]_{\mathcal{A}}) \sqsubseteq [\![\varphi]\!]_{\mathcal{C}}$. The operator $op^{\mathcal{C}}$ is monotone giving us that $op^{\mathcal{C}}(f([\![\varphi]\!]_{\mathcal{A}})) \sqsubseteq op^{\mathcal{C}}([\![\varphi]\!]_{\mathcal{C}})$. Furthermore, $c \sqsubseteq op^{\mathcal{C}}([\![\varphi]\!]_{\mathcal{C}})$ because c subsumes a, showing that $\mathcal{C}, c \models op(\varphi)$.
- ($\bigwedge$) Consider a formula $\bigwedge \bar{\varphi}$ such that $\mathcal{A}, a \models \bigwedge \bar{\varphi}$. Since $\bigwedge$ is interpreted as a meet operator we know that $a \sqsubseteq [\![\varphi_i]\!]_{\mathcal{A}}$ for each φ_i in $\bar{\varphi}$. By the induction hypothesis $f(a) \sqsubseteq [\![\varphi_i]\!]_{\mathcal{C}}$ for each such φ_i, so $f(a) \sqsubseteq \bigsqcap \{[\![\varphi_i]\!]_{\mathcal{C}} | i \in dom(\bar{\varphi})\}$. It follows that $\mathcal{C}, c \models \bigwedge \bar{\varphi}$.

In the second part of the proof we show that property preservation implies the existence of a subsumption. The function $f : A \to C$ maps $x \in A$ to the greatest element of C satisfying the same formulae.

$$f(x) \mathrel{\hat{=}} \bigsqcap \{[\![\varphi]\!]_{\mathcal{C}} | \mathcal{A}, x \models \varphi\}$$

A verification of the conditions of a subsumption is as follows.

(*Constant*) For a constant p, $f(a) \sqsubseteq p^{\mathcal{C}}$ whenever $a \sqsubseteq p^{\mathcal{A}}$, because $f(a)$ is the meet of a set containing $p^{\mathcal{C}}$.

(*Modality*) Consider an element a' for which $a \sqsubseteq op^{\mathcal{A}}(a')$. Let $\bar{\varphi}$ be the sequence (possibly transfinite) of all formulae satisfied by a' that have distinct interpretations in one of the algebras. That is, if ψ and ψ' are two formulae in $\mathsf{L}(\mathcal{A}, a')$, such that either $[\![\varphi_i]\!]_{\mathcal{A}} \neq [\![\varphi_j]\!]_{\mathcal{A}}$ or $[\![\varphi_i]\!]_{\mathcal{C}} \neq [\![\varphi_j]\!]_{\mathcal{C}}$, the formulae ψ and ψ' occur in $\bar{\varphi}$. The element a' satisfies $\bigwedge \bar{\varphi}$. The monotonicity of op entails that $a \sqsubseteq [\![op(\bigwedge \bar{\varphi})]\!]_{\mathcal{A}}$. Moreover, $f(a') = [\![\bigwedge \bar{\varphi}]\!]_{\mathcal{C}}$ because $\bar{\varphi}$ contains formulae satisfied by a' that have distinct interpretations in $\mathcal{C}$. The element a satisfies $op(\bigwedge \bar{\varphi})$ and by construction $f(a) \sqsubseteq [\![op(\bigwedge \bar{\varphi})]\!]_{\mathcal{C}}$ or equivalently, $f(a) \sqsubseteq op^{\mathcal{C}}([\![\bigwedge \bar{\varphi}]\!]_{\mathcal{C}})$. We have established that $f(a') = [\![\bigwedge \bar{\varphi}]\!]_{\mathcal{C}}$, leading to the conclusion that $f(a) \sqsubseteq op^{\mathcal{C}}(f(a'))$.

We have shown that f is a subsumption function. From the condition $\mathsf{L}(\mathcal{A}, a) \subseteq \mathsf{L}(\mathcal{C}, c)$, it follows that $c \sqsubseteq f(a)$, completing the proof. ⊣

The modalities in the logical characterisation are monotone, but the subsumption need not be monotone. The statements below identify sufficient conditions for subsumption functions to be monotone, $\bot$-strict and additive.

Corollary 5.6. *There is a subsumption between two pointed algebras if and only if there is a monotone subsumption between them.*

Proof. A monotone subsumption is a subsumption. We show that a subsumption implies the existence of a monotone one. Consider a logic L over a signature *Sig* and L' over $Sig' = Sig \cup \{\mathsf{id}\}$, where id is a modality interpreted as the identity function. An L'-subsumption f is monotone because if $a \sqsubseteq \mathsf{id}(a')$, it holds that $f(a) \sqsubseteq \mathsf{id}(f(a'))$. Observe that $\mathsf{L}(\mathcal{A}, a) \subseteq \mathsf{L}(\mathcal{C}, c)$ if and only if $\mathsf{L}'(\mathcal{A}, a) \subseteq \mathsf{L}'(\mathcal{C}, c)$. By Theorem 5.5, c L-subsumes a if and only if c L'-subsumes a. Thus, if a subsumption exists, so does a monotone one. ⊣

Proposition 5.7. *Subsumptions for a logic containing* false *are* $\bot$*-strict.*

Proof. Let $\mathcal{A}$ and $\mathcal{C}$ be two structures. The symbol false is interpreted as $\bot$ in both. Since $\bot \sqsubseteq \mathsf{false}^{\mathcal{A}}$, the subsumption satisfies that $f(\bot) \sqsubseteq \mathsf{false}^{\mathcal{C}}$, which is equivalent to asserting that $f(\bot) = \bot$. ⊣

Finitary Subsumption

The preceding results concern logics with infinitary conjunction and disjunction. Practical specification languages are finitary. Over finite lattices, finitary Boolean operators provide the same expressive power as infinitary operators. Over infinite structures, the properties that can be expressed differ. For example, if $\mathsf{EX}_{\rightarrow}$ is the only modality in a logic, reachability of a state in an infinite-state transition system can be expressed using infinitary disjunction but not with finitary disjunction. Consequently, to characterise preservation of finitary properties, a weaker notion than subsumption is required. Recall from Section 4.2 that a finite and finitary signature is completely finite. A completely finite logic is one over a completely finite signature. Stratified subsumption, which we introduce below, is a relaxation of subsumption for completely finite logics.

Definition 5.8. A sequence of n functions $\bar{f}$ in $A \to C$ is an *n-stratified subsumption* if $f_{i+1} \sqsubseteq f_i$ for all $i < n$ and if the conditions below hold.

1. For every $p \in Const$ and $a \in A$, $a \sqsubseteq p^{\mathcal{A}}$ implies that $f_0(a) \sqsubseteq p^{\mathcal{C}}$.
2. For every modality op, element a and $ar(op)$-sequence $\bar{a}$, if $a \sqsubseteq op^{\mathcal{A}}(\bar{a})$, it holds that $f_n(a) \sqsubseteq op^{\mathcal{C}}(f_{n-1}\langle \bar{a} \rangle)$.

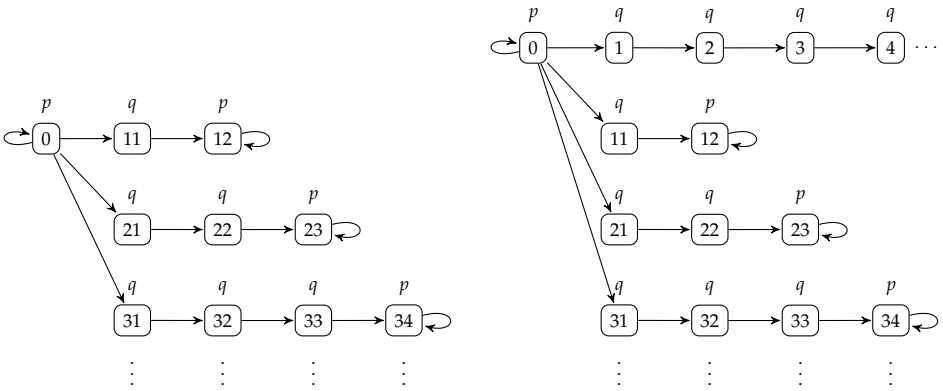


Figure 19: Two infinite state transition systems. M only has finite paths labelled q whereas N has an infinite path labelled q.

The algebra $(\mathcal{C}, c)$ *subsumes* $(\mathcal{A}, a)$ *up to depth* n if there is an n-stratified subsumption $\bar{f}$ such that $c \sqsubseteq f_{n-1}(a)$.

Example 5.9. A standard example illustrating the difference between finitary and infinitary modal logic is recalled below. Consider the transition systems M and N in Figure 19 and a logic with the next-state modality $\mathsf{EX}_{\rightarrow}$, infinitary conjunction and without negation. Every property satisfied by M is satisfied by N but every property satisfied by N is not satisfied by M. The transition system N has an infinite path labelled q, so the state 0 satisfies the formula

$$\bigwedge_{i \in \mathbb{N}} \mathsf{EX}^i_{\rightarrow} q$$

but no state in M satisfies this formula.

We illustrate a subsumption from M to N. The function mapping each state in M to the identically numbered state in N is a subsumption. The function mapping each state in N to the identically numbered state in M, if such a number exists, is not a subsumption because 1 is not subsumed by any state in M, so 0 in N is not subsumed by 0 in M.

The situation changes if only finitary conjunction is permitted. Any formula containing n nested modalities that is satisfied by 1 is also satisfied by the state labelled $(n+1)1$ in M. For every nesting depth n, there is a state in M that satisfies the same formulae as 1. The function that maps 1 to the state $(n+1)1$ is a stratified subsumption of depth n. ⌟

Stratified subsumptions and property preservation are related in terms of the modal depth of formulae. The depth of a symbol from a set S in a formula φ is denoted $dep(S, \varphi)$ and defined in Section 4.2. The *modal depth* of a formula $mdep(\varphi) \triangleq dep(Mod, \varphi)$ is the depth of a modal operator in φ. Define the set $\mathsf{L}_n(\mathcal{A}, a)$ that contains all formulae φ satisfied by a such that $mdep(\varphi) \leq n$. A stratified subsumption of fixed depth characterises property preservation for formulae of bounded modal depth. The characterisation requires the following lemma about completely finite logics.

Lemma 5.10. *In a logic* L *over a completely finite signature, the set of formulae of bounded modal depth with pairwise distinct interpretations is finite.*

Proof. We need to show that for every n, the set of formulae $D \subseteq \mathsf{L}_n$, such that $[\![\varphi]\!]_{\mathcal{A}} \neq [\![\psi]\!]_{\mathcal{A}}$ for every pair in D is finite. We use induction on modal depth.

(*Base case*) The claim holds for formulae with no modal operators because there are only finitely many distinct Boolean combinations of a finite set of constants.

(*Induction hypothesis*) Assume that there are finitely many distinct formulae of modal depth at most n.

(*Induction step*) A formula of depth $n+1$ is of the form $op(\bar{\varphi})$, where op is either a modality or a Boolean operator. If op is a modality, the formulae in $\bar{\varphi}$ are in L_n. By the induction hypothesis, there are finitely many distinct interpretations of formulae in L_n, so the same holds for L_{n+1}. If op is a Boolean operator, one or more formulae in $\bar{\varphi}$ are in L_{n+1}. The set of modalities is finite, so by the induction hypothesis, there are finitely many formulae $op_i(\varphi_i)$ in $\bar{\varphi}$ with distinct interpretations. Consequently, finitely many distinct formulae have distinct interpretations in L_{n+1}. ⊣

Theorem 5.11. *For a completely finite logic* L *closed under monotone modalities and binary conjunction,* $\mathsf{L}_n(\mathcal{A}, a) \subseteq \mathsf{L}_n(\mathcal{C}, c)$ *if and only if* $(\mathcal{C}, c)$ *subsumes* $(\mathcal{A}, a)$ *up to depth* n*.*

The proof that a stratified subsumption implies property preservation is by induction. To prove that property preservation implies there is a subsumption function, we define a function and show that it is a stratified subsumption.

Proof. The proof that a stratified subsumption implies inclusion of formulae of bounded depth follows the proof of Theorem 5.5 and is omitted. To prove that property preservation implies subsumption, we construct a subsumption function. The construction differs from that used in Theorem 5.5 because the logic lacks infinitary conjunction. For notational simplicity, only unary modalities are considered.

$$f_0(x) \mathrel{\hat{=}} \sqcap \left\{ p^{\mathcal{C}} \mid \mathcal{A}, x \models p \right\}$$
$$f_{i+1}(x) \mathrel{\hat{=}} \sqcap \left\{ op^{\mathcal{C}}(f_i([\![\varphi]\!]_{\mathcal{A}}) \mid \mathcal{A}, x \models op(\varphi) \text{ and } 0 \leq dep(\varphi) < i \right\}$$

We show that $f_n, \ldots, f_0$ is a subsumption of depth n. By Lemma 5.10 the sets of elements in the construction above are finite. The function f_0 is a 0-subsumption by definition. Consider a and a' such that $a \sqsubseteq op^{\mathcal{A}}(a')$. Let $\bar{\varphi}$ be a maximal sequence of formulae that have depth at most n, have distinct interpretations in one of the algebras and are satisfied by a'. This sequence is finite and satisfies that $a \sqsubseteq [\![\wedge\bar{\varphi}]\!]_{\mathcal{A}}$. By assumption that c satisfies the same properties as a, $c \sqsubseteq [\![\wedge\bar{\varphi}]\!]_{\mathcal{C}}$. Since $f_n(a')$ is $[\![\wedge\bar{\varphi}]\!]_{\mathcal{C}}$, the sequence $\bar{f}$ is a stratified subsumption. ⊣

Fixed Point Characterisation

We show that the set of subsumptions is a complete lattice: it has least upper bound and greatest lower bound operators and importantly, there exists a greatest subsumption between two algebras. To establish these properties, we exhibit a closure operator that maps from functions between algebras to subsumptions. Consider two temporal algebras $\mathcal{A}$ and $\mathcal{C}$. The order and operations on C lift pointwise to functions in $A \to C$. An operator on $A \to C$

maps a function to a function. The operators below transform a function f to the greatest $g \sqsubseteq f$ that is monotone or is a subsumption.

$$mono(f) \mathrel{\hat{=}} \bigsqcup \{g \sqsubseteq f \mid g \text{ is monotone}\}$$
$$sub(f) \mathrel{\hat{=}} \bigsqcup \{g \sqsubseteq f \mid g \text{ is a subsumption }\}$$

Lemma 5.12. *Let f be a function in $A \to C$. The following properties hold.*
1. *$mono(f)$ is monotone.*
2. *$sub(f)$ is a subsumption.*

Proof. Let $F \subseteq A \to C$ be a set of functions and g be $\bigsqcup F$. Monotone functions are known to be closed under arbitrary joins. Assume F is a set of subsumptions. For each $f \in F$ and $a \sqsubseteq p^{\mathcal{A}}$, it holds that $f(a) \sqsubseteq p^{\mathcal{C}}$, so $\bigsqcup \{f(a) \mid f \in F\} \sqsubseteq p^{\mathcal{A}}$. It follows that $g(a) \sqsubseteq p^{\mathcal{A}}$. Each subsumption $f \in F$ satisfies that $f(a) \sqsubseteq op^{\mathcal{C}}(f\langle \bar{a} \rangle)$ if $a \sqsubseteq op^{\mathcal{A}}(\bar{a})$. The operator $op^{\mathcal{C}}$ is monotone, so $f(a) \sqsubseteq op^{\mathcal{C}}(g\langle \bar{a} \rangle)$. Since upper bounds are preserved by joins, $g(a) \sqsubseteq op^{\mathcal{C}}(g\langle \bar{a} \rangle)$. ⊣

The operator *mono* will map a monotone function to itself. Monotone functions are fixed points of *mono*. Moreover, the image of $A \to C$ under *mono* contains all monotone functions.

Lemma 5.13. *The operators mono and sub are lower closure operators on $A \to C$.*

Proof. A lower closure operator is reductive, idempotent and monotone. Let *op* be an operator in the lemma statement, f be a function and $op(f)$ be of the form $\bigsqcup F$.

(*Reductivity*) Every g in F is less than f, so $\bigsqcup F \sqsubseteq f$ and $op(f) \sqsubseteq f$.

(*Idempotence*) Assume f is equal to $op(g)$ for some function g. If f is equal to a function $op(g)$ for some g, the set F contains f. Idempotence of *op* follows because $op(f) = op(op(g)) = f$.

(*Monotonicity*) Consider g such that $f \sqsubseteq g$. The function $op(g)$ is of the form $\bigsqcup G$ with F being a subset of G. Monotonicity follows. ⊣

Ward's theorem states that the image of a complete lattice under a closure operator is a complete lattice. The functions generated by aforementioned operators form a complete lattice.

Corollary 5.14. *Between a pair of algebras, the sets of functions that are monotone and those that are subsumptions form a complete lattice.*

Consider the function f that maps all elements to $\top$. The functions $sub(f)$ and $bisub(f)$ are the greatest subsumption and bisubsumption, respectively. The greatest simulation has a fixed point characterisation that is used to compute simulations. Similarly, the preceding closure operators have fixed point characterisations. The operator on $A \to C$ below is used to give a characterisation of monotone functions.

$$imono(f) \mathrel{\hat{=}} \left\{a \mapsto \bigsqcap \{f(a') \mid a \sqsubseteq a'\}\right\}$$

Theorem 5.15. *A function f is monotone if and only if $f \sqsubseteq imono(f)$. The greatest monotone function $g \sqsubseteq f$ is $\mathsf{gfp}(\lambda x.(imono(x) \sqcap f))$.*

A fixed point characterisation of subsumptions follows. The operators below are unary on $A \to C$.

$$ic(f) \mathrel{\hat{=}} \left\{a \mapsto \bigsqcap \left\{p^{\mathcal{C}} \mid a \sqsubseteq p^{\mathcal{A}}\right\}\right\}$$
$$iop(f) \mathrel{\hat{=}} \left\{a \mapsto \bigsqcap \left\{op^{\mathcal{C}}(f\langle\bar{a}\rangle) \mid a \sqsubseteq op^{\mathcal{A}}(\bar{a})\right\}\right\}$$
$$isub \mathrel{\hat{=}} ic \sqcap iop$$

Theorem 5.16. *A subsumption is a function satisfying that $f \sqsubseteq isub(f)$. The greatest subsumption is* $\mathsf{gfp}(isub)$.

Proof. We prove that a subsumption f is less than its image under *isub*. There are two cases: the functions *ic* and *iop*.

(*ic*) If $a \sqsubseteq p^{\mathcal{A}}$, because f is a subsumption, $f(a) \sqsubseteq p^{\mathcal{C}}$. Taking all such propositions, $f(a) \sqsubseteq \bigsqcap \{p^{\mathcal{C}} \mid a \sqsubseteq p^{\mathcal{A}}\}$. When lifted to functions, this condition is $f \sqsubseteq ic(f)$.

(*iop*) For every sequence $\bar{a}$ and modality *op*, $a \sqsubseteq op(\bar{a})$ implies that $f(a) \sqsubseteq op(f\langle\bar{a}\rangle)$. Thus, $f(a) \sqsubseteq \bigsqcap \{op(f(\bar{a})) \mid a \sqsubseteq op(\bar{a})\}$ and, lifting the order to functions, $f \sqsubseteq iop(f)$.

Combining the two cases, $f \sqsubseteq isub(f)$ if f is a subsumption. Conversely, suppose that $f \sqsubseteq g$, where $g = isub(f)$ and f is an arbitrary function. The subsumption conditions are to be verified.

(*Proposition*) If $a \sqsubseteq p^{\mathcal{A}}$, the definition of $ic(f)$ entails that $ic(f)(a) \sqsubseteq p^{\mathcal{C}}$, so $g(a) \sqsubseteq p^{\mathcal{C}}$, implying that $f(a) \sqsubseteq p^{\mathcal{C}}$.

(*Modality*) Consider a sequence $\bar{a}$ and modality *op* such that $a \sqsubseteq op(\bar{a})$. The definition of *iop* entails that $iop(f)(a) \sqsubseteq op(f\langle\bar{a}\rangle)$, thus $g(a) \sqsubseteq op(f\langle\bar{a}\rangle)$, in turn guaranteeing that $f(a) \sqsubseteq op(f\langle\bar{a}\rangle)$.

A function f is a subsumption if and only if $f \sqsubseteq isub(f)$. For a greatest fixed point to exist, *isub* must be monotone. The function *ic* is monotone. Consider $g \sqsubseteq f$ and the functions $iop(g)$ and $iop(f)$. All modalities are monotone implying that $iop(g)(a) \sqsubseteq iop(f)(a)$ and by pointwise lifting $iop(g) \sqsubseteq iop(f)$. The meet of monotone functions is monotone. The Knaster-Tarski theorem applies and shows that $\mathsf{gfp}(isub)$ exists and is the greatest subsumption. ⊣

Fixed point characterisations have several applications. The next example applies Theorem 5.16 to calculate a subsumption.

Example 5.17. The algebras $\mathcal{A}$ and $\mathcal{B}$ from Figure 18 are used here. The value of $ic(f)$ for every function $f : A \to B$ is shown below.

$$\{\emptyset \mapsto \emptyset, \{1\} \mapsto \{3\}, \{2\} \mapsto \{4,5\}, \{1,2\} \mapsto \{3,4,5\}\}$$

Let $f_0 : A \to B$ be the function that maps all elements of A to $\{3,4,5\}$. Observe that $pre^{\mathcal{B}}$ maps $\{3,4,5\}$ to itself. As a result, the function $iop(f_0)$ maps $\emptyset$ to $\emptyset$ and all other elements of A to $\{3,4,5\}$. The function $f_1 \mathrel{\hat{=}} ic(f_0) \sqcap iop(f_0)$ is identical to $ic(f_0)$. Moreover f_1 is a fixed point of $isub(f_1)$ and is, in fact, the subsumption in Figure 18. ⌟

SECTION SUMMARY This section introduced subsumption. Subsumption is a strict generalisation of simulation to algebras. We proved a Hennessy-Milner theorem, showing that subsumption characterises property preservation in logics closed under monotone modalities and infinitary conjunction. We also gave a characterisation of property preservation in logics with finite conjunction and gave a fixed point characterisation of subsumption.

5.3 BISUBSUMPTION

Subsumption characterises property preservation for monotone logics. In a logic closed under negation, one can express that a property is not satisfied by an algebra. Property preservation requires showing that $\mathsf{L}(\mathcal{A}, a) \subseteq \mathsf{L}(\mathcal{C}, c)$ for properties without a leading negation and that $\mathsf{L}(\mathcal{C}, c) \subseteq \mathsf{L}(\mathcal{A}, a)$ for properties with a leading negation. We now extend subsumption to characterise property preservation concerning logics with negation.

When dealing with transition systems, proofs of property preservation for logics closed under negation use bisimulation. A bisimulation is a simulation whose inverse is also a simulation. The simulation and inverse simulation allow one to move 'back-and-forth' between reasoning about one transition system and about the other. The challenge in lifting bisimulation to algebras is in identifying an algebraic criterion that models the inverse condition in bisimulation. We use conjugate functions to capture the back-and-forth nature of bisimulation. By using conjugate functions, we assume that the algebras involved are defined over perfect Boolean lattices.

Definition 5.18. A function $f : A \to C$ is a *bisubsumption* if f is a strict additive subsumption and the conjugate of f is a subsumption. Two elements a and c are in a bisubsumption if there exists a bisubsumption f with a conjugate b such that $c \sqsubseteq f(a)$ and $a \sqsubseteq b(c)$.

Example 5.19. We revisit the transition systems in Figure 18. Observe that the inverse of R is also a simulation. The conjugate $b : C \to A$ of f is the additive extension of the mapping below.

$$\emptyset \mapsto \emptyset, \{3\} \mapsto \{1\}, \{4\} \mapsto \{2\}, \{5\} \mapsto \{2\}$$

See that $f(\{1\}) = \{3\}$, so $f(\{1\}) \cap \{4,5\} = \emptyset$ and conversely, $b(\{4,5\}) = \{2\}$, so $\{1\} \cap b(\{4,5\}) = \emptyset$, as required of conjugate functions. In addition, b is a subsumption for a logic with the modality $\mathsf{EX}_{\rightarrow}$.

For contrast, consider the logic L_2 and subsumption g defined in Example 5.4. The only modality is $\mathsf{EX}_{\leftarrow}$. The conjugate of g is

$$h \mathrel{\hat{=}} \{\emptyset \mapsto \emptyset, \{1\} \mapsto \{3\}, \{2\} \mapsto \{4,5\}, \{1,2\} \mapsto \{3,4,5\}\}$$

To see that h is not a subsumption, consider $\{2\}$.

$$\{2\} \subseteq \mathit{post}^{\mathcal{A}}(\{2\}) \text{ but } h(\{2\}) = \{4,5\} \not\subseteq \mathit{post}^{\mathcal{A}}(h(\{2\})) = \{5\}$$

In fact, if the logic L_2 is extended with negation, $\{5\}$ satisfies the formula $\neg\mathsf{EX}_{\leftarrow} p$, which is not satisfied by $\{2\}$. ⌟

Theorem 5.20. *Let* L *be a logic closed under monotone modalities, infinitary conjunction and negation. Two pointed temporal algebras* $(\mathcal{A}, a)$ *and* $(\mathcal{C}, c)$ *are* L*-equivalent if and only if* a *and* c *are in a bisubsumption.*

The proof that a bisubsumption implies logical equivalence is by structural induction on formulae. The proof is similar to that of Theorem 5.5 with an additional case for negation. To prove the two elements satisfying the same formulae are in a bisubsumption, we construct a two subsumption functions and show they are conjugate.

Proof. Assume that there is a bisubsumption $f : A \to C$ with a conjugate $b : C \to A$. Consider two elements $a \in A$ and $c \in C$. We prove that $\mathcal{A}, a \sqcap b(c) \models \varphi$ if and only if $\mathcal{C}, f(a) \sqcap c \models \varphi$. If a and c are in a bisubsumption, it follows that $\mathsf{L}(\mathcal{A}, a) = \mathsf{L}(\mathcal{C}, c)$.

(*Base case*) Consider a proposition p such that $a \sqcap b(c) \models p$. Then, $a \sqcap b(c) \sqsubseteq p^{\mathcal{A}}$ and because f is a subsumption, $f(a \sqcap b(c)) \sqsubseteq p^{\mathcal{C}}$. Applying the Dedekind law for conjugate functions in Theorem 2.24 yields that $f(a) \sqcap c \sqsubseteq f(a \sqcap b(c))$ and consequently $f(a) \sqcap c \models p$. The case for $f(a) \sqcap c \models p$ is symmetric.

(*Induction hypothesis*) Assume that $\mathcal{A}, a \sqcap b(c) \models \varphi$ if and only if $\mathcal{C}, f(a) \sqcap c \models \varphi$ for every formula φ.

(*Induction step*) There are three cases: a monotone operator, infinitary conjunction and negation. The following inequality is useful. The inequality $[\![\varphi]\!]_{\mathcal{A}} \sqcap b(\top) \sqsubseteq [\![\varphi]\!]_{\mathcal{A}}$ holds and implies $f([\![\varphi]\!]_{\mathcal{A}}) \sqsubseteq [\![\varphi]\!]_{\mathcal{C}}$ via the induction hypothesis.

(*op*) Consider a formula $op(\varphi)$, where op is monotone. A unary operator is considered for notational simplicity. The general case is identical. Suppose that $a \sqcap b(c) \models op(\varphi)$.

$$
\begin{aligned}
& a \sqcap b(c) \sqsubseteq op^{\mathcal{A}}([\![\varphi]\!]_{\mathcal{A}}) \\
\implies & f(a \sqcap b(c)) \sqsubseteq op^{\mathcal{C}}(f([\![\varphi]\!]_{\mathcal{A}})), \text{ because } f \text{ is a subsumption} \\
\implies & f(a) \sqcap c \sqsubseteq op^{\mathcal{C}}(f([\![\varphi]\!]_{\mathcal{A}})), \text{ by the Dedekind law} \\
\implies & f(a) \sqcap c \sqsubseteq op^{\mathcal{C}}([\![\varphi]\!]_{\mathcal{C}}), \text{ by the inequality above} \\
\implies & f(a) \sqcap c \models op(\varphi), \text{ for one direction.}
\end{aligned}
$$

The converse is similar.

($\bigwedge$) Consider a formula $\bigwedge \bar{\varphi}$. If $a \sqcap b(c) \models \bigwedge \bar{\varphi}$, then $a \sqcap b(c) \models \varphi_i$, for every φ_i in $\bar{\varphi}$. By applying the induction hypothesis, if $f(a) \sqcap c \models \varphi_i$ for every such φ_i it follows that $f(a) \sqcap c \models \bigwedge \bar{\varphi}$.

($\neg$) An alternative formulation of the induction hypothesis is that $a \sqcap b(c) \sqcap [\![\psi]\!]_{\mathcal{A}} = \bot$ if and only if $f(a) \sqcap c \sqcap [\![\psi]\!]_{\mathcal{C}} = \bot$ for every ψ. Assume that $a \sqcap b(c)$ satisfies the formula $\neg\varphi$. That is, $a \sqcap b(c) \sqsubseteq \neg[\![\varphi]\!]_{\mathcal{A}}$ or equivalently $a \sqcap b(c) \sqcap [\![\varphi]\!]_{\mathcal{A}} = \bot$. The alternative formulation of the induction hypothesis applies to $[\![\varphi]\!]_{\mathcal{A}}$, showing that $f(a) \sqcap c \sqcap [\![\varphi]\!]_{\mathcal{C}} = \bot$ and consequently, $f(a) \sqcap c \models \neg\varphi$.

We have shown that a bisubsumption implies logical equivalence.

The next step is to show that logical equivalence implies bisubsumption. Assume that $\mathsf{L}(\mathcal{A}, a) = \mathsf{L}(\mathcal{C}, c)$ for elements a and c. Define the functions $f : A \to C$ and $g : C \to A$ as follows.

$$f(x) \mathrel{\hat{=}} \bigsqcap \{[\![\varphi]\!]_{\mathcal{C}} | \mathcal{A}, x \models \varphi\} \text{ and } b(y) \mathrel{\hat{=}} \bigsqcap \{[\![\varphi]\!]_{\mathcal{A}} | \mathcal{C}, y \models \varphi\}$$

We show that f is a bisubsumption. The proof that f and b are subsumptions is identical to the proof of Theorem 5.5. This is because negation is not involved in the definition of a subsumption function. It remains to be shown that f and b are conjugate. Consider $x \in A$ and $y \in C$ such that $f(x) \sqcap y = \bot$. The definition of f expands to $\bigsqcap \{[\![\varphi]\!]_{\mathcal{C}} | \mathcal{A}, x \models \varphi\} \sqcap y = \bot$. Let $\bar{\varphi}$ contain formulae that have pairwise distinct interpretations in one of $\mathcal{A}$ or $\mathcal{C}$. Replacing $\bigsqcap$ with infinitary conjunction leads to the equality $[\![\bigwedge \bar{\varphi}]\!]_{\mathcal{C}} \sqcap y = \bot$ and its equivalent $y \sqsubseteq [\![\neg \bigwedge \bar{\varphi}]\!]_{\mathcal{C}}$. From the definition of g, it follows that $b(y) \sqsubseteq [\![\neg \bigwedge \bar{\varphi}]\!]_{\mathcal{A}}$. Since $\bar{\varphi}$ contains formulae satisfied by x, $x \sqsubseteq [\![\bigwedge \bar{\varphi}]\!]_{\mathcal{A}}$ leading to the conclusion that $x \sqcap b(y) = \bot$. The case for $x \sqcap b(y) = \bot$ is symmetric. Thus, f is a bisubsumption. By the assumption that $\mathsf{L}(\mathcal{A}, a) = \mathsf{L}(\mathcal{C}, c)$ and from the definition of f and g, we have that $c \sqsubseteq f(a)$ and $a \sqsubseteq b(c)$, so a and c are in a bisubsumption. ⊣

Finitary Bisubsumption

Bisubsumptions have conjugates. It is immediate that they are bottom strict, additive and preserve infinitary disjunction. Logical equivalence with respect to completely finite logics closed under conjunction and negation is characterised by stratified bisubsumption. The structure of these definitions and the proofs that we require next closely resembles those for finitary subsumption, so we keep the section short.

Definition 5.21. Let $\bar{f}$ be an $n+1$-termed sequence of strict additive functions in $A \to C$ and let $\bar{b}$ be a sequence of functions in $C \to A$ such that b_i is the conjugate of f_i for each $i \in dom(\bar{f})$. The sequence $\bar{f}$ is a *stratified bisubsumption of depth n* if $\bar{f}$ and $\bar{b}$ are both stratified subsumptions of depth n.

Two elements a and c are in an n-stratified bisubsumption if there is an n-stratified bisubsumption $\bar{f}$ with corresponding conjugates $\bar{b}$ such that $a \sqsubseteq b_n(c)$ and $c \sqsubseteq f_n(a)$.

Theorem 5.22. *For a completely finite logic* L *closed under monotone modalities, binary conjunction and negation,* $\mathsf{L}_n(\mathcal{A}, a) = \mathsf{L}_n(\mathcal{C}, c)$ *if and only if c and a are in an n-stratified bisubsumption.*

The proof is similar to that of Theorems 5.20 and 5.11. Lemma 5.10 is required to show that for every syntactic depth, there are only finitely many semantically distinct formulae.

Fixed Point Characterisation

We give fixed point characterisations of bisubsumption. Consider the temporal algebras $\mathcal{A}$ and $\mathcal{C}$ as before.

$$bisub(f) \mathrel{\hat{=}} \bigsqcup \{ g \sqsubseteq f \mid g \text{ is a bisubsumption } \}$$

Lemma 5.23. *If f is a function in $A \to C$ then $bisub(f)$ is a bisubsumption.*

Proof. Let $F \subseteq A \to C$ be a set of functions and g be $\bigsqcup F$. Assume F is a set of bisubsumptions, B, the conjugates of functions in F, g be $\bigsqcup F$ and h be $\bigsqcup B$. We show that g is a bisubsumption. If $g(x) \sqcap y = \bot$, for every $f \in F$ with conjugate $b \in B$, it holds that $f(x) \sqcap y = \bot$, and that $x \sqcap b(y) = \bot$. It further holds that $\bigsqcup \{x \sqcap b(y) \mid b \in B\} = \bot$. Equivalently, by distributivity in a Boolean algebra, $x \sqcap \bigsqcup \{b(y) \mid b \in B\} = \bot$, leading to the conclusion that $x \sqcap h(y) = \bot$. Thus, g and h are conjugate. ⊣

Lemma 5.24. *The operator bisub is a lower closure operator on $A \to C$.*

The fixed point characterisations provided earlier extend to bisubsumption.

$$ibc(f) \mathrel{\hat{=}} \left\{ a \mapsto \bigsqcap \left\{ p^{\mathcal{C}} \mid a \sqsubseteq p^{\mathcal{A}} \right\} \cup \left\{ \neg q^{\mathcal{C}} \mid a \sqsubseteq \neg q^{\mathcal{A}} \right\} \right\}$$

$$ibop(f) \mathrel{\hat{=}} \left\{ a \mapsto \bigsqcap X_a \cup Y_a \right\}, \text{ where}$$

$$X_a \mathrel{\hat{=}} \left\{ op^{\mathcal{C}}(f(a')) \mid a \sqsubseteq op^{\mathcal{A}}(a') \right\}, \text{ and}$$

$$Y_a \mathrel{\hat{=}} \left\{ \neg op^{\mathcal{C}}(f(a')) \mid a \sqsubseteq \neg op^{\mathcal{A}}(a') \right\}$$

$$ibisub \mathrel{\hat{=}} ic \sqcap iop$$

Theorem 5.25. *A function f is a bisubsumption if and only if $f \sqsubseteq ibisub(f)$. The greatest subsumption is* $\mathsf{gfp}(ibisub)$.

The proof is similar to previous proofs and is omitted.

SECTION SUMMARY This section introduced bisubsumption, an extension of subsumption to characterise logics with negation. Unlike subsumption, bisimulation is defined using conjugate functions and is defined only over perfect Boolean lattices. The algebraic characterisation does generalise beyond the standard setting of modal logic or Hennessy-Milner logic to any logic with completely additive modalities that is also closed under Boolean operations. We provided a logical and a fixed point characterisation of bisubsumption as a parallel to the known results about bisimulation.

5.4 REPRESENTATION THEOREMS

In the previous sections we claimed without proof that subsumption and bisubsumption generalise simulation and bisimulation. In this section, we prove that certain subsumptions have representations that resemble simulation relations between transition systems. We show how simulation and bisimulation can be instantiated using subsumption and bisubsumption, justifying our claim of generalisation.

Logics interpreted over a perfect distributive lattices support infinitary disjunction. When dealing with transition systems, simulation relations are used to prove that two transition systems satisfy the same properties in a logic that is closed under conjunction and disjunction but not under negation. Moreover the modalities in the logic distribute over disjunction. Additive subsumptions strictly generalise simulation to structures over perfect distributive lattices such as monotone state algebras.

Definition 5.26. A function $f : A \to C$ is an *additive* L*-subsumption* if f is an L subsumption and satisfies the equality $f(\bigsqcup S) = \bigsqcup f(S)$ for every subset $S \subseteq A$. A pointed algebra $(\mathcal{C}, c)$ *additively subsumes* $(\mathcal{A}, a)$ if there is an additive subsumption $f : A \to C$ such that $c \sqsubseteq f(a)$.

Additive subsumptions are required to characterise property preservation in logics closed under arbitrary disjunction. Such logics are defined below.

Definition 5.27. A *completely disjunctive* logic L satisfies the conditions below.

1. For every sequence of formulae $\bar{\varphi}$, the formula $\bigvee \bar{\varphi}$ is in L.
2. For every n-ary modality op and sequences of formulae $\bar{\varphi}$, $\bar{\psi}$ and $\bar{\theta}$ satisfying $len(\bar{\varphi}) + len(\bar{\varphi}) = n - 1$, the equivalence below holds.

$$op(\bar{\varphi}, \bigvee \bar{\theta}, \bar{\psi}) \Leftrightarrow \bigvee_{i=0}^{len(\bar{\theta})} op(\bar{\varphi}, \theta_i, \bar{\psi})$$

Theorem 5.28 extends the characterisation provided by subsumption to completely disjunctive logics.

Theorem 5.28. *If* L *is a completely disjunctive logic and there exists a completely additive* L*-subsumption from* $(\mathcal{A}, a)$ *to* $(\mathcal{C}, c)$*, then* $\mathsf{L}(\mathcal{A}, a) \subseteq \mathsf{L}(\mathcal{C}, c)$.

Proof. Let f be a completely additive subsumption from $(\mathcal{A}, a)$ to $(\mathcal{C}, c)$ such that c subsumes a. We proceed by induction.

(*Base case*) The case for constants follows from Theorem 5.5.

(*Induction hypothesis*) Assume that $\mathcal{C}, f(a) \models \varphi$ follows if $\mathcal{A}, a \models \varphi$.

(*Induction step*) The case for modalities and conjunction is as in Theorem 5.5. We consider disjunction. Suppose that $\mathcal{A}, a \models \bigvee \bar{\varphi}$. The definition of $\models$ expands the assumption to $a \sqsubseteq [\![\bigvee \bar{\varphi}]\!]_{\mathcal{A}}$. The subsumption f is monotone, implying that $f(a) \sqsubseteq f([\![\bigvee \bar{\varphi}]\!]_{\mathcal{A}})$. The interpretation of $\bigvee$ yields

that $f(a) \sqsubseteq f(\bigsqcup\{[\![\varphi_i]\!]_{\mathcal{A}} \mid i \in dom(\bar{\varphi})\})$. From the additivity assumption, $f(a) \sqsubseteq \bigsqcup\{f([\![\varphi_i]\!]_{\mathcal{A}}) \mid i \in dom(\bar{\varphi})\}$ and by applying the induction hypothesis, $f(a) \sqsubseteq \bigsqcup\{[\![\varphi_i]\!]_{\mathcal{C}} \mid i \in dom(\bar{\varphi})\}$. We conclude that $\mathcal{C}, c \models \bigvee \bar{\varphi}$. ⊣

Additive subsumptions are a subset of all subsumptions, so it is not obvious that additive subsumptions also form a complete lattice. We give a closure operator characterisation of additive subsumptions below. Consider the algebras $\mathcal{A}$ and $\mathcal{C}$ as before. The operator below transforms a function f to the greatest $g \sqsubseteq f$ that is additive, strict additive or is a subsumption, respectively.

$$add(f) \mathrel{\hat{=}} \bigsqcup\{g \sqsubseteq f \mid g \text{ is completely additive}\}$$
$$sadd(f) \mathrel{\hat{=}} \bigsqcup\{g \sqsubseteq f \mid g \text{ is } \bot\text{-strict and completely additive}\}$$
$$sub(f) \mathrel{\hat{=}} \bigsqcup\{g \sqsubseteq f \mid g \text{ is a subsumption }\}$$

Lemma 5.29. *Let f be a function in $A \to C$. The following properties hold.*
1. *$add(f)$ is completely additive.*
2. *$sadd(f)$ is $\bot$-strict and completely additive.*

Proof. Let $F \subseteq A \to C$ be a set of functions and g be $\bigsqcup F$. Additive functions are known to be closed under arbitrary joins. The join of additive functions is additive. The elements of F are $\bot$-strict. It follows that $g(\bot) = \bigsqcup\{f(\bot) \mid f \in F\}$ is $\bot$. ⊣

Lemma 5.30. *The operators add and sadd are lower closure operators on $A \to C$.*

The proof is identical to the proof for monotone functions and subsumptions. It follows that the set of additive subsumptions is a complete lattice. In the remainder of the section we identify relational representations of additive subsumptions. Assume for the rest of this section that $\mathsf{EX}_{\to}$ is the only modality in the logic we consider.

Simulation and Bisimulation

We show that simulation relations generate additive subsumptions between Boolean predecessor algebras and that additive subsumptions between such algebras generate simulation relations. The standard definition of simulation, with an extension for state and transition labels is recalled below.

Definition 5.31. Let $M_1 = (S_1, E_1, prop_1, act_1)$ and $M_2 = (S_2, E_2, prop_2, act_2)$ be labelled transition systems. A relation $Sim \subseteq S_1 \times S_2$ is a *labelled simulation* if every (r, s) in Sim satisfy the conditions below.
1. The labels on states satisfy $prop_1(r) \subseteq prop_2(s)$
2. For every state r' of S_1, if (r, r') is in E_1 there exists a state s' of S_2 satisfying that (s, s') is in E_2 and $act_1(r, r') \subseteq act_2(s, s')$.

A relation Sim is a *labelled bisimulation* if Sim is a simulation and the inverse relation Sim^{-1} is a simulation.

Recall from Section 3.3 that $salg(M)$ is the algebra generated by a transition system M and that $srel(\mathcal{A})$ is the transition system generated by an algebra $\mathcal{A}$. Consider a simulation relation Sim between transition systems M_1 and M_2 as above and an additive subsumption f from the predecessor algebra $\mathcal{A}_1$ to $\mathcal{A}_2$. The simulation relation defines a function and the subsumption defines a relation, as shown below.

$$salg(Sim) : \mathscr{P}(S_1) \to \mathscr{P}(S_2) \qquad salg(Sim) \mathrel{\hat{=}} \{X \mapsto Sim(X)\}$$
$$srel(f) \subseteq Atom(A_1) \times Atom(A_2) \qquad srel(f) \mathrel{\hat{=}} \{(a, b) \mid b \sqsubseteq f(a)\}$$

We show that the constructions given above generate a subsumption and a simulation relation, respectively.

Lemma 5.32. *The function $salg(Sim)$ is an additive subsumption.*

Proof. Let $\mathcal{M}_1$ and $\mathcal{M}_2$ be the algebras generated by M_1 and M_2, respectively. It follows from the Jónsson–Tarski theorem that $salg(Sim)$ is additive.

(*Constants*) If $X \subseteq p^{\mathcal{M}_1}$ every state r in X is labelled p. The definition of simulation entails that (r, s) is in *Sim* only if s is labelled p, so $Sim(X) \subseteq p^{\mathcal{M}_2}$, and the condition on constants holds.

(*Modalities*) Consider a transition (r_1, r_2) in E_1 such that (r_1, s_1) is in *Sim*. We can transcribe the simulation condition as

$$\{r_1\} \subseteq pre_a^{M_1}(\{r_2\}) \text{ implies } \{s_1\} \subseteq pre_a^{M_2}(\{s_2\})$$

The predecessor function is monotone and (r_2, s_2) is in *Sim*, and the condition above is satisfied by every (r_1, s_1) in *Sim* so

$$\{r_1\} \subseteq pre_a^{M_1}(\{r_2\}) \text{ implies } Sim(\{r_1\}) \subseteq pre_a^{M_2}(Sim(\{r_2\}))$$

By monotonicity of *pre* and the image of *Sim* and because the simulation condition holds for every (r_1, s_1) in *Sim*, we have

$$X \subseteq pre_a^{M_1}(Y) \text{ implies } Sim(X) \subseteq pre_a^{M_2}(Sim(Y))$$

satisfying the second condition for $salg(Sim)$ to be a subsumption. ⊣

Lemma 5.33. *The relation $srel(f)$ is a labelled simulation.*

Proof. Consider the function $f : A_1 \to A_2$, where A_1 and A_2 are perfect Boolean lattices in the predecessor algebras $\mathcal{A}_1$ and $\mathcal{A}_2$, respectively. Consider (r_1, s_1) in $srel(f)$ and (r_1, r_2) in the transition system M_1 generated by $\mathcal{A}_1$.

(*Propositions*) If p is in $prop_1(r_1)$, we have $f(r_1) \sqsubseteq p^{\mathcal{A}_2}$ so p is in $prop(s_1)$.

(*Transitions*) We have $r_1 \sqsubseteq pre^{\mathcal{A}_1}(r_2)$ by assumption and $f(r_1) \sqsubseteq pre^{\mathcal{A}_2}(f(r_2))$ by the definition of subsumption. By construction of M_1 and $srel(f)$, the element s_1 is an atom, and satisfies $s_1 \sqsubseteq f(r_1)$. Since *pre* and f are both strict additive, and $f(r_1)$ is not bottom, there must exist s_2 such that $s_2 \sqsubseteq f(r_2)$ and $s_1 \sqsubseteq pre^{\mathcal{A}_2}(s_2)$. The pair (r_2, s_2) is, by definition, in $srel(f)$. The argument above remains the same if a transition label is included, so the conditions for a simulation are satisfied. ⊣

Lemma 5.34. *If (f, b) is a bisubsumption between two predecessor algebras, the relation $srel(f)$ is a bisimulation.*

Proof. Theorem 2.27, due to Jónsson and Tarski, shows that if f and b are conjugate, the relation $srel(f)$ is the inverse of $srel(b)$. The definition of bisubsumption stipulates that f and b are subsumptions, so by Lemma 5.33 the inverse of $srel(f)$ is a simulation, so $srel(f)$ is a bisimulation. ⊣

We conclude the treatment of simulation and bisimulation with the representation theorem below.

Theorem 5.35. *Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be predecessor algebras and M_1 and M_2 be transition systems.*

1. *Every additive subsumption f from $\mathcal{A}_1$ to $\mathcal{A}_2$ is isomorphic to the subsumption $salg(srel(f))$ from $salg(srel(\mathcal{A}_1))$ to $salg(srel(\mathcal{A}_2))$.*
2. *Every simulation Sim from M_1 to M_2 is isomorphic to $srel(salg(Sim))$, a simulation from $srel(salg(M_1))$ to $srel(salg(M_2))$.*

The proof is similar in structure to the proofs of the representation theorems for state algebras. The main conclusion from this proof is that simulation relations are representations of a specific type of subsumptions between a specific type of algebras. Subsumptions being defined between arbitrary algebras are a strictly more general concept.

Ordered Simulations

A direct application of the representation theory from Chapter 3 is that we can derive relational notions that correspond to additive subsumptions between non-Boolean state algebras. In this section, we introduce the notion of an ordered simulation, which we prove is a relational representation of additive subsumptions between monotone predecessor algebras. Ordered simulations extend simulation to ordered transition systems.

Definition 5.36. Consider the monotone *pre*-transition systems $M_1 = (S_1, \preccurlyeq_1, E_1, prop_1, act_1)$ and $M_2 = (S_2, \preccurlyeq_2, E_2, prop_2, act_2)$. A relation $Sim \subseteq S_1 \times S_2$ is an *ordered simulation* if every (r, s) in Sim satisfies the conditions below.

1. $prop_1(r) \subseteq prop_2(s)$
2. For all states r_1, r_2 of S_1, if $r \preccurlyeq_1 r_1$ and (r_1, r_2) is in E_1 there exist states s_1, s_2 of S_2 satisfying the order $s \preccurlyeq_2 s_1$, with the transition (s_1, s_2) in E_2 and $act_1(r_1, r_2) \subseteq act_2(s_1, s_2)$.

Recall that monotone *pre*-transition systems have representations as perfect distributive lattices with strict additive transformers. Consider two monotone *pre*-transition systems M_1 and M_2 with an ordered simulation Sim from M_1 to M_2. Consider also two distributive predecessor algebras $\mathcal{A}_1$ and $\mathcal{A}_2$ and an additive subsumption f from $\mathcal{A}_1$ to $\mathcal{A}_2$. The constructions below derive a relation from Sim and a function from f.

$$salg(Sim) : \mathscr{D}(S_1) \to \mathscr{D}(S_2) \qquad salg(Sim) \triangleq \{X \mapsto Sim(X)\}$$
$$srel(f) \subseteq Irr_{\sqcup}(A_1) \times Irr_{\sqcup}(A_2) \qquad srel(f) \triangleq \{(a, b) \mid b \sqsubseteq f(a)\}$$

The definition of $srel(f)$ is as before with the difference that the relation is defined over join irreducibles. The definition of $salg(Sim)$ is over downwards closed sets instead of sets.

Lemma 5.37. *The function $salg(Sim)$ is an additive subsumption.*

Proof. Let $\mathcal{M}_1$ and $\mathcal{M}_2$ be the algebras generated by M_1 and M_2, respectively. It follows from the representation of distributive successor algebras that $salg(Sim)$ is additive.

(*Constants*) If $X \subseteq p^{\mathcal{M}_1}$ every state r in X is labelled p. The definition of simulation entails that (r, s) is in Sim only if s is labelled p, so $Sim(X) \subseteq p^{\mathcal{M}_2}$, and the condition on constants holds.

(*Modalities*) Consider a transition (r_2, r_3) in E_1 and a state $r_1 \preccurlyeq_1 r_2$ such that (r_1, s_1) is in Sim for some s_1. We can transcribe the ordered simulation condition using downward closed sets as below:

$$\{r_1\} \subseteq pre_a^{M_1}(\{r_2\})\downarrow \text{ implies } \{s_1\} \subseteq pre_a^{M_2}(\{s_2\})\downarrow$$

Lemma 3.41 shows that the predecessor sets above are downwards closed. The predecessor function is monotone and (r_2, s_2) is in Sim, and the condition above is satisfied by every (r_1, s_1) in Sim so

$$\{r_1\} \subseteq pre_a^{M_1}(\{r_2\}) \text{ implies } Sim(\{r_1\}) \subseteq pre_a^{M_2}(Sim(\{r_2\})).$$

By a similar sequence of arguments as in the proof of Lemma 5.32 it follows that $salg(Sim)$ is an additive subsumption. ⊣

Lemma 5.38. *The relation $srel(f)$ is an ordered simulation.*

Proof. Consider the function $f : A_1 \to A_2$, where A_1 and A_2 are perfect distributive lattices in the predecessor algebras $\mathcal{A}_1$ and $\mathcal{A}_2$, respectively. Consider (r_1, s_1) in $srel(f)$ and (r_2, r_3) in the transition system generated by $\mathcal{A}_1$, where $r_1 \sqsubseteq_1 r_2$.
(*Propositions*) If p is in $prop_1(r_1)$, we have $f(r_1) \sqsubseteq p^{\mathcal{A}_2}$ so p is in $prop_2(s_1)$.
(*Transitions*) We have $r_1 \sqsubseteq pre^{\mathcal{A}_1}(r_3)$ by assumption, the condition $f(r_1) \sqsubseteq pre^{\mathcal{A}_2}(f(r_2))$ by the definition of subsumption. The representation of distributive predecessor algebras and the definition of $srel(f)$, implies that the join irreducible s_1 satisfies $s_1 \sqsubseteq f(r_1)$. Since pre and f are both strict additive, and $f(r_1)$ is not bottom, there must exist s_3 such that $s_3 \sqsubseteq f(r_3)$ and $s_1 \sqsubseteq pre^{\mathcal{A}_2}(s_3)$. The pair (r_2, s_2) is, by definition, in $srel(f)$. The argument above remains the same if a transition label is included, so the conditions for an ordered simulation are satisfied. ⊣

The characterisations previously given for subsumption and additive subsumption all apply to ordered simulation relations. Ordered simulations form a complete lattice, have fixed point characterisations and have finitary analogues. It also follows from our previous results that ordered simulations characterise property preservation in completely disjunctive logics. The subsumption-based characterisations are strictly more general because ordered simulations are representations of strict additive subsumption between distributive predecessor algebras.

Theorem 5.39. *Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be distributive predecessor algebras and M_1 and M_2 be monotone pre-transition systems.*

1. *Every additive subsumption f from $\mathcal{A}_1$ to $\mathcal{A}_2$ is isomorphic to the subsumption $salg(srel(f))$ from $salg(srel(\mathcal{A}_1))$ to $salg(srel(\mathcal{A}_2))$.*
2. *Every ordered simulation Sim from M_1 to M_2 is isomorphic to $srel(salg(Sim))$, an ordered simulation from $srel(salg(M_1))$ to $srel(salg(M_2))$.*

Two-Sorted Simulations

The final application we consider is to define a generalisation of simulation to two-sorted transition systems. The structure of the proofs we require closely resembles those for ordered simulations so as with our treatment of conjunctive transition systems, we introduce the main definitions and theorems but skip the proofs.

Definition 5.40. Let $M_1 = (S_1, Q_1, \preccurlyeq_1, E_1, prop_1, act_1)$ and $M_2 = (S_2, Q_2, \preccurlyeq_2, E_2, prop_2, act_2)$ be two-sorted *pre*-transition systems. A relation $Sim \subseteq S_1 \times S_2$ is a *two-sorted simulation* if every (r, s) in Sim satisfies the conditions below.

1. $prop_1(r) \subseteq prop_2(s)$
2. For all states r_1, r_2 of S_1 such that $(r_1, r_2) \in E_1$, if for all q in Q_1, the order $r_1 \preccurlyeq q$ implies the order $r_2 \preccurlyeq q$, there exist states s_1, s_2 in S_2 satisfying that (s_1, s_2) is in E_2, for all q in Q_2, the order $s_1 \preccurlyeq q$ implies $s_2 \preccurlyeq q$ and $act_1(r_1, r_2) \subseteq act_2(s_1, s_2)$.

A two-sorted simulation generates a function that maps a Galois stable set to its image with respect to the simulation. An additive subsumption

between conjunctive predecessor algebras generates a relation between join irreducibles of the lattices of the two algebras.

$$\begin{aligned} &\mathit{salg}(\mathit{Sim}) : \mathscr{G}(S_1) \to \mathscr{G}(S_2) && \mathit{salg}(\mathit{Sim}) \mathrel{\hat{=}} \{X \mapsto \mathit{Sim}(X)\} \\ &\mathit{srel}(f) \subseteq \mathit{Irr}_{\sqcup}(A_1) \times \mathit{Irr}_{\sqcup}(A_2) && \mathit{srel}(f) \mathrel{\hat{=}} \{(a,b) \mid b \sqsubseteq f(a)\} \end{aligned}$$

The proofs of the lemmas below have the same structure as the proofs of Lemmas 5.37 and 5.38, so we only state the results.

Lemma 5.41. *The function $\mathit{salg}(\mathit{Sim})$ is an additive subsumption.*

Lemma 5.42. *The relation $\mathit{srel}(f)$ is a two-sorted simulation.*

Theorem 5.43. *Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be conjunctive predecessor algebras and M_1 and M_2 be conjunctive pre-transition systems.*

1. *Every additive subsumption f from $\mathcal{A}_1$ to $\mathcal{A}_2$ is isomorphic to the subsumption $\mathit{salg}(\mathit{srel}(f))$ from $\mathit{salg}(\mathit{srel}(\mathcal{A}_1))$ to $\mathit{salg}(\mathit{srel}(\mathcal{A}_2))$.*
2. *Every two-sorted simulation Sim from M_1 to M_2 is isomorphic to $\mathit{srel}(\mathit{salg}(\mathit{Sim}))$, a two-sorted simulation from $\mathit{srel}(\mathit{salg}(M_1))$ to $\mathit{srel}(\mathit{salg}(M_2))$.*

We emphasise again that Theorem 5.43 does not provide a representation for all subsumptions, only additive subsumptions between conjunctive predecessor algebras. The flow above would have to be repeated to generate representations of the analogous relations for other modalities.

SECTION SUMMARY This section recalled the standard notion of simulation and introduced two generalisations of simulation to monotone *pre*-transition systems and to conjunctive *pre*-transition systems. We showed that ordered simulations are relational representations of additive subsumptions between distributive predecessor algebras and that two-sorted simulations are relational representations of additive subsumptions between conjunctive predecessor algebras.

5.5 ABSTRACTION

There are three different perspectives relating subsumptions and abstract interpretation. The first perspective is that the lattice of subsumptions can be viewed as an underapproximation of the lattice of monotone functions. The second perspective is that subsumptions provide a technique for proving that an abstraction is complete with respect to properties expressed in a logic closed under conjunction. The third perspective is that the image of an algebra under the greatest subsumption from the algebra to itself is the coarsest abstraction that preserves properties in a logic closed under conjunction. We now discuss each of these perspectives in more detail.

SUBSUMPTIONS AS AN ABSTRACT DOMAIN The set of subsumptions between two algebras is a complete lattice. The closure operator characterisation of subsumptions in Lemma 5.13 shows that this lattice is an underapproximation of the lattice of monotone functions.

SUBSUMPTIONS AS A PROOF TECHNIQUE A subsumption from a concrete algebra to an abstract algebra shows that every property satisfied by the concrete algebra is satisfied by the abstract algebra. A sound abstraction satisfies a subset (not necessarily strict) of correctness properties of the concrete system.

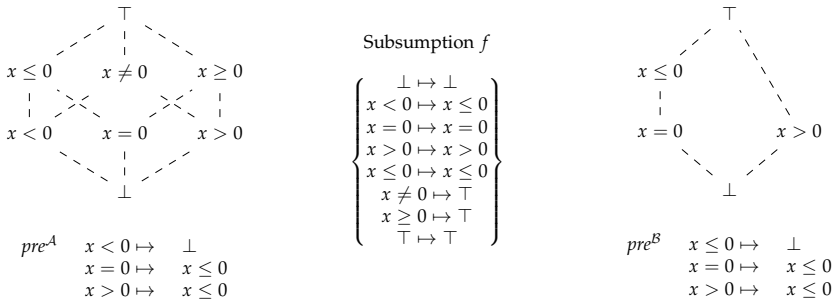


Figure 20: Two abstractions that are equivalent with respect to properties specified in a logic closed under conjunction and the next state modality. The logic does not contain disjunction or negation. The function f is a subsumption and can be used to show that the two algebras satisfy the same logical properties.

The example below illustrates a sound and complete abstraction of a transition system. The abstract lattice in the example is non-distributive, hence cannot be represented as a transition system. We apply subsumption to show that the abstraction is logically complete.

Example 5.44. Revisit the transition system from Example 4.19 in Chapter 4. The transition system on the right is an abstraction of the one on the left.

Consider the logic closed under the signature below. The logic can express that a value is equal to 0 or is strictly greater than 0. The logic includes one modality $\mathsf{EX}_{\rightarrow}$ and has conjunction as the only Boolean operation.

$$\mathit{Const} = \{x = 0, x > 0\} \quad \mathit{Mod} = \{\mathsf{EX}_{\rightarrow}\} \quad \mathit{Bool} = \{\wedge\}$$

Figure 15 contained three abstractions over which this logic could be interpreted. Two abstractions are recalled in Figure 20. The function f is a subsumption for the logic above, which means that every property satisfied by $\mathcal{A}$ is also satisfied by $\mathcal{B}$. The algebra $\mathcal{B}$ is a standard abstraction of $\mathcal{A}$ so the subsumption shows that the two have equivalent properties. ⌟

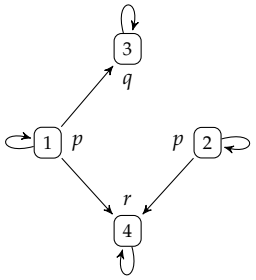
In the Galois connection framework of abstract interpretation, a pair of algebras $\mathcal{A}$ and $\mathcal{C}$ are related by an abstraction function α and a concretisation function γ. If γ is a subsumption, $\mathcal{A}$ is logically sound with respect to $\mathcal{C}$. If α is a subsumption, $\mathcal{C}$ is logically complete with respect to $\mathcal{A}$. Ranzato and Tapparo [2007] showed that logical completeness can be characterised in terms of concretisation functions. The application of subsumption above provides a different view, showing that completeness can also be characterised in terms of abstraction functions.

SUBSUMPTION AND OPTIMAL ABSTRACTION We now show that the fixed point characterisation of subsumption can be used to synthesise abstractions of a transition system. If the greatest subsumption from an algebra to itself is an upper closure operator, the image of the lattice under the

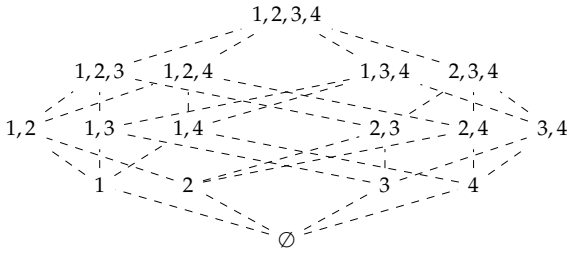
subsumption is the coarsest abstraction of the system that preserves all logical properties. The example below illustrates the synthesis of such an abstraction.

Example 5.45. The logic in this example has propositions (p, q, r) and is closed under $\mathsf{EX}_{\rightarrow}$, conjunction and disjunction. Negation is not permitted. It is known that all properties of a transition system M are also satisfied by a transition system N exactly if there is a simulation from M to N.

Consider the transition system N below. The states 3 and 4 have different labels. The state 2 simulates the state 1.



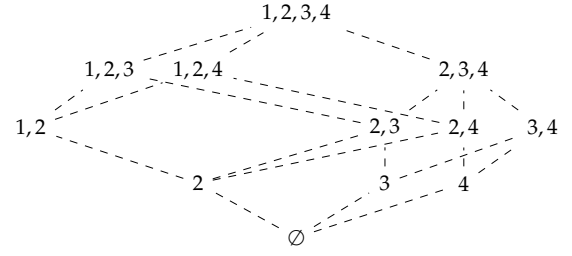
The transition system N generates the algebra $\mathcal{C} = (C, O_C)$, where C is the powerset lattice of states below and O_C contains the predecessor transformer. This lattice is shown below.



The optimal property preserving abstraction of C for infinitary modal logic can be derived using the greatest fixed point characterisation of subsumption. The element 1 is subsumed by 2 and does not appear as a singleton in the abstraction. We derive this abstraction using a calculation. Let $\mathrm{Mono}(C, C)$ be the set of monotone functions from C to itself. The greatest subsumption f additively extends the map below.

$$\{\emptyset \mapsto \emptyset, \{1\} \mapsto \{1, 2\}, \{2\} \mapsto \{2\}, \{3\} \mapsto \{3\}, \{4\} \mapsto \{4\}\}$$

The function f maps x to $x \cup \{2\}$ if x contains 1, and to x otherwise. Observe that f is monotone, idempotent and extensive: it is an upper closure. The image of c under f is shown below.



This is the optimal, property preserving abstraction of c. This abstraction is not a Boolean algebra. By the Jónsson–Tarski theorem, there are optimal abstractions that are not transition systems. ⌟

Thus, there are *property preserving abstractions* of logics closed under conjunction and disjunction that do not have transition system representations. We emphasise property preservation because there are abstract domains which are not transition systems, but these are rarely constructed from a temporal logic.

SECTION SUMMARY This section demonstrated two applications of subsumption in the context of abstract interpretation. The first application is to prove that an abstract algebra satisfies the same properties in a given logic as a concrete algebra. The second application is to synthesise the coarsest, property-preserving abstraction of an algebra.

5.6 BIBLIOGRAPHIC NOTES

This chapter was concerned with methods for showing that every property satisfied by an algebra $\mathcal{B}$ was also satisfied by another algebra $\mathcal{A}$. Proving such a statement for various types of properties is a classic problem in mathematics. The standard tool for such proofs is homomorphism. Isomorphism can further be used to show that two structures satisfy the same properties. Logics such as first-order logic or modal logics are not powerful enough to express isomorphism on arbitrary structures. This means that two structures may not be isomorphic but may satisfy the same properties in a first-order or a modal logic.

Techniques for comparing transition systems based on their logical properties can be classified by how they weaken the notion of isomorphism. These techniques have been discovered independently in set theory, first order logic, modal logic and process algebra, and discussing the numerous independent threads of development is beyond the scope of this dissertation. One way to weaken an isomorphism is to first weaken homomorphism to a relation rather than a function. This leads to a notion resembling simulation relations. Milner [1971] proposed an early, algebraic notion of simulation and later modified this notion to the popular form in his book [1989]. Since simulation applies to logics that are not closed under negation, it does not appear to have been independently discovered as often as bisimulation.

By the Jónsson and Tarski representation theorem, a simulation relation can be represented by a completely additive function between two perfect Boolean algebras. Additive subsumptions appearing in this chapter, when restricted to perfect Boolean algebras, represent simulations. However, additive subsumptions can be defined between algebras over perfect distributive lattices, so they strictly generalise simulations. To the best of the author's knowledge, ordered simulations introduced in this chapter have not previously appeared in the literature.

A second way to weaken isomorphism is to use two symmetric relations. This weakening is one of the main ideas behind bisimulation. The other idea is to define bisimulation coinductively rather than inductively. The idea of a symmetric relation appears multiple times in the modal logic literature [Blackburn et al. 2001] and was also rediscovered by Park and Milner. Sangiorgi [2009] discusses the discovery of bisimulation in different areas of logic and mathematics in great detail. It is often argued that a

back-and-forth argument of the form present in bisimulation appears in Cantor's work on set theory. However, Kueker [1975] argues that Cantor's proof only had one direction and not both.

There have been several attempts to characterise bisimulation algebraically and lift it to new families of structures. Abramsky [1991a] characterises bisimulation as the solution to a system of domain theoretic equations. The domains considered by Abramsky contain tree unwindings of processes and are different from the algebras we study. Cattani and Winskel [2005] studied bisimulation in topological terms and the relationship between their work to the work in this chapter is unclear. Malacaria [1995] gives a category theoretic characterisation of simulation and bisimulation but his work focuses on the category of transition systems. In contrast, although bisubsumptions are defined over perfect Boolean algebras, subsumptions in this chapter are defined over more general structures. Rutten [2000] proposed that bisimulation plays the role in the study of coalgebras that isomorphism plays in universal algebra. As with the work of Malacaria, much work on coalgebras is restricted to coalgebras over the category of sets, so despite the algebraic flavour of the work, it is surprising that bisubsumption and subsumption have not appeared in the literature.

Logical characterisations of notions like simulation and bisimulation appear in the model theory of first-order logic in the form of Fraïsse morphisms. These morphisms were formulated as games by Ehrenfeucht and are better known today as Ehrenfeucht-Fraïsse games. Standard texts on model theory describe both the games and morphisms [Poizat 2000]. Characterisations of bisimulation appear in modal logic through van Benthem's characterisation of zig-zag relations [Blackburn et al. 2001]. The Hennessy-Milner theorem [1985] relates bisimulation equivalence, transition systems, and Hennessy-Milner Logic, which is equivalent to standard modal logic. Abramsky [1991a] used domain theory to generalise the conditions under which the Hennessy-Milner theorem holds. The most comprehensive classification of such results we are aware of is the work of van Glabbeek [2001; Glabbeek [1993]. The work of van Glabbeek covers more preorders between transition systems than our work, but gives separate proofs for each preorder. If we consider only transition systems, our work is narrower in scope, but we believe our algebraic formulation leads to more succinct proofs.

This chapter differs from much work on logical characterisation of relations between structures by its use of lattices. Our work is closest to that of Ranzato and Tapparo [2007] who gave a different characterisation of simulation and bisimulation quotients in abstract interpretation terms. Their work showed that bisimulation and simulation quotients are instances of a generic, lattice-theoretic construction called the *forward complete shell* of an abstract domain. Our work differs in that we study the relationship between two different algebras, not necessarily two related by abstraction. Our work provides algebraic generalisation of simulation and bisimulation, while the work of Ranzato and Tapparo provides a generalisation of simulation and bisimulation quotients.

6

SOLVERS

> Martin Davis and Hilary Putnam noted that Gilmore's program failed on some rather simple examples because of its reliance on expansions into disjunctive normal form for satisfiability testing. This led them to the optimistic (and in restrospect rather naïve) conclusion that the lack of effective methods for testing large formulas of the propositional calculus for satisfiability was the main obstacle to be surmounted. Although their interest in algorithms for what came to be known as the *satisfiability problem* was only because they wanted to use such methods as part of a proof procedure for first-order logic, they secured support from the National Security Agency, to spend the summer of 1958 working on this problem.
>
> ...
>
> When George Logemann and Donald Loveland attempted to implement the program they found that the *rule for eliminating atomic formulas* (later called ground resolution) which replaced a formula
>
> $$(p \vee A) \wedge (\neg p \vee B) \wedge C \text{ by } (A \vee B) \wedge C$$
>
> used too much RAM. So it was proposed to instead use the *splitting rule* which generates the pair of formulas
>
> $$A \wedge C \qquad B \wedge C$$
>
> The idea was that a stack for formulas to be tested could be kept in external storage (in fact a tape drive) so that formulas in RAM never became too large.
>
> – Martin Davis,*The Early History of Automated Deduction*,2001

This chapter introduces *abstract satisfaction*, a framework that applies abstract interpretation to formalise the operations of satisfiability solvers. The contribution of this chapter is to characterise satisfiability of a formula using fixed points and transformers. We show that the Boolean Constraint Propagation (BCP) routine in modern SAT solvers is a greatest fixed point computation in an abstract domain and that clause learning can be understood as implementing an abstract transformer that prunes away regions of the search space that lead to a conflict.

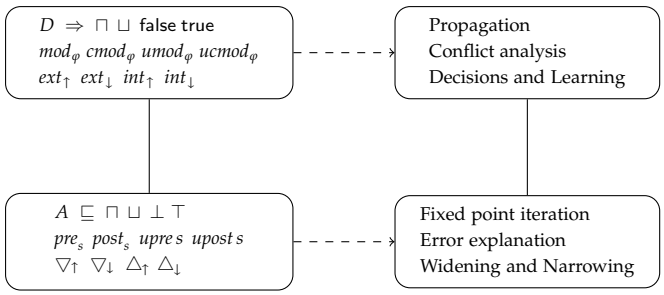
All the research reported in this chapter was conducted jointly with Leopold Haller and is included with his permission.

6.1 OVERVIEW

The Boolean satisfiability problem (SAT), which we also refer to as propositional satisfiability, is to determine if there exists a truth assignment that satisfies a given formula in propositional logic. Since the year 2000, the performance of SAT solvers has improved dramatically every year. A survey by Malik and Zhang [2009] surveys the developments leading to these performance improvements and concludes with the question below.

> *"Given its theoretical hardness, the practical success of* SAT *has come as a surprise to many in the computer science community. [...] Can we take these lessons to other problems and domains?"*

There are several factors that contribute to the success of SAT solvers. One is an algorithm called *Conflict Driven Clause Learning* (CDCL), which combines aspects of the classic DPLL algorithm [Davis et al. 1962] with a technique called conflict analysis [Silva and Sakallah 1996a]. In this chapter we analyse the core components of CDCL using abstract interpretation. The perspective we take in this chapter is summarised below.



A SAT solver consists of a family of data structures which are manipulated using deduction, abduction and heuristic updates to implement BCP, conflict analysis, decisions and learning. The main idea of this chapter is to order the sets of models represented by each state of a data structure in D above by implication. Under such an order, the data structures in a solver define abstract domains. The deduction and abduction rules in a solver become transformers that update abstract domain elements. BCP and conflict analysis in solvers can be understood as fixed point iteration, and heuristic manipulation such as decisions can be modelled by widening and narrowing operators.

This chapter takes a broad view of satisfiability and demonstrates that the core operations in satisfiability solvers have natural abstract interpretation formulations and new soundness proofs. The different notions of completeness in abstract interpretation strictly refine the notions of completeness used in satisfiability and decision procedures. This work is a first step towards a systematic, mathematical lifting of satisfiability algorithms to new problems.

SUMMARY OF CONCEPTS This chapter introduces the following concepts.

1. A domain and transformers for assignments in Section 6.2, which provides the concrete semantics of different approaches to satisfiability.
2. The abstract model transformer and the deduction transformers in Section 6.2 formalise BCP in terms of fixed points and transformers.
3. The pruning transformer formalises propositional clause learning in terms of abstract domains.

	Given	Determine
Emptiness check	P	$F \cap P = \emptyset$
Universality check	P	$F \cup P = U$
Superset	P	Q such that $F \cap P \subseteq Q$
Subset	P	Q such that $F \cap Q \subseteq P$
Interpolation	P, Q	I such that $F \cap P \subseteq I \subseteq F \cap Q$
Generalisation	$X_1 \subseteq X_2 \subseteq X_3 \cdots$	P with $X_1 \subseteq X_2 \subseteq X_3 \cdots \subseteq P$
Specialisation	$X_1 \supseteq X_2 \supseteq X_3 \cdots$	P with $X_1 \supseteq X_2 \supseteq X_3 \cdots \supseteq P$

Table 4: Manipulation of sets that can be viewed as the core operations in solvers. Assume a universe U and a set F, representing a fact assumed to be true. All other sets above are subsets of U.

SUMMARY OF RESULTS This chapter contains the following results.

1. A fixed point characterisation of satisfaction in Theorem 6.2.
2. The combination of abstract interpretation with the characterisation of satisfaction in Theorem 6.3.
3. Theorem 6.16 shows that BCP in SAT solvers is a fixed point computation in an abstract domain.
4. Lemma 6.17 shows that clause learning can be viewed as an abstract transformer for a restricted form of negation, and Theorem 6.18 shows how the combination of BCP with clause learning derives sound transformers from unsound assumptions.

CHAPTER ORGANISATION The view of satisfiability in terms of transformers is presented in Section 6.2. and BCP and clause learning are studied in Section 6.3.

6.2 ABSTRACT SATISFACTION

In this section we formulate satisfiability, validity and related logical problems using domains and transformers. To emphasise the similarity between the operation of satisfiability solvers and abstract interpreters and the applicability of this view to other problem domains, we begin with a set-theoretic formulation of the core operations in SAT solvers. We then redefine these operations in terms of logical formulae and fixed points and finally apply abstract interpretation to abstract these fixed points.

A Set-Theoretic View of Logical Operations

Assume a universe U and a set $F \subseteq U$ that we call a fact. In the logical setting, U will represent a set of structures and F will represent a set of axioms defining a theory. We identify several questions that can be formulated with respect to a fixed U and F. These questions are summarised in Table 4 and described in detail below.

1. The *generalised emptiness problem* is to check if $P \cap F$ is empty.
2. The *generalised universality problem* is to check if $P \cup F$ is the universe.
3. The *generalised superset problem* is to compute a superset Q of $P \cap F$.
4. The *generalised subset* problem is to compute a subset Q of $P \cap F$.

5. If $P \cap F$ is contained in $Q \cap F$, the *interpolation problem* is to compute a set I that contains $P \cap F$ and is contained in $Q \cap F$.
6. Given a finite sequence of sets $X_1 \subseteq X_2 \subseteq X_3 \cdots$ ordered by subset inclusion, the *generalisation problem* is to compute a set P that includes every set in the sequence.
7. Given a finite sequence of sets $X_1 \supseteq X_2 \supseteq X_3 \cdots$ ordered by superset inclusion, the *specialisation problem* is to compute a set P that is included in every set in the sequence.

The questions above can be grouped by duality. Emptiness and universality are dual questions. The superset and subset problems are dual and the generalisation and specialisation problems are dual. The interpolation problem is self-dual.

To relate this view to logic, we can replace $\emptyset$ and U by false and true and F by a set of formulae Φ representing a set of axioms. Instead of a set P, we have a formula ψ. The emptiness problem represents satisfiability of ψ with respect to a set of axioms Φ and universality represents validity of ψ with respect to Φ. The subset order becomes implication, so the superset problem becomes the deduction problem of finding a consequence of ψ. The subset problem becomes the abduction problem of finding a formula θ such that $\Phi \wedge \theta$ implies ψ. The interpolation problem is similar to interpolation in logic, with the difference that the syntax of a logic is used to further restrict the notion of a logical interpolant. Finally, generalisation and specialisation formalise the heuristic notions of inductive generalisation and inductive specialisation. The word inductive is used in the philosophical sense and not in the sense of mathematical induction. We formalise the operations above using transformers on a lattice.

A Lattice-Theoretic View of Logical Operations

Let *Prop* be a set of propositional variables. We define classes of formulae below, representing literals, clauses, cubes, conjunctive normal form (CNF), disjunctive normal form (DNF) and formulae, respectively. Assume that x ranges over *Prop*.

$$
\begin{array}{ll}
l ::= x \mid \neg l & \textit{Lit} \\
\theta ::= l_1 \vee \cdots \vee l_k & \textit{Clause} \\
\pi ::= l_1 \wedge \cdots \wedge l_k & \textit{Cube} \\
\varphi ::= \theta_1 \wedge \cdots \wedge \theta_n & \textit{CNF} \\
\psi ::= \pi_1 \vee \cdots \vee \pi_n & \textit{DNF} \\
\eta ::= x \mid \neg \eta \mid \eta \vee \eta \mid \eta \wedge \eta & \textit{Form}
\end{array}
$$

Recall the set $\mathbb{B} = \{\mathsf{true}, \mathsf{false}\}$ of truth values. The set of *assignments* is $\mathit{Asg} \triangleq \mathit{Prop} \to \mathbb{B}$. Let φ be a formula in propositional logic and σ be an assignment. We assume the standard definition of a satisfaction relation $\models$ over $\mathit{Asg} \times \mathit{Form}$ and write $\sigma \models \varphi$ if (σ, φ) is in $\models$. An assignment σ is a *model* of φ if $\sigma \models \varphi$ and is a *countermodel* otherwise.

The *domain* of *assignments* is the powerset lattice $(\mathscr{P}(\mathit{Asg}), \subseteq)$. Every formula φ defines four *assignment transformers* in $\mathscr{P}(\mathit{Asg}) \to \mathscr{P}(\mathit{Asg})$. Let X be a set of assignments. The *existential model transformer* mod_φ maps X to the largest subset of X containing models of φ. The *existential countermodel transformer* cmod_φ maps X to the largest subset of X containing countermodels

of φ. The *universal model transformer* $umod_\varphi$ adds all models of φ to X. The *universal countermodel transformer* $ucmod_\varphi$ adds all countermodels of φ to X.

$$\begin{aligned} mod_\varphi &\triangleq X \mapsto \{\sigma \in Asg \mid \sigma \in X, \sigma \models \varphi\} \\ cmod_\varphi &\triangleq X \mapsto \{\sigma \in Asg \mid \sigma \in X, \sigma \not\models \varphi\} \\ umod_\varphi &\triangleq X \mapsto \{\sigma \in Asg \mid \sigma \in X \text{ implies } \sigma \not\models \varphi\} \\ ucmod_\varphi &\triangleq X \mapsto \{\sigma \in Asg \mid \sigma \in X \text{ implies } \sigma \models \varphi\} \end{aligned}$$

We use mod_φ to implement the generalised superset problem and $ucmod_\varphi$ to implement the generalised subset problem. We summarise properties of these transformers below. The characterisations that follow allow us to apply results about abstractions of closure operators.

Theorem 6.1. *Assignment transformers have the following properties.*

1. *The transformers mod_φ and $cmod_\varphi$ are lower closures.*
2. *The transformers $umod_\varphi$ and $ucmod_\varphi$ are upper closures.*
3. *The transformers mod_φ and $cmod_\varphi$ are self-conjugate.*
4. *The transformers $umod_\varphi$ and $ucmod_\varphi$ are self-dual-conjugate.*
5. *The pairs $(mod_\varphi, ucmod_\varphi)$ and $(cmod_\varphi, umod_\varphi)$ are De Morgan duals.*
6. *There are two Galois connections*

$$(\mathscr{P}(Asg), \subseteq) \underset{mod_\varphi}{\overset{ucmod_\varphi}{\leftrightarrows}} (\mathscr{P}(Asg), \subseteq), \text{ and}$$
$$(\mathscr{P}(Asg), \subseteq) \underset{cmod_\varphi}{\overset{umod_\varphi}{\leftrightarrows}} (\mathscr{P}(Asg), \subseteq)$$

Proof. Let φ be a formula and M be the set of models of M. We consider each case separately.

(*Lower Closure*) The application $mod_\varphi(X)$ is equivalent to $X \cap M$. It follows that mod_φ is monotone and reductive, and, since $(X \cap M) \cap M = X \cap M$, mod_φ is idempotent. The argument for $cmod_\varphi$ is similar but with M replaced by $Asg \setminus M$.

(*Upper Closure*) The application $umod_\varphi(X)$ is equivalent to $X \cup M$. The rest of the argument is similar to the case for lower closures.

(*Self-Conjugate*) For two sets of assignments X and Y, $mod_\varphi(X) \cap Y$ and $X \cap mod_\varphi(Y)$ are both equivalent to $X \cap Y \cap M$, and must both be $\emptyset$ simultaneously. The same applies for $cmod_\varphi$, so both transformers are self-conjugate.

(*Dual-Self-Conjugate*) For two sets of assignments X and Y, $umod_\varphi(X) \cup Y$ and $X \cup umod_\varphi(Y)$ are both equivalent to $X \cup Y \cup M$, and must both be Asg simultaneously. The same applies for $ucmod_\varphi$, so both transformers are dual-self-conjugate.

(*De Morgan Duality*) From the definition of $\neg \circ mod_\varphi \circ \neg$ we have the De Morgan dual of mod_φ maps X to $\neg(\neg(X) \cup M)$, equivalent to $X \cup \neg M$, equivalent to $ucmod_\varphi(X)$. The argument for $(cmod_\varphi, umod_\varphi)$ is similar.

(*Galois Connection*) For mod_φ and $ucmod_\varphi$ observe that $X \cap M$ is contained in Y exactly if X is contained in $Y \cup \neg M$. ⊣

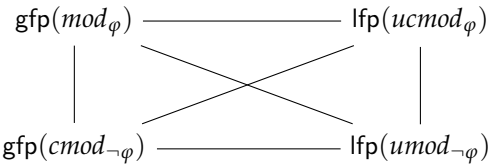
There are different ways to determine if a formula is satisfiable. We can check enumeratively if it has a model and think of such a procedure as applying the transformer mod_φ. We can check if every assignment is a countermodel, which amounts to checking if $\neg\varphi$ is valid, which we think of as applying the transformer $cmod_{\neg\varphi}$. Combined with the dualities and closure

properties above, we obtain four different fixed point characterisations of satisfiability.

Theorem 6.2. *The following statements are equivalent.*

1. *The formula φ is unsatisfiable.*
2. *The greatest fixed point $\mathsf{gfp}(mod_\varphi)$ is empty.*
3. *The least fixed point $\mathsf{lfp}(ucmod_\varphi)$ has all assignments.*
4. *The formula $\neg\varphi$ is valid.*
5. *The least fixed point $\mathsf{lfp}(umod_{\neg\varphi})$ has all assignments.*
6. *The greatest fixed point $\mathsf{gfp}(cmod_{\neg\varphi})$ has all assignments.*

The proofs above are straightforward because the operators are closures. The fixed points in Theorem 6.2 are summarised below.



The figure can be viewed as a design space for the concrete semantics of satisfiability. Every subset of edges in the figure represents one approach to designing a satisfiability solver. We can either directly reason about models, or directly reason about countermodels, or combine reasoning about models and countermodels.

We define pairwise combinations of transformers on $(\mathscr{P}(Asg) \times \mathscr{P}(Asg), \subseteq \times \subseteq)$ below. A combination procedure that checks satisfiability of φ using its models and validity of $\neg\varphi$ using countermodels is modelled by a *model-countermodel* transformer mc_φ defined below. This transformer is based on Cousot's forward-backward iteration [Cousot 2005]. The other transformers below are called the *model-universal model* transformer, *model-universal countermodel* transformer, *countermodel-universal model* transformer and so forth.

$$
\begin{aligned}
mc_\varphi &\mathrel{\hat{=}} (X, Y) \mapsto (mod_\varphi(X \cap Y), cmod_{\neg\varphi}(X \cap Y)) \\
mum_\varphi &\mathrel{\hat{=}} (X, Y) \mapsto (mod_\varphi(X \cap \neg Y), umod_{\neg\varphi}(\neg X \cup Y)) \\
muc_\varphi &\mathrel{\hat{=}} (X, Y) \mapsto (mod_\varphi(X \cap \neg Y), ucmod_\varphi(\neg X \cup Y)) \\
cum_\varphi &\mathrel{\hat{=}} (X, Y) \mapsto (cmod_{\neg\varphi}(X \cap \neg Y), umod_{\neg\varphi}(\neg X \cup Y)) \\
cuc_\varphi &\mathrel{\hat{=}} (X, Y) \mapsto (cmod_{\neg\varphi}(X \cap \neg Y), umod_{\neg\varphi}(\neg X \cup Y)) \\
umuc_\varphi &\mathrel{\hat{=}} (X, Y) \mapsto (umod_{\neg\varphi}(X \cup Y), ucmod_{\neg\varphi}(X \cup Y))
\end{aligned}
$$

To obtain intuition for the transformers above and the fixed point semantics they define, consider the greatest fixed point iteration with the model-countermodel transformer. The iteration begins from the top element of the product domain (Asg, Asg).

$$
\begin{aligned}
(X_0, Y_0) &\mathrel{\hat{=}} (Asg, Asg) \\
(X_1, Y_1) &\mathrel{\hat{=}} mc_\varphi(X_0, Y_0) = (mod_\varphi(X_0 \cap Y_0), cmod_{\neg\varphi}(X_0 \cap Y_0)) \\
&\qquad = (mod_\varphi(Asg), cmod_{\neg\varphi}(Asg)) \\
(X_2, Y_2) &\mathrel{\hat{=}} mc_\varphi(X_1, Y_1) = (mod_\varphi(X_1 \cap Y_1), cmod_{\neg\varphi}(X_1 \cap Y_1)) \\
&\qquad = (mod_\varphi(Asg), cmod_{\neg\varphi}(Asg))
\end{aligned}
$$

If the formula φ is unsatisfiable, we obtain the element $(\emptyset, Asg)$ as the fixed point. Over $\mathscr{P}(Asg) \times \mathscr{P}(Asg)$ this iteration does not provide different results from a single fixed point computation. In an abstract domain, we can obtain better results by transferring information between abstractions.

Abstract Satisfaction

We use the term *abstract satisfaction* for the application of abstract interpretation to approximate fixed point characterisations of satisfiability. If an overapproximation of the set of models of a formula is empty, we can conclude the formula is unsatisfiable. If an underapproximation of the set of countermodels of a formula contains all assignments, we can conclude the formula is unsatisfiable. Due to abstraction, an abstract analysis may produce inconclusive results. We show later that the algorithms in satisfiability solvers can be viewed as procedures to make a sound but incomplete analysis complete.

Let $(O, \sqsubseteq, \sqcup, \sqcap)$ and $(U, \preccurlyeq, \curlyvee, \curlywedge)$ be lattices. We say that O is an *overapproximation* of $\mathscr{P}(Asg)$ if there exists a Galois connection between $(\mathscr{P}(Asg), \subseteq)$ and $(O, \sqsubseteq)$. We say that U is an *underapproximation* of $\mathscr{P}(Asg)$ if there exists a Galois connection between $(\mathscr{P}(Asg), \supseteq)$ and $(U, \preccurlyeq)$, where we use the superset order instead of subset order. The soundness of a transformer is defined as usual with respect to the Galois connection involved.

The *overapproximate transformers* we consider are the *abstract model transformer* $amod_\varphi$ and the *abstract countermodel transformer* $acmod_\varphi$, both in $O \to O$. The *underapproximate transformers* we consider are the *abstract universal model transformer* $aumod_\varphi$ and the *abstract universal countermodel transformer* $aucmod_\varphi$, both in $U \to U$. Unless specified, we assume that all transformers are sound. The *abstract satisfaction theorem* below is a straightforward consequence of the soundness results of abstract interpretation and forms the basis for our treatment of satisfiability.

Theorem 6.3. *Consider sound abstract transformers* $amod_\varphi, acmod_{\neg\varphi} : O \to O$ *on an overapproximating abstraction* O*, and* $aumod_{\neg\varphi}, aucmod_\varphi : U \to U$ *on an underapproximating abstraction* U*. If one of the conditions below holds,* φ *is unsatisfiable.*

1. $\gamma(\mathsf{gfp}(amod_\varphi)) = \emptyset$
2. $\gamma(\mathsf{gfp}(acmod_{\neg\varphi})) = \emptyset$
3. $\gamma(\mathsf{lfp}(aumod_{\neg\varphi})) = Asg$
4. $\gamma(\mathsf{lfp}(aucmod_{\neg\varphi})) = Asg$

The concretisation functions are required in the theorem statement because the least element of an abstract domain may not concretise to $\emptyset$ or because multiple elements of the abstract domain concretise to $\emptyset$.

SECTION SUMMARY In this section, we identified a domain of partial assignments and four assignment transformers on this domain. We presented fixed point characterisations of satisfiability of a formula, and showed that abstract interpretation can be used to approximate these fixed points. We discuss examples of such fixed point approximations in the next section.

6.3 SATISFIABILITY ROUTINES

In this section, we formalise a few satisfiability procedures using lattices and transformers. We consider truth tables and resolution as simple examples, and consider BCP as an example arising in practice.

Truth Tables

A truth table is an enumeration that represents whether each truth assignment satisfies a formula. In abstract satisfaction, truth tables are a representation of the domain of assignments and truth table construction is application of the best abstract transformer for a formula. Binary Decision Diagrams (BDDs) are semantically equivalent but have a more efficient representation.

Example 6.4. This example illustrates the order on truth tables. Consider the formula $\varphi = p \wedge \neg q$. The set of assignments $\{p, q\} \to \mathbb{B}$ is shown in grey below. The truth tables for the formulae p and $\neg q$ are shown below.

p	q
false	false
false	true
true	false
true	true

p
false
false
true
true

$\sqcap$

$\neg q$
true
false
true
false

$=$

$p \wedge \neg q$
false
false
true
false

If the implication order on $\mathbb{B}$ is lifted to truth tables, the truth table for $p \wedge \neg q$ is the pointwise meet of the truth tables for p and $\neg q$. ⌟

Definition 6.5. A *truth table* is a function in $\mathit{Table} \mathrel{\hat{=}} \mathit{Asg} \to \mathbb{B}$. The *domain of truth tables* $(\mathit{Table}, \sqsubseteq, \sqcup, \sqcap)$ is the set of truth tables with the order $T_1 \sqsubseteq T_2$ if $T_1(\sigma) \Rightarrow T_2(\sigma)$ for every assignment σ.

We identify truth tables as an abstract domain below. Define two functions $\alpha : \mathscr{P}(\mathit{Asg}) \to \mathit{Table}$ and $\gamma : \mathit{Table} \to \mathscr{P}(\mathit{Asg})$.

$$\alpha(X) \mathrel{\hat{=}} \{\sigma \mapsto \mathsf{true} \mid \sigma \in X\} \cup \{\sigma \mapsto \mathsf{false} \mid \sigma \notin X\}$$
$$\gamma(T) \mathrel{\hat{=}} \{\sigma \mid T(\sigma) = \mathsf{true}\}$$

Proposition 6.6. *There is a Galois connection* $\mathscr{P}(\mathit{Asg}) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \mathit{Table}$ *in which* α *and* γ *are bijections.*

Proof. The truth table $\alpha(X)$ is the characteristic function for the set X, so the two are in a bijective correspondence. If $\alpha(X) \sqsubseteq T$, every assignment that maps to true in $\alpha(X)$ maps to true in T, so the set X is contained in $\gamma(T)$. ⊣

The Galois connection above is a Galois isomorphism, showing that truth tables do not lose information. Consider the best abstract transformer for mod_φ, denoted amod_φ. The element $\mathit{amod}_\varphi(\top)$ represents the truth table for φ. Truth table construction can be viewed as transformer application.

Resolution

The *resolution principle* states that an assignment satisfying the clauses $C \vee p$ and $\neg p \vee D$ also satisfies $C \vee D$ [Robinson 1965]. The variable p is the *pivot* and $C \vee D$ is the *resolvent*. Resolution is sound but is not complete for deriving arbitrary implications. For example, the formula $p \wedge q$ implies $p \vee \neg q$, but this implication cannot be derived by resolution. Resolution is refutation complete: a formula is unsatisfiable exactly if the empty clause can be derived by resolution. We model resolution as a transformer on a lattice of CNF formulae.

Definition 6.7. The CNF *domain* $\mathit{CNF} \mathrel{\hat{=}} \mathscr{P}(\mathit{Clause})$ contains sets of clauses with the superset order $(\mathit{CNF}, \supseteq, \cap, \cup)$.

The superset order above underapproximates implication because $\varphi \supseteq \psi$ entails $\varphi \Rightarrow \psi$ but the converse is not true. We define two functions $\alpha : \mathscr{P}(Asg) \to CNF$ and $\gamma : CNF \to \mathscr{P}(Asg)$ below.

$$\alpha(X) \triangleq \{\theta \in Clause \mid X \subseteq mod_\theta(Asg)\} \qquad \gamma(\varphi) \triangleq mod_\varphi(Asg)$$

Proposition 6.8. *There is a Galois connection* $(\mathscr{P}(Asg), \subseteq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (CNF, \supseteq)$.

Proof. If $\alpha(X) \supseteq \varphi$, for a CNF formula φ, every clause in φ is also in $\alpha(X)$, so every model of φ is a model of $\alpha(X)$ and $X \subseteq \gamma(\varphi)$ holds as required. ⊣

We formalise resolution with a transformer. The resolvents derived from φ with pivot x are denoted $res(x, \varphi)$. The *resolution transformer* $Res_\varphi : CNF \to CNF$ adds all possible resolvents to a set of clauses. Unlike the standard application of resolution, the transformer below allows for an assumption φ in the background.

$$res(x, \varphi) \triangleq \{C \vee D \mid x \vee C \text{ and } \neg x \vee D \text{ are in } \varphi\}$$
$$Res_\varphi(\psi) \triangleq \varphi \cup \psi \cup \bigcup_{x \in Prop} res(x, \varphi)$$

We describe properties of resolution using abstract interpretation. Logical soundness stating that every clause derived by resolution is implied by φ becomes the condition $\alpha \circ mod_\varphi \supseteq Res_\varphi \circ \alpha$. Res_φ is not idempotent, so multiple applications of resolution yield more resolvents than a single application. The set of clauses derived by resolution is the fixed point $\mathsf{gfp}(Res_\varphi)$. Resolution is not complete for arbitrary implications so, in general, $\alpha(\mathsf{gfp}(mod_\varphi))$ is a strict superset of $\mathsf{gfp}(Res_\varphi)$. The refutation completeness of resolution becomes the condition that $\gamma(\mathsf{gfp}(Res_\varphi))$ is the empty set exactly if $\mathsf{gfp}(Res_\varphi)$ contains the empty clause.

Boolean Constraint Propagation

The workhorse of all solvers based on DPLL is BCP. BCP repeatedly applies a transformation called the *unit rule* to a data structure called a *partial assignment*. We show that partial assignments are an abstract domain, that the unit rule is the best abstract transformer for a clause and that BCP computes a greatest fixed point.

Example 6.9. We illustrate BCP with the formula below.

$$\varphi \triangleq p \wedge (\neg p \vee \neg q) \wedge (q \vee r \vee \neg s) \wedge (q \vee r \vee s)$$

Initially, nothing is known about the formula, encoded by the empty set. Then, BCP concludes that p must be true in every satisfying assignment. Since p must be true, BCP concludes that q must be false to satisfy the clause $\neg p \vee \neg q$.

$$\pi_0 \triangleq \top \qquad \pi_1 \triangleq (p{:}\mathsf{true}) \qquad \pi_2 \triangleq (p{:}\mathsf{true}, q{:}\mathsf{false})$$

All the remaining clauses have more than one literal unassigned, so BCP terminates. BCP is a sound but incomplete deduction procedure. BCP need not begin with π_0 as above. We can begin by assuming that p is true, that q is false and that r is false, written $\pi \triangleq (p{:}\mathsf{true}, q{:}\mathsf{false}, r{:}\mathsf{false})$. Given π, BCP concludes, from $(q \vee r \vee \neg s)$, that s must be false and from $(q \vee r \vee s)$

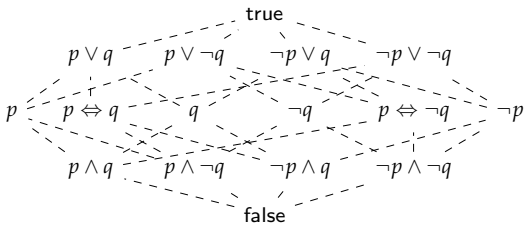
that s must be true. This situation, denoted $\bot$, is a *conflict*. No assignment extending π satisfies φ. ⌟

We show that partial assignments are an abstract domain. Consider the lattice $\mathscr{P}(\mathbb{B})$, where we write $\top$ for $\mathbb{B}$ and $\bot$ for $\emptyset$.

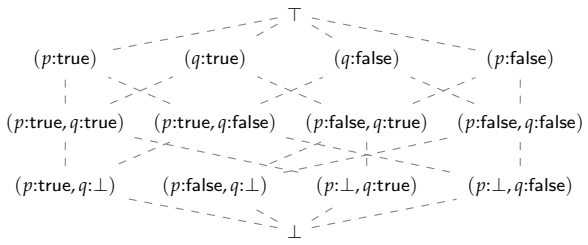
Definition 6.10. A *partial assignment* is a function in $PAsg \triangleq Prop \to \mathscr{P}(\mathbb{B})$. The *domain of partial assignments* consists of $(PAsg, \sqsubseteq, \sqcup, \sqcap)$, where $\pi_1 \sqsubseteq \pi_2$ if $\pi_1(x) \sqsubseteq \pi_2(x)$ for every variable x.

For brevity, we denote a partial assignment in which p is true and other variables map to $\top$ by $(p{:}\mathsf{true})$. The next example illustrates assignments and partial assignments over two variables.

Example 6.11. A set of assignments over two variables p and q can be described by a formula that all assignments in the set satisfy. The lattice $\mathscr{P}(Asg)$ with one formula representing each set is shown below. This lattice is sometimes called the Lindenbaum-Tarski algebra of a logic.



The domain of partial assignments over these variables is shown below.



This domain is non-distributive and non-relational, meaning it cannot represent the set of assignments corresponding to $p \iff q$. ⌟

The partial assignments domain as defined above was introduced for constant propagation by Kildall [1973] and is equivalent to the Cartesian abstraction [Cousot 2005] of $\mathscr{P}(Asg)$. The domains $PAsg$ and $\mathscr{P}(Asg)$ are related by the functions $\alpha_{PAsg} : \mathscr{P}(Asg) \to PAsg$ and $\gamma_{PAsg} : PAsg \to \mathscr{P}(Asg)$ below, which are known to form a Galois connection.

$$\alpha_{PAsg}(X) \triangleq \left\{ x \mapsto \bigsqcup \{\sigma(x) \mid \sigma \in S\} \mid x \in Prop \right\}$$
$$\gamma_{PAsg}(\pi) \triangleq \{\sigma \in Asg \mid \text{ for all } x \text{ in } Prop, \sigma(x) \sqsubseteq \pi(x)\}$$

The partial assignments $(p{:}\mathsf{true}, q{:}\bot)$ and $\bot$ concretise to the empty set so the Galois connection above is not a Galois insertion.

We now formalise the unit rule as a transformer on this lattice. The *unit rule* states that if all except one literals in a clause are false under a partial assignment, the remaining literal must be true. We write $\pi(\theta)$ for the evaluation of a clause given a partial assignment. The evaluation $\pi(\theta)$ is false if π and makes all literals in θ false.

Definition 6.12. The unit rule is a function $\mathsf{unit} : \mathit{Clause} \times \mathit{PAsg} \to \mathit{PAsg}$.

$$\mathsf{unit}(\theta, \pi) \triangleq \begin{cases} \bot & \text{if } \pi(\theta) \text{ is } \mathsf{false} \\ \pi \cup \{p \mapsto \mathsf{true}\} & \text{if } \theta \text{ is } \psi \vee p \text{ and } \pi(\psi) = \mathsf{false} \\ \pi \cup \{p \mapsto \mathsf{false}\} & \text{if } \theta \text{ is } \psi \vee \neg p \text{ and } \pi(\psi) = \mathsf{false} \\ \pi & \text{otherwise} \end{cases}$$

Example 6.13. We illustrate the unit rule with the formula $\varphi \triangleq \neg p \wedge (p \vee \neg q)$. If we begin with $\top$, applying the unit rule to $\neg p$ yields $(p{:}\mathsf{false})$, and applying the unit rule to $(p \vee \neg q)$ with the partial assignment $(p{:}\mathsf{false})$ yields $(p{:}\mathsf{false}, q{:}\mathsf{false})$.

We present another derivation of the partial assignment $(p{:}\mathsf{false}, q{:}\mathsf{false})$. Assume we have best abstract transformers for literals. The abstract transformer for φ is derived by replacing conjunction and disjunction by pointwise meet and join.

$$\mathit{amod}_\varphi \triangleq \mathit{amod}_{\neg p} \sqcap (\mathit{amod}_p \sqcup \mathit{amod}_{\neg q})$$

We compute a greatest fixed point in the partial assignments domain.

$$\begin{aligned} \pi_0 &\triangleq \top \\ \pi_1 &\triangleq \mathit{amod}_\varphi(\pi_0) = (p{:}\mathsf{false}, q{:}\top) \\ \pi_2 &\triangleq \mathit{amod}_\varphi(\pi_0) = (p{:}\mathsf{false}, q{:}\mathsf{false}) \\ \pi_3 &\triangleq \mathit{amod}_\varphi(\pi_0) = (p{:}\mathsf{false}, q{:}\mathsf{false}) \end{aligned}$$

We derived the same sequence of partial assignments using the abstract transformer above as with the unit rule. ⌟

Lemma 6.14. *For a fixed clause θ, the unit rule is equivalent to the best abstract transformer:* $\mathsf{unit}(\theta, \pi) = \alpha_{\mathit{PAsg}} \circ \mathit{mod}_\theta \circ \gamma_{\mathit{PAsg}}(\pi)$.

Proof. Consider a partial assignment π and the best abstract transformer $\mathit{amod}_\theta \triangleq \alpha_{\mathit{PAsg}} \circ \mathit{mod}_\theta \circ \gamma_{\mathit{PAsg}}$. We use the cases in the definition of unit.

($\pi(\theta)$ *is* false) If π makes every literal in θ false, $\mathsf{unit}(\theta, \pi) = \bot$. No assignment in $\gamma_{\mathit{PAsg}}(\pi)$ will θ, so $\mathit{mod}_\theta(\gamma_{\mathit{PAsg}}(\pi))$ is the empty set and by definition of α_{PAsg}, from $\mathit{amod}_\theta(\pi) = \bot$.

($\theta = \psi \vee p$ *and* $\pi(\psi) = \mathsf{false}$) Here, $\mathsf{unit}(\theta, \pi) = \pi \cup \{p \mapsto \mathsf{true}\}$. Since p is unassigned $\pi(p) = \top$, and the set $\gamma(\pi)$ contains assignments in which p is assigned to true and other assignments in which p is assigned to false false, but none of these assignments is a model of φ. The set $\mathit{mod}_\theta(\gamma_\pi(\pi))$ only includes assignments that satisfy p because no other literal is satisfied. All other variables are unaffected. Thus, $\alpha_{\mathit{PAsg}}(\mathit{mod}_\theta(\gamma_\pi(\pi)))$ equals $\pi \cup \{p \mapsto \mathsf{true}\}$.

(π *undefined for multiple variables in* θ) The unit rule leaves π unchanged. At least two literals in θ are undefined in π, so $\mathit{mod}_\theta(\gamma_{\mathit{PAsg}}(\pi))$ contains an assignment that makes one true and the other false and vice-versa. Consequently, the variables for both literals map to $\top$ in $\alpha_{\mathit{PAsg}}(\mathit{mod}_\theta(\gamma_{\mathit{PAsg}}(\pi)))$ and π is unchanged, as required. ⊣

BCP maps a formula φ and a partial assignment π to the result of applying the unit rule repeatedly with all clauses till no changes are observed. Formally, BCP is a function $\mathsf{bcp} : \mathit{CNF} \times \mathit{PAsg} \to \mathit{PAsg}$.

We model BCP using transformers and fixed points. Let φ be a formula, θ represent a clause and amod_θ be the best abstract transformer for mod_θ. We

model the effect of *concrete deduction* from a partial assignment Δ with the concrete transformer $mod_{\varphi,\Delta}$.

$$mod_\varphi : PAsg \times \mathscr{P}(Asg) \to \mathscr{P}(Asg) \qquad mod_{\varphi,\Delta}(x) \mathrel{\hat{=}} mod_\varphi(x \sqcap \gamma(\Delta))$$

The abstract *deduction transformer* below overapproximates $mod_{\varphi,\Delta}$.

$$\begin{aligned} ded_\varphi &: PAsg \times PAsg \to PAsg \\ ded_{\varphi,\Delta}(\pi) &\mathrel{\hat{=}} \bigsqcap \{amod_\theta(\pi \sqcap \Delta) \mid \theta \text{ is in } \varphi\} \end{aligned}$$

The soundness constraint $mod_{\varphi,\Delta} \circ \gamma_{PAsg} \subseteq \gamma_{PAsg} \circ ded_{\varphi,\Delta}$ implies that all conclusions derived by $ded_{\varphi,\Delta}$ are satisfied by all models of φ in Δ. Example 6.15 shows that the deduction transformer is not complete.

Example 6.15. The formula

$$\varphi \mathrel{\hat{=}} (\neg p \vee q) \wedge (p \vee \neg q) \wedge (\neg p \vee \neg q) \wedge (p \vee q)$$

is unsatisfiable. The best abstract transformer for mod_φ satisfies the equality

$$\alpha_{PAsg}(mod_{\varphi,\top}(\gamma_{PAsg}(\top))) = \bot$$

whereas the deduction transformer produces the result

$$\alpha_{PAsg}(mod_{\varphi,\top}(\gamma_{PAsg}(\top))) = \top$$

showing that the deduction transformer is incomplete. ⌟

Theorem 6.16. *The concrete result of Boolean Constraint Propagation is equivalent to the greatest fixed point of deduction:* $\gamma_{PAsg}(\mathsf{bcp}(\varphi, \Delta)) = \gamma_{PAsg}(\mathsf{gfp}(ded_{\varphi,\Delta}))$.

Proof. A run of BCP can be viewed as a sequence of distinct partial assignments. Each distinct partial assignment π_1 is generated by applying the unit rule with a clause to the previous assignment π_0. Applying the deduction transformer to π will generate a partial assignment $\pi_2 \sqsubseteq \pi_1$, so the result of BCP, denoted π_{bcp}, will satisfy $\mathsf{gfp}(ded_{\varphi,\Delta})\pi_{\mathsf{bcp}}$. In the other direction every application of $ded_{\varphi,\Delta}$ before a fixed point is reached changes the value of a set of variables, and each variable in that set is changed by the best abstract transformer for a clause. By Lemma 6.14, we can construct a sequence of BCP applications that produces the same result by applying the unit rule. ⊣

In abstract interpretation terms, BCP is a bottom-up abstract interpretation of Boolean expressions with locally decreasing iterations. See the work of Granger [1992] and the course notes of Cousot [2005] for details. Several classic methods of determining Boolean satisfiability such as the original DPLL algorithm and its successors can be understood as refining BCP to achieve completeness.

Clause Learning

Conflict analysis is a different approach to determine if a formula is satisfiable. Conflict analysis, if applied in full generality, is expensive, so techniques are required to prune the search space. In the CDCL algorithm in modern solvers BCP is used to discover partial assignments from which deduction leads to a conflict and conflict analysis is applied to this region of the search space. The result of conflict analysis is represented as a clause, hence the term *clause*

learning. In this section, we abstract away from several practical details and model the effect of clause learning as a transformer.

The concrete effect of learning is to ensure a region of the search space that contains no satisfying assignments is not explored repeatedly. We model this effect with a transformer that takes a partial assignment π and a set of assignments X and eliminates the assignments represented by *PAsg* from X. The *pruning transformer* is defined below.

$$\mathit{prune} : \mathit{PAsg} \times \mathscr{P}(\mathit{Asg}) \to \mathscr{P}(\mathit{Asg}) \qquad \mathit{prune}_\pi(X) \mathrel{\hat{=}} X \cap \neg\gamma_{\mathit{PAsg}}(\pi)$$

We show that clause learning generates an abstract pruning transformer.

The main challenge in implementing an abstract pruning transformer is to implement negation. Let π be a partial assignment. A partial assignment can be viewed as a conjunction of literals, also called a cube, so the negation of a partial assignment is a clause. Implementing negation by mapping a partial assignment to a clause is trivial, but generates a representation that is outside the abstract domain. The question now becomes how we can transfer information from a clause to a partial assignment. The solution used by SAT solvers is to use the unit rule to transfer information about how to prune the search space from a clause to a partial assignment. The function below is a *clausal negation* operation.

$$\mathit{clneg} : \mathit{PAsg} \to \mathit{Clause} \quad \mathit{clneg} \mathrel{\hat{=}} \pi \mapsto \begin{cases} \emptyset \text{ if } \pi = \top \\ \{p \mid \pi(p) = \mathsf{false}\} \text{ otherwise} \end{cases}$$

By combining clausal negation with the unit rule, we obtain an abstract pruning transformer

$$\mathit{aprune} : \mathit{PAsg} \times \mathit{PAsg} \to \mathit{PAsg} \qquad \mathit{aprune}_{\pi_1}(\pi_2) \mathrel{\hat{=}} \mathsf{unit}(\mathit{clneg}(\pi_1), \pi_2)$$

Lemma 6.17. *For a fixed partial assignment π, the transformer aprune_π is a sound overapproximation of prune_π.*

Proof. We consider cases from the definition of *clneg* and the unit rule. Let π' be a partial assignment to which aprune_π is applied.

(π *is* $\top$) The transformer prune_π maps all elements to the empty set, so aprune_π is trivially sound.

($\pi'(\mathit{clneg}(\pi)) = \mathsf{false}$) Every literal in $\mathit{clneg}(\pi)$ is made false by π', so by the definition of the unit rule, $\mathit{aprune}_\pi(\pi') = \bot$. We also have the inequality $\pi' \sqsubseteq \pi$, so, by monotonicity of concretisation and by negation, the equality $\gamma_{\mathit{PAsg}}(\pi') \cap \neg\gamma_{\mathit{PAsg}}(\pi) = \emptyset$ follows.

($\mathit{clneg}(\pi) = \psi \vee p$ *and* $\pi'(\psi) = \mathsf{false}$) We have $\mathit{aprune}_\pi(\pi') = \pi' \sqcap (p{:}\mathsf{true})$, so $\gamma_{\mathit{PAsg}}\mathit{aprune}_\pi(\pi')$ equals $\gamma(\pi') \cap \gamma((p{:}\mathsf{true}))$. By abuse of notation, treating a cube as a partial assignment, we can view π as $\neg\psi \sqcap (p{:}\mathsf{false})$, so the set $\mathit{prune}_\pi(\gamma_{\mathit{PAsg}}(\pi'))$ is $\gamma_{\mathit{PAsg}}(\pi') \cap \neg\gamma(\pi)$, and the latter is equivalent to $\gamma_{\mathit{PAsg}}(\pi') \cap \neg\gamma(\neg\psi) \cap \neg\gamma(p{:}\mathsf{false})$. Since $\gamma_{\mathit{PAsg}}(\pi')$ is contained in $\neg\gamma(\neg\psi)$, we can simplify further to $\gamma_{\mathit{PAsg}}(\pi') \cap \gamma(p{:}\mathsf{true})$, and show that aprune_π is sound in this case.

($\mathit{clneg}(\pi) = \psi \vee p$ *and* $\pi'(\psi) = \mathsf{true}$) Similar to the previous case.

(*Otherwise*) The transformer prune_π is reductive, and aprune_π is identity in this case, so we have soundness. ⊣

The theorem below shows that clause learning can be viewed as a method to move from a conclusion that φ is unsatisfiable under an assumption encoded by the partial assignment Δ to a sound transformer $ded_\varphi \sqcap aprune_\Delta$. By definition, $ded_\varphi \sqcap aprune_\Delta$ is at least as precise as ded_φ and in practice, is usually more precise.

Theorem 6.18. *If* $\mathsf{gfp}(ded_{\varphi,\Delta})$ *represents the empty set, the transformer* $ded_\varphi \sqcap aprune_\Delta$ *is a sound overapproximation of* mod_φ*.*

Proof. If the greatest fixed point $ded_{\varphi,\Delta}$ concretises to the empty set, no element of $\gamma_{PAsg}(\Delta)$ is a model of φ. It follows that $mod_\varphi(X) \sqcap \neg\gamma_{PAsg}(\Delta)$ equals $mod_\varphi(X)$ for all X. The expression above defines a transformer $prune_\Delta \circ mod_\varphi$ and, by Lemma 6.17, it follows that $ded_\varphi \sqcap aprune_\Delta$ is a sound transformer. ⊣

SECTION SUMMARY The goal of this section was to apply the abstract interpretation perspective to satisfiability procedures. We showed that truth table construction can be viewed as transformer application in an abstract domain isomorphic to the concrete domain. Resolution can be viewed as an abstract transformer on a domain of CNF formulae. We showed that the unit rule in SAT solvers is the best abstract transformer for a clause, and BCP is an abstract greatest fixed point computation. The final result was to show that clause learning in modern solvers can be be understood as an abstract pruning operation.

6.4 BIBLIOGRAPHIC NOTES

The fields of static program analysis and automated deduction have developed in parallel, largely independent of one another. Similar ideas have been developed in both fields in similar time spans, but the different vocabulary used, presentation and evaluation performed obscures these similarities. This chapter is a first step towards a clearer understanding of similarities between techniques in the two fields.

AUTOMATED DEDUCTION Consult essays by Bibel [2007], Bundy [1999] or Davis [2001] for a survey of the early period of automated deduction. The modern presentation of resolution is based on the work of Robinson [1965]. The idea of eliminating conflicting variables appeared earlier in the work of Davis and Putnam [1960] and in Blake's dissertation [1937]. The paper of Davis and Putnam [1960] also contains the unit rule, presented as a special case of an elimination rule. The quotation at the beginning of this chapter is taken from an article in which Davis [2001] explains that the application of resolution was memory intensive and led to the development of a splitting rule. The culture of algorithmic developments motivated by practical concerns is a hallmark of satisfiability research and manifests today in the form of competitions and openly available tools and benchmarks.

The algorithm in [Davis et al. 1962], which implemented a splitting rule, is called DPLL to acknowledge its origins in the work of Davis and Putnam [1960]. Over the course of five decades, the original DPLL algorithm was improved and modified to derive the Conflict Driven Clause Learning (CDCL) algorithm implemented in modern solvers. Conflict analysis was proposed as an alternate approach to Boolean satisfiability by Silva and Sakallah [1996b] and implemented using a graph-based data structure in GRASP [Silva and Sakallah 1996a]. Consult Malik and Zhang [2009] for a brief, high-level view

of these developments, and Zhang and Malik [2002] for a more detailed, albeit dated, survey.

The solver Chaff [Moskewicz et al. 2001] contained several new implementation ideas, which greatly improved the performance of clause learning SAT solvers. Since then, the performance of SAT solvers has improved continuously both in terms of running time and the size of formulae they handle.

In this chapter, we have proposed a framework to analyse and study satisfiability algorithms. This perspective is by no means comprehensive and omits several crucial factors behind the success of modern solvers. Nonetheless, an understanding of SAT solvers as abstract interpreters provides intuitive insights about SAT solvers.

One explanation for the performance of SAT solvers is that they operate on an abstract domain and refine the transformers they used in a demand-driven fashion. Modern solvers are known to perform badly on formulae that require intensive reasoning about equalities and disequalities, such as the pigeonhole Principle. Since partial assignments are a non-relational abstract domain, it is not surprising that they perform badly on such problems. The abstract interpretation perspective also suggests a way to perform better on such problems, which is to use a relational abstract domain.

STATIC ANALYSIS Static analysis is, of necessity, incomplete and computes approximations. A surprising insight of our work is that satisfiability procedures operate over imprecise abstractions but obtain sound and complete results. The main reason is that SAT solvers use techniques to refine the precision of an analysis.

Our work allows us to identify similar ideas in program analysis and satisfiability. For example, the lattice of constants [Kildall 1973], in the case of Boolean variables is equivalent to partial assignments used in SAT solvers. Applying the unit rule to every clause exactly once has the effect of evaluating a formula in the constants abstract domain. Such evaluation is called *bottom-up abstract interpretation* in the literature [Cousot 2005]. BCP is repeated unit rule application, which corresponds to the non-obvious idea of evaluating a Boolean expression multiple times in an abstract domain. This idea was called conditional constant propagation by Wegman and Zadeck [1991] and was generalised to arbitrary abstract domains by Granger [1992] who called it the method of *locally decreasing iterations*.

The splitting rule in DPLL can be viewed as a technique to improve the precision of a fixed point iteration by decomposing it into separate iterations. The idea of refining an analysis this way was called *qualified dataflow analysis* by Holley and Rosen [1980] and has been generalised and studied in the framework of *trace partitioning* by Rival and Mauborgne [2007].

A very popular refinement technique in the model checking literature is Counterexample-Guided Abstraction Refinement (CEGAR) [Clarke et al. 2003]. Refinement in SAT solvers is very different from CEGAR. Each iteration of the CEGAR loop requires constructing a new abstraction and new transformers. SAT solvers never change the domain. This immutability is crucial for efficiency as abstract domain implementations can be highly optimised. In fact, SAT algorithms can be understood as a portfolio of techniques for refinement without domain manipulation.

We are not aware of existing program analysis techniques that generalise CDCL in a strict mathematical sense but there are several tantalising similarities that deserve closer study. Transformer refinement used by Ball et al. [2001] to move from a Cartesian abstraction to a Boolean abstraction has a similar

effect to clause learning. Counterexample DAGs in [Gulavani et al. 2008] play a similar role to implication graphs, while the combination of testing with weakest preconditions in Yogi [Beckman et al. 2008] and with interpolants in lazy annotation [McMillan 2010] resembles the interplay between decisions and conflict analysis in modern solvers.

We conjecture that algorithms for solving satisfiability in a theory (SMT) have abstract interpretation characterisations and may exist independently in the static analysis literature. The analysis of a formula based on its propositional structure in DPLL(T) [Nieuwenhuis et al. 2006] is remarkably similar to the program analysis using control flow paths. The Nelson-Oppen combination procedure was recently shown to be an instance of the iterative reduced product [Cousot et al. 2011]. We believe that these are but a few directions that must be explored en route to an exciting unification of the theory and practice of decision procedures and static analysers.

7

INTERPOLANTS

> Important results have many things to say. At first sight, the Interpolation Theorem of Craig (1957) seems a rather technical result for connoisseurs inside logical meta-theory. But over the past decades, its broader importance has become clear from many angles. In this paper, I discuss my own current favourite views of interpolation: no attempt is made at being fair or representative. First, I discuss the *entanglement of inference and vocabulary* that is crucial to interpolation. Next, I move to the role of interpolants in facilitating *generalized inference across different models*. Then I raise the perhaps surprising issue of 'what is the right formulation of Craig's Theorem?', high-lighting the existence of non-trivial *options in formulating meta-theorems*.
>
> ...
>
> Finally, I discuss the 'end of history'. Craig's Theorem is about *the last significant property of first-order logic* that has come to light. Is there something deeper going on here, and if so, can we prove it?
>
> – Johan van Benthem, *The Many Faces of Interpolation*, 2008

This chapter applies abstract interpretation to formalise and analyse methods for construction of formulae called Craig interpolants. The contribution is a strict generalisation of existing constructions to a entire lattice of constructions. We show that the maximal abstraction of this lattice contains only three constructions, two of which exist in the literature. Thus, there is a precise sense in which existing techniques are optimal. A second contribution is a new characterisation of the strength of interpolants variables contained in interpolants generated by different constructions.

The labelled interpolation system and the theorem on interpolant strength were developed jointly with Mitra Purandare and Georg Weissenbacher, and have been included with their permission.

7.1 OVERVIEW

Interpolation theorems provide insights about what can be expressed in a logic or derived in a proof system. An interpolation theorem states that if A and B are logical formulae such that A implies B, there is a formula I defined only over the symbols occurring in both A and B such that A implies I and I implies B. This statement was proved by Craig [1957] for first-order logic and has since been shown to hold for several other logics and logical theories. Consult [Mancosu 2008] for a survey of the history and consequences of this theorem in mathematical logic. This chapter is concerned with constructing interpolants from propositional resolution proofs.

An *interpolation system* is an algorithm for computing interpolants from proofs. Interpolation systems lie at the heart of interpolation-based program verification. Consider three formulae $S(x)$, $T(x, x')$ and $\varphi(x')$, where $S(x)$ encodes a set of states S, $T(x, x')$ encodes a transition relation T and $\varphi(x')$ encodes a correctness property of the transition system. The *image* of S under the relation T is given by the formula $\exists x.S(x) \wedge T(x, x')$. The standard approach to determine if the states reachable from S satisfy the property φ is to iteratively compute images until a fixed point is reached. However, image computation and fixed point detection both involve quantifier elimination and are computationally expensive.

Consider the formula $S(x) \wedge T(x, x') \Rightarrow \varphi(x')$. If this formula is valid, the states reachable from S by a transition in T satisfy φ. Let A be the formula $S(x) \wedge T(x, x')$ and let B be the formula $\varphi(x')$ and I be an interpolant for $A \Rightarrow B$. The formula I represents a set of states that contains the image of S and satisfies the property φ. Thus, as shown by McMillan [2003], one can implement a property-preserving, approximate image operator with an interpolation system. Contemporary SAT solvers can generate resolution proofs, so an interpolation system for such proofs yields a verification algorithm for finite-state systems that uses only a SAT solver. The efficiency and precision of such a verification algorithm is contingent on the size and logical strength of the interpolants used. Hence, it is important to understand the properties of interpolants generated by different interpolation systems.

We are aware of three interpolation systems for propositional resolution proofs. The first, which we call the HKP-system, was discovered independently by Huang [1995], Krajíček [1997] and Pudlák [1997]. Another was proposed by McMillan [2003] and a third *parameterised* system was proposed by the author and his collaborators [D'Silva et al. 2010] as a generalisation of the other systems. One may, however, ask if the HKP-algorithm and McMillan's algorithm have properties that distinguish them from other instances of the parameterised algorithm. This chapter develops a framework to answer this and related questions.

SUMMARY OF CONCEPTS The following new concepts are introduced in this chapter.

1. *Interpolation systems*, introduced in Definition 7.13, enable a uniform presentation of interpolation algorithms in the literature.
2. *Interpolation parameters*, introduced in Definition 7.14, provide a way to generate interpolation systems from a colouring scheme.
3. The lattice of interpolation parameters in Section 7.4 is the basis for relating interpolation systems and abstract interpretation.

SUMMARY OF RESULTS The main results of the chapter are given below.

1. Theorem 7.16 generalises existing interpolation algorithms to the family of interpolation systems generated by locality preserving parameters.
2. We show in Section 7.3 that the interpolation algorithm of Huang, Krajíček and Pudlák, and that of McMillan are special cases of parameterised interpolation systems.
3. Theorem 7.30 shows that the set of parameters that partitions variables by colours is a join-semi-lattice with a maximal element.
4. Theorem 7.5 identifies an order on parameters which correlates with logical strength of interpolants and shows that sound parameters for interpolation form a complete lattice with respect to this order.

CHAPTER ORGANISATION Section 7.2 recalls background on propositional logic, interpolants and resolution proofs. Our new, parameterised family of interpolation systems is introduced in Section 7.3 and analysed using abstract interpretation in Section 7.4. In Section 7.5, we show that interpolation systems can be ordered by several criteria including labelling functions, logical strength and number of variables.

7.2 PROPOSITIONAL INTERPOLANTS

To keep this chapter accessible independent of the preceding material, some definitions are repeated. Let *Prop* be a *finite* set of propositions. The set of Boolean operators is $\mathit{Bool} \mathrel{\hat{=}} \{\mathsf{true}, \mathsf{false}, \neg, \wedge, \vee\}$. and the set of quantifiers is $\mathit{Quant} \mathrel{\hat{=}} \{\exists, \forall\}$. The grammar for various classes of propositional formulae and a description of each class follows.

$$
\begin{aligned}
\mathsf{lit} &::= \mathsf{prop} \mid \mathsf{neg}(\mathsf{prop}) \\
\mathsf{clause} &::= \mathsf{lit} \mid \mathsf{disj}(\mathsf{clause}, \mathsf{clause}) \\
\mathsf{cnf} &::= \mathsf{clause} \mid \mathsf{conj}(\mathsf{cnf}, \mathsf{cnf}) \\
\mathsf{form} &::= \mathsf{prop} \mid \mathsf{bool}(\widehat{\mathsf{x}}) \\
\mathsf{qbf} &::= \mathsf{form} \mid \mathsf{bool}(\widehat{\mathsf{quant}}) \mid \mathsf{quant}(\mathsf{prop}, \mathsf{form}) \\
\mathsf{inst} &\mathrel{\hat{=}} \{\mathsf{prop} \mapsto \mathit{Prop}, \mathsf{neg} \mapsto \{\neg\}, \mathsf{conj} \mapsto \{\wedge\}, \\
&\qquad \mathsf{disj} \mapsto \{\vee\}, \mathsf{bool} \mapsto \mathit{Bool}, \mathsf{quant} \mapsto \mathit{Quant}\}
\end{aligned}
$$

A *literal* is a proposition or the negation of a proposition. A *clause* is a disjunction of finitely many literals. A formula in *conjunctive normal form* (CNF) is a conjunction of finitely many clauses. A *propositional formula* is the composition of propositions with Boolean operators. A *quantified Boolean formula* (QBF) is a Boolean formula prefixed by a quantifier and the proposition being quantified or the Boolean combination of QBFs. A formula without quantifiers is *quantifier-free*. An *implication*, written $F \Rightarrow G$, is shorthand for $\neg F \vee G$ and universal quantification $\forall p.F$ is shorthand for $\neg\exists p.\neg F$. The formula $\exists P.F$ denotes the existential quantification of F with respect to a set of propositions $P = \{p_0, \ldots, p_n\}$. Let $F[p \mapsto G]$ denote that every occurrence of the proposition p in the formula F is replaced by the formula G. *Shannon expansion*, shown below, is a method of eliminating quantifiers from a formula.

$$
\begin{aligned}
\mathit{elim}(F) &\mathrel{\hat{=}} F \text{ if } F \text{ is quantifier-free} \\
\mathit{elim}(\exists p.F) &\mathrel{\hat{=}} \mathit{elim}(F)[p \mapsto \mathsf{true}] \vee \mathit{elim}(F)[p \mapsto \mathsf{false}] \\
\mathit{elim}(\forall p.F) &\mathrel{\hat{=}} \mathit{elim}(F)[p \mapsto \mathsf{true}] \wedge \mathit{elim}(F)[p \mapsto \mathsf{false}]
\end{aligned}
$$

The semantics of formulae is given by truth assignments, also called environments. An *environment* $\varepsilon : Prop \to \mathbb{B}$ associates a truth value with each proposition. The set of all environments is Env and the semantic domain is $\mathscr{P}(Env)$. The semantics function $[\![\cdot]\!]$ maps a formula to the set of environments that make the formula true. The semantics of operators is given below. Conjunction, disjunction and negation are interpreted as intersection, union and set complement.

$$\begin{aligned}
[\![p]\!] &\triangleq \{\varepsilon \in Env \mid \varepsilon(p) = \mathsf{true}\} \\
[\![F \wedge G]\!] &\triangleq [\![F]\!] \cap [\![G]\!] \\
[\![F \vee G]\!] &\triangleq [\![F]\!] \cup [\![G]\!] \\
[\![\neg F]\!] &\triangleq Env \setminus [\![F]\!] \\
[\![\exists p.F]\!] &\triangleq \{\varepsilon[p \mapsto \mathsf{true}], \varepsilon[p \mapsto \mathsf{false}] \mid \varepsilon \in [\![F]\!]\}
\end{aligned}$$

An element of $[\![F]\!]$ is a *model of* F. A formula is *satisfiable* if it has a model and is *unsatisfiable* otherwise. A formula is *valid* if every environment is a model of the formula. A formula G is a *consequence* of F if $[\![F]\!] \sqsubseteq [\![G]\!]$. Equivalently, F is *stronger than* G, or G is *weaker than* F. Note that G is a consequence of F if and only if $F \Rightarrow G$ is valid. The following fact about quantification is well known.

Lemma 7.1. *If the proposition p does not occur in F, the environment sets $[\![F]\!]$, $[\![\exists p.F]\!]$ and $[\![\forall p.F]\!]$ are equivalent.*

The semantics function maps formulae to sets of environments. A map from environments to formulae is defined next. Let $\vee\Phi$ be the disjunction of a finite set of formulae. The formula $\wedge\Phi$ is similarly defined. Let P be a set of propositions, ε be an environment and E be a set of environments. The formulae $form(P, \varepsilon)$ and $form(P, E)$ are defined below.

$$\begin{aligned}
form(P, \varepsilon) &\triangleq \wedge(\{p \in P \mid \varepsilon(p) = \mathsf{true}\} \cup \{\neg p \in P \mid \varepsilon(p) = \mathsf{false}\}) \\
form(P, E) &\triangleq \vee \{form(P, \varepsilon) \mid \varepsilon \in E\}
\end{aligned}$$

The *vocabulary* of a formula F, denoted $voc(F)$, is the set of propositions in the formula. In other words, the non-logical symbols in the formula. This chapter is concerned with formulae called interpolants. Craig [1957] proved that if an implication $F \Rightarrow G$ in first order logic is valid, there exists a formula I such that the implications $F \Rightarrow I$ and $I \Rightarrow G$ are both valid. Moreover, I satisfies a *vocabulary condition* that the non-logical symbols in I occur in both F and G. The formula I is called a *Craig interpolant*. Although the propositional reduct of this theorem has a simple proof, examining it is instructive.

Theorem 7.2. *For every valid implication $F \Rightarrow G$, there is a formula I such that $F \Rightarrow I$ and $I \Rightarrow G$ are valid and that $voc(I) \subseteq voc(F) \cap voc(G)$.*

Proof. Let $F \Rightarrow G$ be a valid implication. Let P be $voc(F) \cap voc(G)$ and I be the formula $form(P, [\![F]\!])$. The conditions for I to be an interpolant have to be verified. The vocabulary condition holds by construction.

($F \Rightarrow I$) An environment in $[\![F]\!]$ satisfies $form(P, \varepsilon)$, which is one disjunct in I. Thus, I is a consequent of F.

($I \Rightarrow G$) For every environment ε in $[\![I]\!]$ there exists an environment ε' in $[\![F]\!]$ that only differs from ε on propositions in $voc(F) \setminus voc(G)$. These propositions do not occur in G, and since $F \Rightarrow G$, it holds that ε is in $[\![G]\!]$ and that $I \Rightarrow G$. ⊣

The construction above requires examining all models of a formula and is computationally infeasible. Feasible interpolation methods are considered in the next section. The remainder of this section is about the set of interpolants for a fixed implication. For two formulae F and G, let $V_{F\setminus G}$ denote $voc(F) \setminus voc(G)$ and $V_{F\cap G}$ denote $voc(F) \cap voc(G)$.

Lemma 7.3. *A valid implication $F \Rightarrow G$ has a strongest interpolant $sint(F, G)$ and a weakest interpolant $wint(F, G)$.*

Proof. Define the two formulae below.

$$sint \mathrel{\hat{=}} elim(\exists V_{F\setminus G}.F) \qquad\qquad wint \mathrel{\hat{=}} elim(\forall V_{G\setminus F}.G)$$

The formula *sint* is equivalent to the interpolant in the proof of Theorem 7.2. Consider an interpolant I. It holds that $\exists P.F \Rightarrow \exists P.I$ and by Lemma 7.1, that $\exists P.F \Rightarrow I$. It follows that *sint* is the strongest interpolant. A similar argument applies to *wint*. ⊣

The strongest and weakest interpolants for an implication are written as *sint* and *wint* if the implication is clear. The set of interpolants is closed under several Boolean operations.

Example 7.4. Let I_1 and I_2 be interpolants for an implication. The formulae below are all interpolants.

$$I_3 \mathrel{\hat{=}} I_1 \wedge I_2 \qquad I_4 \mathrel{\hat{=}} (p \vee I_1) \wedge (\neg p \vee I_2) \qquad I_5 \mathrel{\hat{=}} I_1 \vee I_2$$

The variable p in I_4 can be quantified in two ways. In fact, $elim(\exists p.I_4)$ is equivalent to I_5 and $elim(\forall p.I_4)$ is equivalent to I_3. ⌟

A *Boolean function* $\varphi(\bar{x}, \bar{y})$ is a formula over Boolean operators and (potentially empty) sequences of variables. The function φ is *monotone in* y_i if for formulae F and G such that F implies G, φ with y_i replaced by F implies f with y_i replaced by G. The function φ is *idempotent in* $\bar{y}$ if for every formula F, replacing every element of $\bar{y}$ by F is equivalent to F.

Lemma 7.5. *Consider an implication $F \Rightarrow G$ and a Boolean function $\varphi(\bar{x}, \bar{y})$ that is monotone and idempotent in $\bar{y}$. The formulae below are interpolants.*

1. *$\varphi(\bar{p}, \bar{I})$, where $\bar{p}$ is a sequence of propositions in $V_{F\cap G}$ and $\bar{I}$ is a sequence of interpolants.*
2. *$elim(\bar{Q}.\varphi(\bar{p}, \bar{I}))$, where $\bar{p}$ is a sequence of propositions, $\bar{I}$ is a sequence of interpolants and $\bar{Q}$ is a quantification sequence over propositions not in $V_{F\cap G}$. Moreover, propositions in $V_{F\setminus G}$ are existentially quantified and those in $V_{G\setminus F}$ are universally quantified.*

Proof. The two cases are considered individually.

($\varphi(\bar{p}, \bar{I})$) Since φ is idempotent, $\varphi(\bar{p}, \bar{sint})$ and $\varphi(\bar{p}, \bar{wint})$, are the strongest and weakest interpolants, respectively. By monotonicity, $\varphi(\bar{p}, \bar{I})$ is implied by *sint* and implies *wint*. The propositions $\bar{p}$ are drawn from $V_{F\cap G}$, so the vocabulary condition holds. The formula is an interpolant.

($elim(\bar{Q}.\varphi(\bar{p}, \bar{I}))$) A similar argument shows that $\varphi(\bar{p}, \bar{I})$ is implied by *sint* and implies *wint*. Furthermore, these in $\bar{Q}$ are not in *sint* or *wint*, so by Lemma 7.1, $elim(\bar{Q}.\varphi(\bar{p}, \bar{I}))$ is logically between the extremal interpolants. The vocabulary condition holds, so the formula is an interpolant. ⊣

A corollary of Lemma 7.5 is that interpolants are closed under conjunction and disjunction. For p and q from the common vocabulary, the formulae below are also interpolants.

$$(p \Rightarrow I_1) \wedge (\neg p \Rightarrow I_2) \qquad (p \Rightarrow I_1) \wedge (q \Rightarrow I_2) \wedge ((\neg p \wedge \neg q) \Rightarrow I_3)$$

A large number of interpolants can be generated in this manner. The next statement shows that the models of such formulae have a limited structure. Let $Int(F, G)$ be the set of interpolants for an implication $F \Rightarrow G$ and $[\![\Phi]\!]$ be the set $\{[\![F]\!] \mid F \in \Phi\}$, where Φ is a set of formulae.

Theorem 7.6. *The set of interpolants modulo logical equivalence, represented by* $[\![Int(F, G)]\!]$*, is either a singleton or is a perfect Boolean algebra.*

Proof. Let M be $[\![Int(F, G)]\!]$. A consequence of Theorem 7.2 is that M cannot be empty (even though M may contain the empty set).

A proof that if M is not a singleton, it is a complete Boolean algebra is required. An operation on a set of environments X is given below.

$$X^c \triangleq [\![sint]\!] \cup ((Env \setminus X) \cap [\![wint]\!]))$$

Consider the structure $(M, \subseteq, \cup, \cap, \lambda x.x^c)$. By Lemma 7.3, $[\![sint]\!]$ and $[\![wint]\!]$ are the least and greatest elements of M, respectively. Furthermore, M is closed under intersection and union because interpolants are closed under conjunction and disjunction. The lattice is finite, hence complete. It remains to be shown that the lattice is complemented. If I is an interpolant, $I' = sint \vee (\neg I \wedge wint)$ is an interpolant as well. Observe that $[\![I]\!]^c$ is $[\![I']\!]$. In addition, $I' \vee I$ is equivalent to $wint$ and $I' \wedge I$ is equivalent to $sint$. The function $\lambda x.x^c$ is a complement operation, completing the proof. ⊣

Theorem 7.6 does not state that the set of interpolants is a complete Boolean algebra. Observe that implication is a preorder on formulae. This preorder gives rise to an equivalence relation $\Leftrightarrow$ on formulae and the quotient of $Int(F, G)$ with respect to $\Leftrightarrow$ is isomorphic to $[\![Int(F, G)]\!]$.

The constructions in the previous section were not feasible. In practice, satisfiability solvers are used to conclude if a propositional formula is satisfiable. Interpolants are constructed from produce resolution refutations generated by such solvers. Resolution refutations are studied in this section.

A proof rule is an n-ary relation over formulae. Resolution is typically applied to CNF formulae represented as sets. Let Lit be the set of literals over $Prop$. The set of clauses is $Clause \triangleq \mathscr{P}(Lit)$ and the set of CNF formulae is $CNF \triangleq \mathscr{P}(Clause)$. The empty clause, denoted $\square$, is equivalent to false and the empty formula is equivalent to true.

Definition 7.7. The *resolution principle* states that an assignment satisfying the clauses $C \vee p$ and $D \vee \neg p$ also satisfies $C \vee D$. It is presented as the proof rule below.

$$\frac{C \vee p \qquad D \vee \neg p}{C \vee D} \quad [\text{Res}]$$

The clauses $C \vee p$ and $D \vee \neg p$ are *antecedents*, the proposition p is the *pivot*, and $C \vee D$ is the *resolvent*.

A clause C is *derived from* a CNF formula F by resolution if C occurs in F or if C is the resolvent of two clauses that were derived from F. Resolution is *sound*: if C is derived from F, it holds that $[\![F]\!] \sqsubseteq [\![C]\!]$. A proof system is *complete* if whenever $[\![F]\!] \sqsubseteq [\![G]\!]$, G can be derived from F. Resolution is not complete. For instance, p implies $p \vee q$, but $p \vee q$ cannot be derived from p by resolution. However, resolution is complete for proving satisfiability. A CNF formula is unsatisfiable if and only if the empty clause can be derived by resolution. A derivation of a clause by resolution is represented by a graph called a *resolution proof*.

Definition 7.8. A resolution proof $P = (L, E, \mathit{clause}, \mathit{piv})$ is a directed, acyclic graph (L, E) with locations L, a set of edges E, a clause labelling function $\mathit{clause} : L \to \mathit{Clause}$ that labels vertices with clauses, and a partial function $\mathit{piv} : L \to \mathit{Prop}$ associating vertices with the pivot used for resolution.

1. Vertices have in-degree 0 or 2. Vertices with in-degree 0 are *initial* and those with in-degree 2 are *internal*. The vertex u in an edge (u, v) is the *parent* of v.
2. The function piv is defined for internal vertices.
3. For an internal vertex v with parents u_1 and u_2, the clause labelling v is the resolvent of $\mathit{clause}(u_1)$ and $\mathit{clause}(u_2)$ with pivot $\mathit{piv}(v)$.

The parent of an internal vertex containing $\mathit{piv}(v)$ is denoted v^+ and the parent containing $\neg\, \mathit{piv}(v)$ is denoted v^-.

A vertex u is an *ancestor* of v if there is a path from u to v or if u is v. A pair of vertices is *ancestor-free* if neither is an ancestor of the other. A *resolution refutation* is a proof with a vertex labelled with the empty clause.

We require a few additional notions to formalise the set of resolution proofs as a domain. The notion of coherence formalises that parts of proofs with the same vertices represent the same proof fragments.

Definition 7.9. A vertex v in a proof $\pi_1 = (L_1, E_1, \mathit{clause}_1, \mathit{piv}_1)$ and a vertex w in a proof $\pi_2 = (L_2, E_2, \mathit{clause}_2, \mathit{piv}_2)$ are coherent if

1. v and w are distinct, or
2. $v = w$, and $\mathit{clause}_1(v) = \mathit{clause}_2(w)$, and $\mathit{piv}_1(v) = \mathit{piv}_2(w)$, and the ancestors of v and w are the same vertices, and are coherent.

Two proofs are coherent if their vertices are pairwise coherent.

A formula F in CNF defines a proof $\pi(F)$ with a vertex for each clause in F and no edges.

$$
\begin{aligned}
\pi(F) \mathrel{\hat{=}} (L, \emptyset, \mathit{clause}, \mathit{piv}) & \\
L \quad & \text{contains a vertex } v_C \text{ for each } C \{F\} \\
\mathit{clause} \quad & \text{maps each vertex } v_C \text{ to the clause } C \\
\mathit{piv} \quad & \text{is undefined for all vertices}
\end{aligned}
$$

A *pointed proof* is a pair (π, v), where π is a proof and v is a vertex in π. The set of pointed, coherent resolution proofs is Proof. The set of all pointed proofs extending a proof π is defined below.

$$\mathit{point}(\pi) \mathrel{\hat{=}} \{(\pi, v) \mid v \text{ is a vertex in } \pi\}$$

The set of pointed proofs defined by a formula is $\mathit{point}(\pi(F))$. The resolution rule lifts naturally from clauses to pointed proofs. The clauses labelling the points in a proof are antecedents for the resolution rule. Consider proofs (π_1, v_1) and (π_2, v_2) with v_1 and v_2 being ancestor-free, $\mathit{clause}(v_1) = C \vee p$ and $\mathit{clause}(v_2) = \neg p \vee D$. The resolvent of (π_1, v_1) and (π_2, v_2) is a proof (π, v), where v does not occur in π_1 or π_2. The pivot at v is p, the clause at v is $C \vee D$ and the parents of v are $v^+ = v_1$ and $v^- = v_2$. Resolution of proofs can be written as a proof rule ProofRes $\subseteq \mathit{Proof} \times \mathit{Proof} \times \mathit{Proof}$. A tuple $((\pi_1, v_1), (\pi_2, v_2), (\pi_3, v_3))$ is in ProofRes if (π_3, v_3) is the resolvent of (π_1, v_1) and (π_2, v_2).

For a fixed set of propositions, let

$(\mathscr{P}(\mathit{Proof}), \subseteq, \cup, \cap)$ be the domain of resolution proofs.

The resolution rule is a transformer on this domain defined in steps below. We overload notation because only the last function below is required.

$$\begin{aligned} res &: Prop \times \mathscr{P}(Proof) \times \mathscr{P}(Proof) \to \mathscr{P}(Proof) \\ res(p, P, Q) &\triangleq \bigcup point(\pi, v), \text{ where } (\pi, v) \text{ is the resolvent of} \\ &\qquad (\pi_1, v_2) \in P \text{ and } (\pi_2, v_2) \in Q \text{ with pivot } p \\ res &: \mathscr{P}(Proof) \times \mathscr{P}(Proof) \to \mathscr{P}(Proof) \\ res(P, Q) &\triangleq \bigcup \{res(p, P, Q) \mid p \in Prop\} \end{aligned}$$

The resolution proof transformer is lifted to $\mathscr{P}(Proof)$ by defining $res(P)$ to be $res(P, P)$. The set of all resolution proofs over a formula F has a fixed point characterisation in terms of a function over $\mathscr{P}(Proof)$.

Proposition 7.10. *The set of all proofs that can be derived from a formula F by resolution is* $\mathsf{lfp}(res(X \cup point(\pi(F))))$.

There is a simple relationship between the domain of proofs and clauses. Recall that the set of CNF formulae, CNF, is the powerset of clauses. The domain of CNF formulae is $(CNF, \subseteq, \cup, \cap)$. Consider a function that projects out the labels of vertices from a proof. When lifted to sets of pointed proofs, we have an abstraction function from $\mathscr{P}(Proof)$ to CNF. The concretisation function takes a set of clauses and constructs all possible proofs with vertices labelled with exactly those clauses. Resolution, when defined over clauses, is an abstract transformer with respect to resolution defined over proofs.

SECTION SUMMARY This section recalled the notion of a Craig interpolants and included results about the properties of interpolants. We also recalled resolution and resolution proofs. Interpolation algorithms that operate on resolution proofs will be discussed next.

7.3 A COLOURED INTERPOLATION SYSTEM

This section presents a new method for constructing interpolants from resolution refutations. The method is strictly more general than existing methods and uses the idea of labelled proof systems [Basin et al. 2000]. In keeping with recent literature, we use a different definition of an interpolant from Craig's original definition.

An implication $F \Rightarrow G$ is valid exactly when $F \wedge \neg G$ is unsatisfiable. A CNF pair (F, G) is a pair of CNF formulae. A CNF pair is unsatisfiable if their conjunction is unsatisfiable. A definition of interpolants better suited to satisfiability problems follows.

Definition 7.11. An *interpolant* for an unsatisfiable CNF pair (F, G) is a formula I satisfying that $F \Rightarrow I$ is valid, that $I \wedge G$ is unsatisfiable and that $voc(I)$ is contained in $voc(F) \cap voc(G)$.

A system for constructing interpolants is now presented. Two symbols L and R are used to indicate the left and right component of a conjunction. Colouring functions indicate whether a proposition occurs in right, left, both or neither component of a conjunction.

Definition 7.12. A colouring function $col : Prop \to \mathscr{P}(\{\text{L}, \text{R}\})$. is a map from propositional variables to colours. The set of colouring functions is Col.

The set of colouring functions forms a complete lattice $(Col, \sqsubseteq, \sqcap, \sqcup)$ under the pointwise order. The definition of a pointwise order is recalled for completeness. For two colouring functions, $col_1 \sqsubseteq col_2$ if $col_1(p) \subseteq col_2(p)$ for every proposition p. The pointwise join $col_1 \sqcup col_2$ is a function that maps p to $col_1(p) \cup col_2(p)$. The resolvent $cRes(p, col_1, col_2)$ of two colouring functions with respect to a pivot p sets the colour of the pivot to the empty set and joins all other colours.

$$cRes(p, col_1, col_2)(q) \triangleq \begin{cases} \emptyset & \text{if } q = p, \text{ and is} \\ col_1(q) \cup col_2(q) & \text{otherwise.} \end{cases}$$

Interpolants are constructed from proofs using an interpolation system. An *interpolating clause* (i-clause) is a triple (F, col, I) consisting of a clause F, a colouring function col and a propositional formula I, called the *partial interpolant*. The partial interpolant need not be CNF. The components of an i-clause $E = (F, col, I)$ are denoted $clause(E)$, $fun(E)$ and $int(E)$. The set of i-clauses is $iClause \triangleq Clause \times Col \times Form$ and the set of i-CNF formulae is $iCNF \triangleq \mathscr{P}(iClause)$. An i-clause with the empty clause in the first component is written $E_\square$.

Definition 7.13. An interpolation system $Int = (T, \text{iRes})$ consists of a *translation function* $T : CNF \times CNF \to iCNF$, that maps a CNF pair to an i-CNF formula and an *interpolating resolution rule* iRes.

$$\frac{(C_1, col_1, I_1) \qquad (C_2, col_2, I_2)}{(C, col, I)} \quad [\text{iRes}]$$

The translation function extends clauses with colouring functions and partial interpolants without changing the clauses: $clause(T(F, G)) = F \cup G$.

The terms antecedent, resolvent and pivot apply to i-resolution as expected. An i-clause E is derived from an i-CNF formula F if it occurs in F or is the resolvent of applying the interpolating resolution rule to clauses derived from F. An i-clause E is derived from a CNF formula $F \wedge G$ if it is derived from $T(F, G)$.

The notions of soundness and completeness are refined for an interpolation system to separate soundness and completeness issues in deduction and interpolation. Let Int be an interpolation system. The system Int is *proof-sound* (p-sound) if for every i-clause E derived from an i-CNF formula F, the conjunction of clauses in F implies $clause(E)$. The system Int is *proof-complete* (p-complete) if for every clause C implied by a CNF formula $F \wedge G$, there exists an i-CNF clause E derived from $T(F, G)$ such that $clause(E) = C$. The system Int is *interpolation-sound* (i-sound) if for every unsatisfiable CNF pair (F, G) and derivation of an i-clause $E_\square$ from $T(F, G)$, the formula $int(E_\square)$ is an interpolant for (F, G). The system Int is *interpolation-complete* (i-complete) if for every unsatisfiable CNF pair (F, G) and interpolant I, if an i-clause $E_\square$ can be derived from $T(F, G)$, there exists a derivation with $int(E_\square)$ equivalent to I. Note that i-completeness only applies if there exists a proof that the formula is unsatisfiable.

A family of interpolation systems that includes and strictly generalises existing interpolation systems is introduced next. A *parameter* $\mathcal{P} : Clause \to Col$ associates a colouring function with each clause. For a clause C, literal $p \in C$ over a proposition q, and parameter $\mathcal{P}$, the colour of q is written $\mathcal{P}(C)(p)$ for convenience. Fix a colouring function col. The *upward restriction* of a clause C by a colour A, denoted $C\restriction_A$, is the subclause of literals whose

colouring includes A. The *downward restriction*, denoted $C\lfloor_{\text{A}}$, is similarly defined. The two restrictions are defined below.

$$C\lceil_{\text{A}} \triangleq \{p \in C \mid col(p) \supseteq \{\text{A}\}\} \qquad C\lfloor_{\text{A}} \triangleq \{p \in C \mid col(p) \subseteq \{\text{A}\}\}$$

Definition 7.14. A parameter $\mathcal{P}$ defines an interpolation system $Int_{\mathcal{P}} \triangleq (T_{\mathcal{P}}, \text{pRes})$ as defined below.

$$\begin{aligned} T_{\mathcal{P}}(F,G) \triangleq \{(C, \mathcal{P}(C), I) \mid C \in F \cup G\}\text{, where} \\ I \triangleq C\lfloor_{\text{R}} \text{ for } C \in F \text{ and colouring } \mathcal{P}(C), \text{ and} \\ I \triangleq \neg C\lfloor_{\text{L}} \text{ for } C \in G \text{ and colouring } \mathcal{P}(C) \end{aligned}$$

The parameterised inference rule follows.

$$\frac{(C_1 \vee p, col_1, I_1) \qquad (C_2 \vee \neg p, col_2, I_2)}{(C_1 \vee C_2, cRes(p, col_1, col_2), I)} \quad [\text{pRes}]$$

The partial interpolant I in the i-resolvent is defined below.

$$I \triangleq \begin{cases} I_1 \vee I_2 & \text{if } col_1(p) \cup col_2(p) = \text{L} \\ (p \vee I_1) \wedge (\neg p \vee I_2) & \text{if } col_1(p) \cup col_2(p) = \text{LR} \\ I_1 \wedge I_2 & \text{if } col_1(p) \cup col_2(p) = \text{R} \end{cases}$$

Every parameter gives rise to an interpolation system. Some of these interpolation systems are not sound. A few restrictions on the parameter ensure soundness. Literals occurring in a formula should not have empty colours, literals only in the left part of a formula should be coloured L and those only in the right part should be coloured R.

Definition 7.15. A parameter $\mathcal{P}$ is *locality preserving* for a CNF pair (F, G) if the following conditions hold.

1. For a clause C in (F, G) and literal $p \in C$, $\mathcal{P}(C)(p) \neq \emptyset$.
2. For a clause $C \in Clause$ and literal $p \in V_{F \setminus G}$, $\mathcal{P}(C)(p) \subseteq \{\text{L}\}$.
3. For a clause $C \in Clause$ and literal $p \in V_{G \setminus F}$, $\mathcal{P}(C)(p) \subseteq \{\text{R}\}$.

A locality preserving interpolation system is one generated by a locality preserving parameter.

Theorem 7.16. *Locality preserving interpolation systems are sound for interpolation.*

The proof is by induction on the structure of a derivation. Let (F, G) be an unsatisfiable CNF pair. The steps of the proof are divided into three lemmas. One lemma shows that partial interpolants satisfy the vocabulary condition. Two more lemmas show that partial interpolants satisfy the implications below. These implications are called the left and right condition, respectively.

$$F \wedge \neg C\lceil_{\text{L}} \Rightarrow I \qquad\qquad G \wedge \neg C\lceil_{\text{R}} \Rightarrow \neg I$$

Lemma 7.17. *In an interpolation system generated by a locality preserving parameter, every partial interpolant satisfies the vocabulary condition.*

Proof. Fix a CNF pair (F, G). The proof is by induction on the structure of a derivation.

(*Base case*) Clauses in F and G are considered separately. A clause C in F is translated to $(C, col, C\lfloor_{\text{R}})$. By the locality condition, the colour of propositions in $V_{F \setminus G}$ is L. Propositions in $V_{F \setminus G}$ do not appear in $C\lfloor_{\text{R}}$. A similar argument applies to clauses in B. The vocabulary condition holds for the base case.

(*Induction hypothesis*) Assume that the vocabulary condition is satisfied by partial interpolants of clauses derived by i-resolution.

(*Induction step*) Consider two i-clauses resolved with pivot p. Let the colouring functions be col_1 and col_2. If $col_1(p) \cup col_2(p) \neq \text{LR}$, no new propositions are introduced in the partial interpolant of the resolvent. By the induction hypothesis, the vocabulary condition holds. Suppose $col_1(p) \cup col_2(p)$ is LR. There are two cases. If these colours have not changed in the derivation, the propositions occur in $V_{F \cap G}$. If the colour changed, it must have been via i-resolution. However, every instance of a proposition that is not in $V_{F \cap G}$ has the same colour, which, by locality, is not LR. The pivot p must be from the common vocabulary. ⊣

Partial interpolants satisfy the vocabulary condition. The following lemma clarifies the connection of a partial interpolant to the left part of the formula.

Lemma 7.18. *If E is an i-clause derived from a* CNF *pair (F, G) in a locality preserving interpolation system, $F \wedge \neg(clause(E)\restriction_{\text{L}}) \Rightarrow int(E)$.*

Proof. The proof is by induction on the structure of a derivation. Consider an i-clause E derived from $T(F, G)$ with C being $clause(E)$.

(*Base case*) There is one case each for F and G.

($C \in F$) The partial interpolant is $C\restriction_{\text{R}}$. Observe that C is $C\restriction_{\text{L}} \vee C\restriction_{\text{R}}$. The condition $F \wedge \neg C\restriction_{\text{L}} \Rightarrow C\restriction_{\text{R}}$ is equivalent to $F \Rightarrow C$, which holds because C occurs in A.

($C \in G$) The partial interpolant is $C\restriction_{\text{R}}$. Locality ensures literals in C have non-empty colours. Since $C\restriction_{\text{L}} \subseteq C\restriction_{\text{R}}$, the implications $C\restriction_{\text{L}} \Rightarrow C\restriction_{\text{R}}$ and $\neg C\restriction_{\text{L}} \Rightarrow \neg C\restriction_{\text{R}}$ hold. In turn, $F \wedge \neg C\restriction_{\text{L}}$ implies $\neg C\restriction_{\text{R}}$.

(*Induction hypothesis*) Consider two i-clauses $E_1 = (C_1, col_1, I_1)$ and $E_2 = (C_2, col_2, I_2)$, a pivot p such that $p \in C_1$ and $\neg p \in C_2$, and an i-resolvent $E = (C, col, I)$ derived with pivot p. Assume that the implications below hold.

$$F \wedge \neg C_1\restriction_{\text{L}} \Rightarrow I_1 \qquad \text{and} \qquad F \wedge \neg C_2\restriction_{\text{L}} \Rightarrow I_2$$

(*Induction step*) There are three cases. Let D_1 be $C_1 \setminus p$ and D_2 be $C_2 \setminus \neg p$.

($col_1(p) \cup col_2(p) = \text{L}$) The pivot p is in $C_1\restriction_{\text{L}}$ and $\neg p$ is in $C_2\restriction_{\text{R}}$. The disjunction of the implications in the induction hypothesis yields:

$$F \wedge \neg D_1\restriction_{\text{L}} \wedge \neg D_2\restriction_{\text{L}} \Rightarrow I_1 \vee I_2$$

Furthermore, colours only change from L and R to LR, so the restriction $C\restriction_{\text{L}}$ is equal to $D_1\restriction_{\text{L}} \vee D_2\restriction_{\text{L}}$. Thus, $F \wedge \neg C\restriction_{\text{L}} \Rightarrow I_1 \vee I_2$ as required.

($col_1(p) \cup col_2(p) = \text{LR}$) There are different cases to consider because there are several possible values for $col_1(p)$ and $col_2(p)$.

($col_1(p), col_2(p) \in \{\text{L}, \text{LR}\}$) The induction hypotheses are:

$$F \wedge \neg p \wedge \neg D_1\restriction_{\text{L}} \Rightarrow I_1 \qquad \text{and} \qquad F \wedge p \wedge \neg D_2\restriction_{\text{L}} \Rightarrow I_2$$

Moving the p literals over the implication yields

$$F \wedge \neg D_1\restriction_{\text{L}} \wedge D_2\restriction_{\text{L}} \Rightarrow (p \vee I_1) \wedge (\neg p \vee I_2)$$

as required.

($col_1(p) = \text{R}$) Observe that $col_2(p)$ must be in $\{\text{L}, \text{LR}\}$ because the join of the two colours is LR. The pivot will not appear on the left of the implication for I_1. The induction hypotheses are:

$$F \wedge \neg D_1\restriction_{\text{L}} \Rightarrow I_1 \qquad \text{and} \qquad F \wedge p \wedge \neg D_2\restriction_{\text{L}} \Rightarrow I_2$$

Weakening the consequent of the first implication and conjunction with the second leads to

$$F \wedge \neg D_1\restriction_{\text{L}} \wedge D_2\restriction_{\text{L}} \Rightarrow (p \vee I_1) \wedge (\neg p \vee I_2)$$

which was to be shown. The case for $col_2(p) = \text{R}$ is identical.

($col_1(p) \cup col_2(p) = \text{R}$) If the pivot has colour R, the induction hypotheses reduce to

$$F \wedge \neg D_1\restriction_{\text{L}} \Rightarrow I_1 \qquad \text{and} \qquad F \wedge \neg D_2\restriction_{\text{L}} \Rightarrow I_2$$

Since $C = D_1 \vee D_2$, it follows that $F \wedge \neg C\restriction_{\text{L}} \Rightarrow I_1 \wedge I_2$. ⊣

The next lemma states the relationship between the partial interpolant and the right part of a formula.

Lemma 7.19. *If E is an i-clause derived from a* CNF *pair (F, G) in a locality preserving interpolation system, $G \wedge \neg(clause(E)\restriction_{\text{R}}) \Rightarrow \neg int(E)$.*

Proof. The proof is by induction on the structure of a derivation. Consider an i-clause E derived from $T(F, G)$ with C being $clause(E)$.

(*Base case*) The base case is similar to the base case of Lemma 7.18.

(*Induction hypothesis*) Consider two i-clauses $E_1 = (C_1, col_1, I_1)$ and $E_2 = (C_2, col_2, I_2)$, a pivot p such that $p \in C_1$ and $\neg p \in C_2$, and an i-resolvent $E = (C, col, I)$ derived with pivot p. Assume that the implications below hold.

$$G \wedge \neg C_1\restriction_{\text{R}} \Rightarrow \neg I_1 \qquad \text{and} \qquad G \wedge \neg C_2\restriction_{\text{R}} \Rightarrow \neg I_2$$

(*Induction step*) There are three cases to consider. Let D_1 be $C_1 \setminus p$ and D_2 be $C_2 \setminus \neg p$.

($col_1(p) \cup col_2(p) = \text{L}$) The pivot does not occur in the implications in the induction hypothesis. The condition $G \wedge \neg D_1\restriction_{\text{L}} \wedge \neg D_2\restriction_{\text{L}} \Rightarrow \neg(I_1 \vee I_2)$ follows by conjoining the two implications.

($col_1(p) \cup col_2(p) = \text{LR}$) There are different cases to consider because there are several possible values for $col_1(p)$ and $col_2(p)$.

($col_1(p), col_2(p) \in \{\text{R}, \text{LR}\}$) The induction hypotheses are:

$$G \wedge \neg p \wedge \neg D_1\restriction_{\text{R}} \Rightarrow \neg I_1 \qquad \text{and} G \wedge p \wedge \neg D_2\restriction_{\text{R}} \qquad \Rightarrow \neg I_2$$

Moving the p literals over the implication and taking conjunctions on the left and disjunction on the right yields

$$G \wedge \neg D_1\restriction_{\text{L}} \wedge D_2\restriction_{\text{L}} \Rightarrow \neg((\neg p \wedge I_1) \vee (p \wedge I_2))$$

This formula is equivalent, by a standard Boolean equality, to the form in the Lemma statement.

($col_1(p) = \text{L}$) The reasoning here, and for $col_2(p) = \text{L}$ are identical to the corresponding case in Lemma 7.18.

($col_1(p) \cup col_2(p) = \text{R}$) This case is similar to the case for the colours being L in Lemma 7.18. ⊣

The proof of Theorem 7.16 follows.

Proof. Let (F, G) be an unsatisfiable CNF pair and $E_\square$ be a derivation of the empty clause in a locality preserving interpolation system. Let I be $int(E)$. The vocabulary condition on $int(E)$ holds due to Lemma 7.17. Lemma 7.18 states that $F \wedge \neg clause(E)\restriction_{\text{L}} \Rightarrow I$ holds. Since $clause(E)$ is the empty clause, $F \Rightarrow I$. Lemma 7.19 states that $G \wedge \neg clause(E)\restriction_{\text{R}} \Rightarrow \neg I$ holds. This implication simplifies to $G \Rightarrow \neg I$. The conditions for I to be an interpolant hold. ⊣

The claim that the parameterised interpolation system is a strict generalisation of existing systems is now justified. Besides the parameterised system, two systems exist in the literature. A *symmetric interpolation system* was discovered independently by Huang [1995], Krajíček [1997] and Pudlák [1997]. The term symmetric will be justified shortly. McMillan [2003] introduced another system and gave a soundness proof in [2005]. The definitions of these systems, given next, ignore the colouring function.

Definition 7.20. The translation function and resolution rule of the symmetric interpolation system, $Int_{HKP} = (T_{HKP}, \text{iRes}_{HKP})$, follow. The colouring function col is arbitrary.

$$T_{\mathcal{P}}(F, G) \mathrel{\hat{=}} \{(C, col, \mathsf{false}) \mid C \in F\} \cup \{(C, col, \mathsf{true}) \mid C \in G\}$$

$$\frac{(C_1 \vee p, col_1, I_1) \qquad (C_2 \vee \neg p, col_2, I_2)}{(C_1 \vee C_2, col, I)} \quad [\text{iRes}_{HKP}]$$

The partial interpolant I in the i-resolvent is defined below.

$$I \mathrel{\hat{=}} \begin{array}{lll} I_1 \vee I_2 & \text{if } p \text{ occurs in} & V_{F \setminus G} \\ (p \vee I_1) \wedge (\neg p \vee I_2) & \text{if } p \text{ occurs in} & V_{F \cap G} \\ I_1 \wedge I_2 & \text{if } p \text{ occurs in} & V_{G \setminus F} \end{array}$$

Definition 7.21. McMillan's interpolation system $Int_M = (T_M, \text{iRes}_M)$, is defined below. The colouring function col is arbitrary. The translation $T_M(F, G)$ is the set of i-clauses $\{(C, col, I) \mid C \in F \cup G\}$, where I is as below.

$$I \mathrel{\hat{=}} \left\{ p \in C \mid voc(p) \subseteq V_{G \setminus F} \right\}, \text{ for } C \in F$$

$$I \mathrel{\hat{=}} \mathsf{true} \text{ for } C \in G$$

$$\frac{(C_1 \vee p, col_1, I_1) \qquad (C_2 \vee \neg p, col_2, I_2)}{(C_1 \vee C_2, col, I)} \quad [\text{iRes}_M]$$

The partial interpolant I in the i-resolvent is defined below.

$$I \mathrel{\hat{=}} \begin{array}{lll} I_1 \vee I_2 & \text{if } p \text{ occurs in} & V_{F \setminus G} \\ I_1 \wedge I_2 & \text{otherwise} & \end{array}$$

Definition 7.22. Fix a CNF pair (F, G). The colouring function col_{HKP} maps propositions in $V_{F \setminus G}$ to L, propositions in $V_{F \cap G}$ to LR and propositions in $V_{G \setminus F}$ to R. The colouring function col_M maps propositions in $V_{F \setminus G}$ to L and all other propositions to R. The parameter $\mathcal{P}_{HKP}$ maps all clauses to col_{HKP} and $\mathcal{P}_M$ maps all clauses to col_M

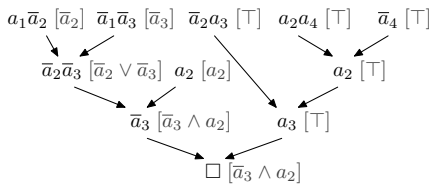


Figure 21: Interpolant construction using McMillan's method

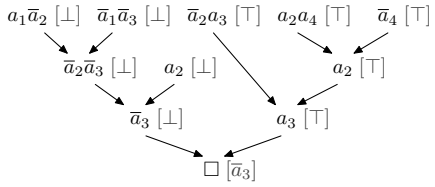


Figure 22: Interpolant construction using Huang, Krajíček and Pudlák

Example 7.23 shows that the two systems produce different interpolants and that different interpolants can be obtained by interchanging A and B. Example 7.24 identifies interpolants not obtained in either system.

Example 7.23. Consider the two formulae below.

$$A \mathrel{\hat{=}} (a_1 \vee \overline{a}_2) \wedge (\overline{a}_1 \vee \overline{a}_3) \wedge a_2 \qquad B \mathrel{\hat{=}} (\overline{a}_2 \vee a_3) \wedge (a_2 \vee a_4) \wedge \overline{a}_4$$

The e-clauses in McMillan's system are shown on the left of Figure 21 and those in the other system are on the right. The partial interpolants in both systems are shown in square brackets. The interpolants are different. The interpolant for (B, A) in McMillan's system is $a_2 \wedge a_3$. By negating it, we obtain $\neg a_2 \vee \neg a_3$, which is also an interpolant for (A, B) but is not the interpolant obtained from McMillan's system. In contrast, the interpolant for (B, A) in the HKP system is a_3, which, when negated, yields the same interpolant as before. ⌟

Example 7.24. Consider the two formulae below.

$$A \mathrel{\hat{=}} \overline{a}_1 \wedge (a_1 \vee \overline{a}_2) \qquad B \mathrel{\hat{=}} (\overline{a}_1 \vee a_2) \wedge a_1$$

A refutation for $A \wedge B$ is shown in Figure 23. The interpolant obtained in both systems is $\overline{a}_1 \wedge \overline{a}_2$. The interpolant for (B, A) obtained from Int_{HKP} is $a_1 \vee a_2$ and that obtained from Int_M is $a_1 \wedge a_2$. By negating these, we get the additional interpolant $\neg a_1 \vee \neg a_2$. The pair (A, B) has two more interpolants, namely $\neg a_1$ and $\neg a_2$. These interpolants can be obtained with Int_{HKP} and Int_M from different proofs. ⌟

SECTION SUMMARY This section introduced the new notions of interpolating clauses and a coloured interpolation system. We showed that a special class of locality preserving interpolation systems generates interpolants and interpolation algorithms that exist.

7.4 INTERPOLATION AND ABSTRACT INTERPRETATION

We introduce a lattice of parameters and show that locality preserving parameters are closed under joins on this lattice. Let $\mathcal{S}$ be the powerset lattice $(\wp(\{\text{L}, \text{R}\}), \subseteq, \cup, \cap)$. Define the *dual* of an element of $\mathcal{S}$ as follows: $\widehat{\text{L}} \mathrel{\hat{=}} \text{R}$,

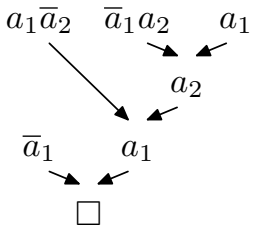


Figure 23: A proof from which the parameterised system produces an interpolant not produced by existing techniques

$\widehat{\text{R}} \triangleq \text{L}$, $\widehat{\text{LR}} \triangleq \text{LR}$ and $\widehat{\emptyset} \triangleq \emptyset$. That is, the dual of L is R and vice versa, but LR and $\emptyset$ are self-duals. The term *dual* is due to Huang [Huang 1995] who defined the dual of Int_{HKP}. The lattice of parameters is defined below.

Definition 7.25. The lattice of distinction functions, $(\mathbb{D}, \sqsubseteq^{\mathbb{D}}, \sqcup^{\mathbb{D}}, \sqcap^{\mathbb{D}})$, where $\mathbb{D} = \mathit{Prop} \to \mathcal{S}$, is derived from $\mathcal{S}$ by pointwise lifting. The lattice of parameters, $(\mathit{Clause} \to \mathbb{D}, \sqsubseteq, \sqcup, \sqcap)$, is derived from $\mathbb{D}$, also by pointwise lifting. The dual of a distinction function and a parameter are similarly defined by pointwise lifting.

In addition, define the function $\delta_{(A,B)}$ that maps a parameter $\mathcal{P}$ to one that agrees with $\mathcal{P}$ on $x \in V_A \cup V_B$ but maps all other variables to their duals.

$$\delta_{(A,B)}(\mathcal{P})(C)(x) \triangleq \begin{cases} \mathcal{P}(C)(x) \text{ if } x \in V_A \cup V_B \\ \widehat{\mathcal{P}}(C)(x) \text{ if } x \in V_{(A,B)} \end{cases}$$

Locality preserving parameters define correct interpolation systems, so operations on parameters that preserve locality are of particular interest. Such operations are illustrated in Example 7.26 and formally identified in Lemma 7.27.

Example 7.26. Consider the formula $A = \{\{a_1, \neg a_2\}, \{\neg a_1, \neg a_3\}, \{a_2\}\}$ and B, which is $\{\{\neg a_2, \neg a_3\}, \{a_2, a_4\}, \{\neg a_4\}\}$, both from Example 7.23. Define the $\mathcal{P}_4, \mathcal{P}_5$ and $\mathcal{P}_6$ as below. Let $C \in \mathit{Clause}$ be a clause.

- $\mathcal{P}_4(C)(x)$ is L for $x \in V_A$, and is R for $x \notin V_A$.
- $\mathcal{P}_5(C)(x)$ is L for $x \notin V_B$, and is R for $x \in V_B$.
- $\mathcal{P}_6(C)(x)$ is L for $x \in V_A$, is LR for $x \in V_{(A,B)}$, and is R for $x \in V_B$.

These parameters are locality preserving for (A, B) and their duals are locality preserving for (B, A). We also have that $\delta_{(A,B)}(\mathcal{P}_4) = \mathcal{P}_5$ and $\mathcal{P}_4 \sqcup \mathcal{P}_5 = \mathcal{P}_6$, so $\delta_{(A,B)}$ and $\sqcup$ preserve locality. In contrast, $\mathcal{P}_4 \sqcap \mathcal{P}_5$ is not locality preserving for (A, B). ⌟

Lemma 7.27. *Let (A, B) be a CNF pair.*

1. *If $\mathcal{P}_1$ and $\mathcal{P}_2$ are locality preserving for (A, B), then so is $\mathcal{P}_1 \sqcup \mathcal{P}_2$.*
2. *If $\mathcal{P}$ is locality preserving for (A, B), then $\widehat{\mathcal{P}}$ is locality preserving for (B, A). Further, if C is derived by resolution and E and F are the corresponding clauses in $Int_{\mathcal{P}}$ and $Int_{\widehat{\mathcal{P}}}$ respectively, then $int(E) = \neg int(F)$.*
3. *If $\mathcal{P}$ is locality preserving for (A, B), so is $\delta_{(A,B)}(\mathcal{P})$.*

Proof. We consider each item separately.

(1) Consider each condition in Definition 7.15. Observe that $\mathcal{P}_1 \sqcup \mathcal{P}_2$ is the pointwise join of the two parameters. It follows that for every $C \in \mathit{Clause}$ and $x \in \mathit{Var}(C)$, if $\mathcal{P}_1(C)(t) \neq \emptyset$ and $\mathcal{P}_2(C)(t) \neq \emptyset$, then $(\mathcal{P}_1 \sqcup \mathcal{P}_2)(C)(t) \neq \emptyset$. The same argument applies for the other two locality conditions.

(2) The sets $Var(A) \setminus Var(B)$ and $Var(B) \setminus Var(A)$ are identical in both (A, B) and (B, A). To preserve locality, every $x \in Var(A) \setminus Var(B)$ must be labelled R by $\widehat{\mathcal{P}}$. As $\mathcal{P}$ is locality preserving, these variables are labelled L and by the definition of $\widehat{\mathcal{P}}$, will be labelled R. A symmetric argument applies for $x \in Var(B) \setminus Var(A)$

We prove the second property by induction on the structure of resolution proofs.

(*Base case*) Consider $C \in A \cup B$ and the corresponding e-clauses $E \in T_{\mathcal{P}}(A, B)$ and $F \in T_{\widehat{\mathcal{P}}}(B, A)$. For every $t \in C$, if $\mathcal{P}(C)(x) = \text{L}$, then $\widehat{\mathcal{P}}(C)(x) = \text{R}$. It follows from the definition of $T_{\mathcal{P}}$ and $T_{\widehat{\mathcal{P}}}$ that $int(E) = \neg int(F)$. Observe in addition that $df(E) = \widehat{df(F)}$.

(*Induction step*) For a derived clause $C = \text{Res}(x, C_1, C_2)$ and consider the corresponding e-clauses $E = \text{iRes}(x, E_1, E_2)$ and $F = \text{iRes}(x, F_1, F_2)$ derived in $Int_{\mathcal{P}}$ and $Int_{\widehat{\mathcal{P}}}$, respectively. For the induction hypothesis, assume that $int(E_1) = \neg int(F_1)$ and $df(E_1) = \widehat{df(F_1)}$ and likewise for E_2 and F_2. For the induction step, consider the ProofRes rule in Definition 7.14. There are three cases for defining $int(E)$. If case L applies in $Int_{\mathcal{P}}$, then, by the induction hypothesis, case R applies for $Int_{\widehat{\mathcal{P}}}$. That is, $int(E) = I_1 \vee I_2$ and $int(F) = \neg I_1 \wedge \neg I_2$, so $int(E) = \neg(int(F))$ as required. The other cases are similar.

(3) This property holds because of the equality $\mathcal{P}(C)(x) = (\delta_{(A,B)}(\mathcal{P}))(C)(x)$ for the case that C is in $A \cup B$ and x is in $V_A \cup V_B$. ⊣

Abstract Domains of Parameters

Algorithms derived from Int_{HKP}, Int_M and the parameterised system have a running time that is linear in proof size, however Int_{HKP} and Int_M are more space efficient because they do not modify the distinction function. Intuitively, an interpolation system is space efficient if the value of the distinction function at a pivot variable does not change in a proof. Formally, a parameter $\mathcal{P}$ is *derivation invariant* with respect to (A, B) if every e-clause E derived from (A, B) in $Int_{\mathcal{P}}$ and every $C \in A \cup B$ satisfies $df(E)(x) = \mathcal{P}(C)(x)$ whenever x is in $Var(cl(E)) \cap Var(C)$.

Example 7.28. Consider the pair (A, B) and the parameters $\mathcal{P}_1$ and $\mathcal{P}_3$ in Example 7.24. For every clause C derived from (A, B) and corresponding e-clause E in the example, $df(E)(x)$ is the same as $\mathcal{P}(C')(x)$, where $C' \in (A, B)$. The parameters in Example 7.26 are also derivation invariant. In contrast, the parameter $\mathcal{P}_2$ in Example 7.24 is not derivation invariant because the value of the distinction function at a_2 changes in the proof. ⌟

We identify a family of abstractions that give rise to derivation invariant parameters. These abstractions are defined over partitions of *Prop*. A *partition* π of a set S is a set of disjoint subsets of S, called *blocks*, that are pairwise disjoint and whose disjoint union is S. Let $[x]_\pi$ denote the block containing $x \in S$. A partition π is *coarser than* a partition π', denoted $\pi \preceq \pi'$, if for every block $\beta \in \pi$, there is a block $\beta' \in \pi'$ such that $\beta \subseteq \beta'$. It is known that the set of partitions forms a complete lattice. Let $(\text{Part}(Prop), \preceq, \sqcup, \sqcap)$ be the lattice of partitions of *Prop*. For a CNF pair (A, B), let $\pi^{(A,B)}$ denote the partition given below.

$$\pi^{(A,B)} \triangleq \left\{ \{x \mid x \in V_A\}, \left\{x \mid x \in V_{(A,B)}\right\}, \{x \mid x \in V_B\}, \right.$$

$$\{x \mid x \notin Var(A) \cup Var(B)\}\}$$

Given a partition $\pi \in \text{Part}(Prop)$ we define a function Υ_π that maps a parameter to another one, assigning the same symbol in $\mathcal{S}$ to variables in the same block.

$$\begin{aligned}\Upsilon_\pi(\mathcal{P}) &\triangleq \mathcal{P}', \text{ where}\\ \mathcal{P}'(C)(x) &\triangleq \bigcup_{C' \in Clause} \bigcup_{y \in [x]_\pi} \mathcal{P}(C')(y) \text{ for } C \in Clause \text{ and } x \in Prop.\end{aligned}$$

A parameter $\mathcal{P}$ is *partitioning* if $\Upsilon_\pi(\mathcal{P}) = \mathcal{P}$ for some $\pi \in \text{Part}(Prop)$. In Theorem 7.30, we show that each function Υ_π defines an abstract domain of parameters and relate such parameters to derivation invariance and locality preservation.

Example 7.29. Consider the CNF pair (A, B) in Example 7.26 with the three partitions below.

$$\begin{aligned}\pi_A &= \{\{x | x \in V_A\}, \{x | x \notin V_A\}\},\\ \pi_B &= \{\{x | x \in V_B\}, \{x | x \notin V_B\}\}\\ \pi^{(A,B)} &\triangleq \text{ as given above}\end{aligned}$$

Assume that $Var(A \cup B) = Prop$. The parameters $\mathcal{P}_4, \mathcal{P}_5$ and $\mathcal{P}_6$ are partitioning, as witnessed by the partitions π_A, π_B and $\pi^{(A,B)}$ respectively.

Consider the CNF pair (A, B), the parameters $\mathcal{P}_1, \mathcal{P}_2$ and $\mathcal{P}_3$ and the partition $\pi = \{Prop\}$. Observe that $V_A = V_B = \emptyset$, so $\mathcal{P}_1$ and $\mathcal{P}_3$ are partitioning with respect to π. However, $\mathcal{P}_2$ is not partitioning. ⌟

Theorem 7.30. *Partitioning parameters have the properties below.*

1. *The function Υ_π is a closure operator.*
2. *A partitioning parameter is derivation invariant.*
3. *If $\mathcal{P}_1$ and $\mathcal{P}_2$ are partitioning, so are $\mathcal{P}_1 \sqcup \mathcal{P}_2$, $\widehat{\mathcal{P}}_1$ and $\delta_{(A,B)}(\mathcal{P}_1)$*
4. *If $\mathcal{P}$ is locality preserving and $\pi \preceq \pi^{(A,B)}$, then $\Upsilon_\pi(\mathcal{P})$ is locality preserving.*
5. *For every pair (A, B) the coarsest partition π for which $\Upsilon_\pi(\Lambda_{(A,B)}) \subseteq \Lambda_{(A,B)}$ holds is $\pi^{(A,B)}$.*

Proof. We consider each case separately.

(1) We show that Υ_π is a closure operator. The function is extensive because for all $C \in Clause$ and $x \in Prop$, $\mathcal{P}(C)(x) \subseteq \Upsilon_\pi(\mathcal{P})(C)(x)$. For every $C \in Clause$ and $y \in [x]_\pi$, $\Upsilon_\pi(\mathcal{P})(C)(x) = \Upsilon_\pi(\mathcal{P})(C)(y)$, so the function is idempotent. If $\mathcal{P}_1 \sqsubseteq \mathcal{P}_2$, then for all $C \in Clause$ and $x \in Prop$, $\mathcal{P}_1(C)(x) \subseteq \mathcal{P}_2(C)(x)$. The values $\Upsilon_\pi(\mathcal{P}_1)(C)(x)$ and $\Upsilon_\pi(\mathcal{P}_2)(C)(x)$ are defined as the union over a set of variables of $\mathcal{P}_1$ and $\mathcal{P}_2$ respectively. Monotonicity follows because union is monotone.

(2) Let E be an e-clause derived with $Int_\mathcal{P}$ from (A, B). We show that $\mathcal{P}$ is derivation invariant by induction on the structure of the derivation.

(*Base Case*) If $E \in T_\mathcal{P}(A, B)$, as $\mathcal{P}$ is partitioning, $\mathcal{P}(C)(x) = df(E)(x)$ for every clause $C \in Clause$ and variable $x \in Prop$.

(*Induction Step*) Consider $E = \text{iRes}(x, E_1, E_2)$ for e-clauses E_1 and E_2. For the induction hypothesis, assume that for every C in $A \cup B$ and $x \in Var(cl(E_1)) \cap Var(C)$, the equality $df(E_1)(x) = \mathcal{P}(C)(x)$ holds and the corresponding equality holds for E_2. Consider $C \in A \cup B$ and $x \in Var(cl(E)) \cap Var(C)$. Now, x must be in $Var(cl(E_1))$ only, $Var(cl(E_2))$ only or both. If $x \in Var(cl(E_1))$ only, $df(E)(x) = df(E_1)(x)$ and by the

induction hypothesis, $df(E)(x) = \mathcal{P}(C)(x)$. The remaining cases are similar.

(3) Consider $\mathcal{P}_1$ and $\mathcal{P}_2$ which are partitioning. That is, there exist π_1 and π_2 such that $\Upsilon_{\pi_1}(\mathcal{P}_1) = \mathcal{P}_1$ and $\Upsilon_{\pi_2}(\mathcal{P}_2) = \mathcal{P}_2$. Let $\mathcal{P} = \mathcal{P}_1 \sqcup \mathcal{P}_2$ and $\pi = \pi_1 \sqcap \pi_2$. Because $\mathcal{P}_1$ and $\mathcal{P}_2$ are partitioning, it follows that for all x and $y \in [x]_\pi$, $\mathcal{P}(C)(x) = \mathcal{P}(C)(y)$. Thus, $\Upsilon_\pi(\mathcal{P}) = \mathcal{P}$ and $\mathcal{P}$ is partitioning. The other cases hold because the dual and $\delta_{(A,B)}$ are defined pointwise on variables, so the partition for $\mathcal{P}_1$ is the partition for $\widehat{\mathcal{P}}_1$ and $\delta_{(A,B)}(\mathcal{P})$.

(4) If $\pi \preceq \pi^{(A,B)}$, for every $x \in V_A$, if $y \in [x]_\pi$, then $y \in V_A$. For a locality preserving $\mathcal{P}$ and $C \in A \cup B$, it holds that $\mathcal{P}(C)(y) \subseteq \text{L}$. Hence, $\bigcup_{C \in Clause} \bigcup_{y \in [x]_\pi} \mathcal{P}(C)(y) \subseteq \text{L}$. The same applies for $x \in V_B$, so $\Upsilon_\pi(\mathcal{P})$ is locality preserving.

(5) It follows from the previous part that $\Upsilon_{\pi^{(A,B)}}(\Lambda_{(A,B)}) \subseteq \Lambda_{(A,B)}$. It suffices to show that there is no $\pi^{(A,B)} \prec \pi$ such that $\Upsilon_\pi(\Lambda_{(A,B)}) \subseteq \Lambda_{(A,B)}$ for all (A, B). We prove it by contradiction. It suffices to find a pair (A, B) and $\mathcal{P} \in \Lambda_{(A,B)}$ such that $\Upsilon_\pi(\mathcal{P}) \notin \Lambda_{(A,B)}$. Consider (A, B) with V_A, $V_{(A,B)}$ and V_B being non-empty. Let $\mathcal{P}$ map $x \in V_A$ to L, $x \in V_B$ to R and $x \in V_{(A,B)}$ to LR. Consider variables $x \in V_A, y \in V_{(A,B)}$ and $z \in V_B$. As $\pi^{(A,B)} \prec \pi$, either $[x]_\pi = [y]_\pi$, or $[y]_\pi = [z]_\pi$, or $[x]_\pi = [z]_\pi$. If $[x]_\pi = [y]_\pi$, then $\mathcal{P}(C)(x) = \text{LR}$, violating the condition $\mathcal{P}(C)(x) \subseteq \text{L}$ in Definition 7.15. Thus, $\Upsilon_\pi(\Lambda_{(A,B)}) \not\subseteq \Lambda_{(A,B)}$. The other two cases are similar, leading to a contradiction as required. ⊣

We highlight that part 5 of Theorem 7.30 applies to all (A, B) and all parameters $\mathcal{P} \in \Lambda_{(A,B)}$. For a specific parameter $\mathcal{P} \in \Lambda_{(A,B)}$ and a specific pair (A, B), there may exist $\pi^{(A,B)} \prec \pi$ such that $\Upsilon_\pi(\mathcal{P})$ is locality preserving.

Existing Systems as Abstractions

The setting of the previous section is now applied to study existing systems. We define two parameters that were shown in [D'Silva et al. 2010] to correspond to McMillan's system and the HKP system. Let (A, B) be a CNF pair. Define the value of the parameters $\mathcal{P}_M$ and $\mathcal{P}_{HKP}$ for $C \in Clause$ and $x \in Prop$ as below.

- $\mathcal{P}_M(C)(x)$ is L if $x \in V_A$ and is R otherwise.
- $\mathcal{P}_{HKP}(C)(x)$ is L if $x \in V_A$ R if $x \in V_B$ and is LR for $x \in V_{(A,B)}$.

Lemma 7.31 shows that the parameters above are two of three that exist in the coarsest partitioning abstraction defined by $\pi^{(A,B)}$. The third system, $\delta_{(A,B)}(\mathcal{P}_M)$, was also identified in [D'Silva et al. 2010] but the connections presented here were not.

Lemma 7.31. *Let (A, B) be a CNF pair. The image of $\Lambda_{(A,B)}$ under $\Upsilon_{\pi^{(A,B)}}$ is the set below.*

$$\left\{\mathcal{P}_M, \mathcal{P}_{HKP}, \delta_{(A,B)}(\mathcal{P}_M)\right\}$$

Proof. There are two steps. The first step is to show that each parameter in the lemma is a fixed point of $\Upsilon_{\pi^{(A,B)}}$. We skip this step. The second is to show that no other such fixed points exist. As only elements of $\Lambda_{(A,B)}$ are considered, assume that $\mathcal{P}$ is locality preserving. By definition of the closure

operator we have that $\Upsilon_{\pi(A,B)}(\mathcal{P}) = \mathcal{P}$ only if for every $C_1, C_2 \in \mathit{Clause}$ and $x, y \in V_{(A,B)}$, $\mathcal{P}(C_1)(x) = \mathcal{P}(C_2)(y)$. It follows that for all C and $x \in V_{(A,B)}$, $\mathcal{P}(C)(x)$ must be either L, LR or R. Thus, the only three possible parameters are the ones above. $\dashv$

The parameter $\mathcal{P}_{HKP}$ has several properties. It is the greatest locality preserving parameter with respect to the order $\sqsubseteq$. It is symmetric in the sense that $\delta_{(A,B)}(\mathcal{P}_{HKP}) = \mathcal{P}_{HKP}$. These properties, summarised below, may explain why Int_{HKP} has been discovered repeatedly.

$$\mathcal{P}_{HKP} = \bigsqcup_{\mathcal{P} \in \Lambda_{(A,B)}} \mathcal{P} \quad \text{and} \quad \mathcal{P}_M \sqcup \delta_{(A,B)}(\mathcal{P}_M) = \mathcal{P}_{HKP}$$

The Domains of E-Clauses and Clauses

We remarked earlier that an interpolation system is an extension of resolution. This intuition is now made precise using the method in [Cousot and Cousot 1992c]. E-clauses constitute a concrete domain and interpolation systems define concrete interpretations. We show that sets of clauses form an abstract domain and that the resolution rule defines a complete abstract interpretation of an interpolation system.

Recall that *iClause* is the set of e-clauses and that for $E = (C, \Delta, I)$, $cl(E) = C$. The powerset of e-clauses forms the concrete domain $(\wp(\mathit{iClause}), \subseteq, \cup, \cap)$. A parameter $\mathcal{P}$ defines an interpolation system $\mathit{Int}_{\mathcal{P}} = (T_{\mathcal{P}}, \mathrm{pRes})$, which gives rise to a concrete interpretation consisting of two functions. The translation function $T_{\mathcal{P}} : \wp(\mathit{Clause}) \times \wp(\mathit{Clause}) \to \wp(\mathit{iClause})$ and a function $\mathit{cRes} : \wp(\mathit{iClause}) \to \wp(\mathit{iClause})$ encoding the effect of the pRes rule. The function *cRes* is defined in a sequence of steps.

- $\mathit{cRes} : \mathit{Prop} \times \mathit{iClause} \times \mathit{iClause} \to \mathit{iClause}$ is defined as follows. If $E_1, E_2 \in \mathit{iClause}$ with $cl(E_1) = x \vee C$ and $cl(E_2) = D \vee \neg x$, then $\mathit{cRes}(x, E_1, E_2)$ is given by the rule in Definition 7.14. $\mathit{cRes}(x, E_1, E_2)$ is defined as $(\emptyset, \emptyset, \mathsf{false})$ otherwise.
- The function lifts to interpolating clauses as below.

$$\mathit{cRes} : \mathit{iClause} \times \mathit{iClause} \to \mathit{iClause}$$
$$\mathit{cRes}(E_1, E_2) \triangleq \{\mathit{cRes}(x, E_1, E_2) | x \in \mathit{Prop}\}$$

- Finally, a third lifting is to sets of clauses.

$$\mathit{cRes} : \wp(\mathit{iClause}) \to \wp(\mathit{iClause}) \quad \mathit{cRes}(X) \triangleq \bigcup_{E_1, E_2 \in X} \mathit{cRes}(E_1, E_2)$$

The concrete semantic object of interest is the set of e-clauses that can be derived in an interpolation system $\mathit{Int}_{\mathcal{P}}$ and the interpolants obtained from these e-clauses. These sets are defined below.

$$\mathcal{E}_{\mathcal{P}} \triangleq \mu X.(T_{\mathcal{P}}(A, B) \cup \mathit{cRes}(X))$$
$$\mathcal{I}_{\mathcal{P}} \triangleq \{\mathit{int}(E) | E \in \mathcal{E}_{\mathcal{P}} \text{ and } cl(E) = \Box\}$$

The set $\mathcal{I}_{\mathcal{P}}$ contains all interpolants that can be derived with $\mathit{Int}_{\mathcal{P}}$ from (A, B). Observe that each interpolation system $\mathit{Int}_{\mathcal{P}}$ defines a different concrete interpretation and a different set of interpolants $\mathcal{I}_{\mathcal{P}}$. Note also that the definition of *cRes* is independent of the parameter $\mathcal{P}$. Hence, to analyse the properties of the set $\mathcal{I}_{\mathcal{P}}$, we only have to analyse $T_{\mathcal{P}}$. We exploit this observation later.

We now relate resolution with interpolation systems. Define the domain $(\wp(Clause), \subseteq, \cup, \cap)$ of CNF formulae. We define the function corresponding to the resolution rule as $res : Prop \times Clause \times Clause \to Clause$ and then lift it to $res : \wp(Clause) \to \wp(Clause)$, in a manner similar to our treatment of *cRes*.

Abstraction and concretisation functions between $(\wp(iClause), \subseteq)$ and $(\wp(Clause), \subseteq)$ are defined next. Let $\alpha : \wp(iClause) \to \wp(Clause)$ be a function that maps $X \in \wp(iClause)$ to the set of clauses $cl(X)$. The function $\gamma : \wp(Clause) \to \wp(iClause)$ concretises $Y \in \wp(Clause)$ to the set of e-clauses $\{(C, \Delta, I) | C \in Y, \Delta \in \mathbb{D}, I \in Form\}$. Lemma 7.32 states that α and γ define a Galois insertion and that *res* is the best approximation of *cRes*.

What do soundness and completeness mean in this setting? If $\alpha(cRes(X)) \subseteq res(\alpha(X))$, every clause that can be derived by reasoning over interpolating clauses can also be derived by reasoning with clauses. We also want that the interpolation system can derive all clauses that can be derived by resolution.

Lemma 7.32. *The functions α and γ define a Galois insertion between $\wp(iClause)$ and $\wp(Clause)$ and satisfy the equalities $res = (\alpha \circ cRes \circ \gamma)$ and $(res \circ \alpha) = (\alpha \circ cRes)$.*

The best approximation of $T_{\mathcal{P}}$ is union: $(\alpha \circ T_{\mathcal{P}} \circ \gamma) = \cup$. The abstract semantic object corresponding to $\mathcal{E}_{\mathcal{P}}$ is the set of clauses that can be derived by resolution from (A, B). The viewpoint presented here is summarised below.

$$\mathcal{C} \triangleq \mu X.((A \cup B) \cup res(X)) = \alpha(\mathcal{E}_{\mathcal{P}})$$

The algebra $(\wp(Clause), \subseteq, \cup, res)$ is a complete abstract interpretation of $(\wp(iClause), \subseteq, T_{\mathcal{P}}, cRes)$.

SECTION SUMMARY This section showed that the parameters introduced earlier form a lattice and showed that interpolation systems can be viewed as abstractions of this lattice. A specific family of abstractions are those that induce a partition on variables. We showed that partitioning parameters are an abstraction of the space of parameters and that existing systems are maximal elements of the lattice of partitioning parameters.

7.5 LOGICAL STRENGTH AND VARIABLE ELIMINATION

Interpolation systems are used in verification tools. The performance of such a tool depends on the logical strength and size of the interpolants obtained. The influence of interpolant strength on the termination of a verification tool is discussed in [D'Silva et al. 2010]. Interpolant size affects the memory requirements of a verification tool. The set of variables in an interpolant gives an upper bound on its size, so we study the smallest and largest sets of variables that can occur in an interpolant. We now analyse the logical strength of interpolants and the set of variables that occur in an interpolant.

Logical Strength as a Precision Order

The subset ordering on the domain $\wp(iClause)$ is a *computational order*. This means that fixed points are defined with respect to the subset order. The elements of $\wp(iClause)$ can, moreover, be ordered by *precision*, where the notion of precision is application dependent. Cousot and Cousot have emphasised that although the computational and precision orders often

coincide, this is not necessary [Cousot and Cousot 1992b]. We use a precision order based on implication to model interpolant strength.

Given X and Y in $\wp(iClause)$, the set X is more precise than Y if for every interpolant in Y, there is a logically stronger interpolant in X. Formally, define the relation $\preceq_{iClause}$ on the product lattice $\wp(iClause) \times \wp(iClause)$ to satisfy $X \preceq_{iClause} Y$ if for all $E_1 \in Y$ such that $cl(E_1) = \square$, there exists $E_2 \in X$ satisfying that $cl(E_2) = \square$ and that $int(E_2) \Rightarrow int(E_1)$. Let (A, B) be a CNF pair, $\mathcal{P}_1$ and $\mathcal{P}_2$ be two parameters and $\mathcal{E}_1$ and $\mathcal{E}_2$ be the sets of e-clauses derived in these two systems. The system $Int_{\mathcal{P}_1}$ is *more precise* or *stronger* than $Int_{\mathcal{P}_2}$ if $\mathcal{E}_1 \preceq_{iClause} \mathcal{E}_2$. If *cRes* is monotone with respect to $\preceq_{iClause}$, the problem of computing logically stronger interpolants can be reduced to that of ordering translation functions by precision. However, *cRes* is not monotone with respect to $\preceq_{iClause}$ because $\preceq_{iClause}$ does not take distinction functions into account.

We now derive an order for $\wp(iClause)$ that is stronger than $\preceq_{iClause}$ and with respect to which *cRes* is monotone. The order from [D'Silva et al. 2010] is adapted to our setting. We define an order on $\mathcal{S}$ and lift it pointwise. Define the order $\preceq_{\mathcal{S}}$ on $\mathcal{S}$ as $\textsc{r} \preceq_{\mathcal{S}} \textsc{lr} \preceq_{\mathcal{S}} \textsc{l} \preceq_{\mathcal{S}} \emptyset$. The set $\mathcal{S}$ with this order forms the lattice $(\mathcal{S}, \preceq_{\mathcal{S}}, max, min)$. By pointwise lifting, we obtain the lattice $(col, \preceq_{\mathbb{D}}, \Uparrow_{\mathbb{D}}, \Downarrow_{\mathbb{D}})$. We use the symbols $\Uparrow_{\mathbb{D}}$ and $\Downarrow_{\mathbb{D}}$ to distinguish them from the computational meet and join, $\sqcup^{\mathbb{D}}$ and $\sqcap^{\mathbb{D}}$, and to emphasise the connection to logical implication.

Let $C\downarrow_A$ be the restriction of C to variables in A. Define a relation $\sqsubseteq_{iClause}$ on $\wp(iClause) \times \wp(iClause)$ as: $X \sqsubseteq_{iClause} Y$ if for each $E_1 \in Y$ there is an $E_2 \in X$ such that $cl(E_1) = cl(E_2)$, $df(E_1) \preceq_{\mathbb{D}} df(E_2)$ and $int(E_2) \Rightarrow int(E_1) \vee (cl(E_1)\downarrow_A \cap cl(E_1)\downarrow_B)$. Intuitively, in a strong interpolant, literals are added to the partial interpolant by the translation function whereas in a weaker interpolant, literals are added in the resolution step. The partial interpolant $int(E_1)$ in the definition of $\sqsubseteq_{iClause}$ is weakened with $(cl(E_1)\downarrow_A \cap cl(E_1)\downarrow_B)$ to account for this difference. Nonetheless, if $X \sqsubseteq_{iClause} Y$ and $cl(E_1) = \square$ for $E_1 \in Y$, there exists $E_2 \in X$ such that $cl(E_2) = \square$ and $int(E_2) \Rightarrow int(E_1)$. Thus, $X \sqsubseteq_{iClause} Y$ implies that $X \preceq_{iClause} Y$. Theorem 7.34 shows that *cRes* is monotone with respect to $\sqsubseteq_{iClause}$. To order interpolation systems by precision, the precision order on distinction functions is lifted pointwise to parameters to obtain the lattice $(Clause \to col, \preceq, \Uparrow, \Downarrow)$.

Example 7.33. Revisit the functions $\mathcal{P}_1, \mathcal{P}_2$ and $\mathcal{P}_3$ in Example 7.33. It holds that $\mathcal{P}_3 \preceq \mathcal{P}_2 \preceq \mathcal{P}_1$ and the corresponding interpolants imply each other. ⌟

Theorem 7.34. *Let (A, B) be a CNF pair, and $\mathcal{P}_1$ and $\mathcal{P}_2$ be locality preserving parameters for (A, B).*

1. *If $\mathcal{P}_1 \preceq \mathcal{P}_2$, then $T_{\mathcal{P}_1}(A, B) \sqsubseteq_{iClause} T_{\mathcal{P}_2}(A, B)$.*
2. *If $X \sqsubseteq_{iClause} Y$ for $X, Y \in \wp(iClause)$, then $cRes(X) \sqsubseteq_{iClause} cRes(Y)$.*
3. *The structure $(\Lambda_{(A,B)}, \preceq, \Uparrow, \Downarrow)$ is a complete lattice [D'Silva et al. 2010].*

Proof. We consider each case below.

(1) Consider $T_{\mathcal{P}_1}(A, B)$, $T_{\mathcal{P}_2}(A, B)$ and $F \in T_{\mathcal{P}_2}(A, B)$. It follows from the definition of a translation function that there exists $E \in T_{\mathcal{P}_1}(A, B)$ such that $cl(E) = cl(F)$. If $C \in A$, we also have that $int(E) \subseteq (cl(F)\downarrow_A \cap cl(F)\downarrow_B)$, and so $int(E) \Rightarrow int(F) \vee (cl(F)\downarrow_A \cap cl(F)\downarrow_B)$. If $C \in B$, then by definition, $\neg int(F) = \{t \in cl(F) | \mathcal{P}_2(cl(F))(t) = \textsc{l}\}$. Because $\mathcal{P}_2$ is locality preserving, $\neg int(F) \subseteq (cl(F)\downarrow_A \cap cl(F)\downarrow_B)$ and we can conclude that $\neg int(F) \subseteq \neg(int(E)) \vee (cl(F)\downarrow_A \cap cl(F)\downarrow_B)$ and so $int(E) \subseteq int(F) \vee (cl(F)\downarrow_A \cap cl(F)\downarrow_B)$.

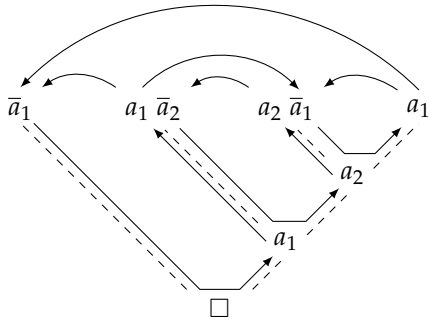


Figure 24: A resolution proof and its logical flow graph. Dashed edges represent resolution and solid edges represent flows. Every occurrence of a literal is in a cycle.

(2) Consider $X \sqsubseteq_{iClause} Y$ and $F \in cRes(Y)$. There exists $x \in Prop$ and $F_1, F_2 \in X$ such that $F = cRes(x, F_1, F_2)$. By the monotony hypothesis, there exist E_1 and E_2 in X such that $E_1 \sqsubseteq_{iClause} F_1$ and $E_2 \sqsubseteq_{iClause} F_2$. From the definition of $\sqsubseteq_{iClause}$ we conclude that $E = cRes(x, E_1, E_2)$ satisfies that $cl(E) = cl(F)$. It remains to be shown that $int(E) \Rightarrow int(F) \vee (cl(E)\downarrow_A \cap cl(E)\downarrow_B)$. This can be shown by a straightforward case analysis. ⊣

The following corollary of Theorem 7.34 formally states that if $\mathcal{P}_1 \preceq \mathcal{P}_2$, then the interpolants obtained from $Int_{\mathcal{P}_1}$ imply the interpolants obtained from $Int_{\mathcal{P}_2}$.

Corollary 7.35. *If $\mathcal{P}_1$ and $\mathcal{P}_2$ are locality preserving parameters for the CNF pair (A, B), then $\mu X.(T_{\mathcal{P}_1}(A, B) \cup cRes(X)) \preceq_{iClause} \mu X.(T_{\mathcal{P}_2}(A, B) \cup cRes(X))$.*

Variable Elimination

Every interpolant I for an unsatisfiable CNF pair (A, B) satisfies that $Var(I) \subseteq V_{(A,B)}$. We ask what the largest and smallest possible sets V are such that $Var(I) \subseteq V$. To develop some intuition for this question, we visualise the *flow* of literals in a proof. Flow graphs have been used by Carbone to study interpolant size in the sequent calculus [Carbone 1997]. We only use them informally.

Example 7.36. The flow of literals in the refutation from Example 7.24 is shown in Figure 24. Dashed edges connect antecedents with resolvents and solid edges depict flows. Each literal is a vertex in the flow graph. Positive literals flow upwards and negative literals flow downwards. Observe that a_1 appears in multiple cycles connecting literals in A, literals in B and literals in A and B. In contrast, a_2 appears in only cycle which connects an A and a B literal. Recall that every interpolant constructed from this refutation contained a_2. ⌟

Informally, a refutation defines a set of may and must variables. Every literal flowing from the A to the B part, like a_1 above, *may* be added to the interpolant. A literal that only flows from an A literal to a B literal, like a_2, *must* be added to the interpolant. To obtain the interpolant with the smallest set of variables, we need a parameter that adds to the interpolant exactly those literals that flow between A and B. We define two parameters for (A, B) as follows.

- $\mathcal{P}_{min}(C)(x)$ is L for $C \in A$ and $x \in Prop$ and is R for $C \in B$ and $x \in Prop$.
- $\mathcal{P}_{max} \hat{=} \delta_{(A,B)}(\mathcal{P}_{min})$.

Observe that both these parameters are locality preserving. Lemma 7.37 states that the parameters above determine the smallest and largest sets of variables that occur syntactically in an interpolant.

Lemma 7.37. *Let $\square$ be derived from (A, B) and E_{min} and E_{max} be the corresponding e-clauses derived in $Int_{\mathcal{P}_{min}}$ and $Int_{\mathcal{P}_{max}}$ respectively. Let E be the corresponding e-clause in $Int_{\mathcal{P}}$ for a locality preserving parameter $\mathcal{P}$. It holds that $Var(int(E_{min})) \subseteq Var(int(E)) \subseteq Var(int(E_{max}))$.*

Proof. We first show that if $x \in Var(int(E))$, then $x \in Var(int(E_{max}))$. Observe that if $x \in Var(int(E))$, then $x \in V_{(A,B)}$ and either x or $\neg x$ must occur in some $C \in A \cup B$. Let F be the clause corresponding to C in $Int_{\mathcal{P}_{max}}$. If $C \in A$, $\mathcal{P}_{max}(C)(x) = \text{R}$ and if $C \in B$, $\mathcal{P}_{max}(C)(x) = \text{L}$. In both cases, by the definition of $T_{\mathcal{P}_{max}}$ it holds that $x \in Var(int(F))$.
We show that if $x \in Var(int(E_{min}))$, then $x \in Var(int(E))$. We proceed by induction on the structure of the derivation and consider the step in which x was added to the partial interpolant. Let F be the e-clause derived by the iRes rule in $Int_{\mathcal{P}_{min}}$, given as $F = cRes(x, F_1, F_2)$ where F_1 and F_2 are antecedents. It must be that $df(F_1)(x) \cup df(F_2)(x) = \text{LR}$. Further, it must be that $x \in cl(F_1)$ and $\neg x \in cl(F_2)$ originated in A and B respectively, or vice versa, or are derived from two literals that originated from these two parts of the formulae. Let G, G_1 and G_2 be the corresponding e-clauses derived in $Int_{\mathcal{P}}$. There are three possibilities for $df(G_1)(x) \cup df(G_2)(x)$. If the value is LR, then x is added to the interpolant in this derivation step. If the value is L, then the literal that originated from B was added to the interpolant by the translation function. If the value is R, the literal originating from A was added to the interpolant by the translation function. In all cases, $x \in Var(int(G))$ as required. ⊣

We draw two further insights from Lemma 7.37. Observe that $\mathcal{P}_{min}$ and $\mathcal{P}_{max}$ are distinct from $\mathcal{P}_M$ and $\mathcal{P}_{HKP}$. A consequence is that McMillan's system and the HKP-system do not necessarily yield the interpolant with the smallest set of variables in an interpolant. This was demonstrated in Example 7.24, where the interpolants in both these systems contained the variables $\{a_1, a_2\}$, but the interpolant with the smallest set of variables contains only $\{a_2\}$.

A more general insight is a way to determine if specific interpolants cannot be obtained from a refutation. To revisit Example 7.24 (for the last time), observe that $Var(int(E_{min})) = \{a_2\}$ and that $Var(int(E_{max})) = \{a_1, a_2\}$. It follows that no interpolation system in the family we consider can be used to derive the interpolant $\neg a_1$.

SECTION SUMMARY We showed that parameters can be ordered in different ways to correlate with the logical strength of the interpolants obtained and with the set of variables in the interpolants.

7.6 BIBLIOGRAPHIC NOTES

Although Craig's interpolation theorem was published in 1957, the study of interpolation systems is relatively recent. Constructive proofs of Craig's theorem implicitly define interpolation systems. The first such proof is due to

Maehara [1961] who introduced *split sequents* to capture the contribution of the A and B formulae in a sequent calculus proof. Carbone [1997] generalised this construction to flow graphs to study the effect of cut-elimination on interpolant size. The analysis of variables in interpolants is based on ideas drawn from Carbone's work.

Interpolant size was first studied by Mundici [1982]. Krajíček [1997] observed that lower bounds on interpolation systems for propositional proofs have implications for separating complexity classes and gave an interpolation system for resolution. Pudlák [1997] published the same system simultaneously. Huang [1995] gave an interpolation system for resolution and its dual but his work appears to have gone unnoticed. We have shown that the symmetric interpolation system in these papers is maximal in the natural order derived by pointwise lifting from the powerset order on colours.

McMillan [2003] introduced interpolation to model checking by inventing a new propositional interpolation system and applying it to derive a finite-state model checker that only uses a SAT solver. The system of McMillan was dualised in this chapter using coloured interpolation systems. Yorsh and Musuvathi [2005] study interpolation for first-order theories, but also gave a new and elaborate correctness proof for the HKP-system. The invariant for the correctness proof of the labelled interpolation system generalises the induction hypothesis in their proof. The precision order $\sqsubseteq_{iClause}$ is a modification of their induction hypothesis to relate interpolants by strength rather than correctness.

The study of variables that can be eliminated from a formula is an issue of gaining interest [Gulwani and Musuvathi 2008; Kovács and Voronkov 2009]. Several researchers have noticed that an interpolant can contain fewer variables than $V_{(A,B)}$. Related observations have been made by Simmonds et al. [2007] and have often featured in personal communication. We have shown that studying variables that cannot be eliminated from a proof can provide insights into the limitations of a family of interpolation systems. Our formalisation does not directly use any existing formal framework but is heavily influenced by the flow graphs of Carbone [1997] and by labelled deduction systems [Basin et al. 2000].

The formalisation used in this chapter is based on the work of Cousot and Cousot [1992c] treating inference rules as fixed points in the context of abstract interpretation. Unlike the work in this chapter, Cousot and Cousot [1992c] do not study specific proof systems or interpolation.

8

CONCLUSION

> The product of mathematics is clarity and understanding. Not theorems, by themselves. Is there, for example any real reason that even such famous results as Fermat's Last Theorem, or the Poincaré conjecture, really matter? Their real importance is not in their specific statements, but their role in challenging our understanding, presenting challenges that led to mathematical developments that increased our understanding.
>
> The world does not suffer from an oversupply of clarity and understanding (to put it mildly). How and whether specific mathematics might lead to improving the world (whatever that means) is usually impossible to tease out, but mathematics collectively is extremely important.
>
> I think of mathematics as having a large component of psychology, because of its strong dependence on human minds. Dehumanized mathematics would be more like computer code, which is very different. Mathematical ideas, even simple ideas, are often hard to transplant from mind to mind. There are many ideas in mathematics that may be hard to get, but are easy once you get them. Because of this, mathematical understanding does not expand in a monotone direction.
>
> ...
>
> In short, mathematics only exists in a living community of mathematicians that spreads understanding and breaths life into ideas both old and new. The real satisfaction from mathematics is in learning from others and sharing with others. All of us have clear understanding of a few things and murky concepts of many more. There is no way to run out of ideas in need of clarification. The question of who is the first person to ever set foot on some square meter of land is really secondary.
>
> – William P.Thurston, *What's a Mathematician to do?*, 2010

8.1 SOME LOGICAL CONCLUSIONS

This dissertation was motivated by practical problems arising in compiler optimisation, static analysis and model checking. The first goal was to develop a framework to translate between techniques developed in model checking for reasoning about transition systems and techniques developed in static analysis for reasoning about lattices with transformers. Chapter 3 contributed to this goal by establishing a translation between algebras for linear- and branching-time logics and sets of traces. A second goal was to generalise techniques for reasoning about equivalences between transition systems to equivalences between lattices with transformers. This goal was achieved in Chapter 5 with the notion of subsumption, which generalises simulation to non-distributive lattices. A third goal was to study SAT solvers from an abstract interpretation perspective. This goal was achieved in Chapter 6 by showing that two components in modern solvers can be understood as transformer application in abstract domains. The final goal was to understand and extend interpolation algorithms. This goal was achieved in Chapter 7 using coloured interpolation systems.

The work in this dissertation is a first step towards a greater programme of unification of topics in program verification and logic. Within program verification three broad families of techniques are symbolic execution, which is based on satisfiability solvers, model checking, which involves temporal logics and transition systems, and static analysis, which is formulated over lattices. Four broad areas of logic pertinent to this discussion are automated deduction, the model theory of first-order and temporal logics, particularly techniques such as bisimulation and Ehrenfeucht-Fraïsse games, the automata-theoretic treatment of temporal logics, and algebraic logic. There are very strong connections between all the areas mentioned above and only some of them have been made explicit in the literature.

The verification techniques mentioned above can be viewed as algorithmic counterparts of the logical areas. Symbolic execution is based on automatic theorem provers and satisfiability solvers. Symbolic model checking when combined with abstraction is used to manipulate quotients of transition systems. Bisimulation and simulation induce quotients that preserve logical properties in a given logic. Techniques such as Counterexample-Guided Abstraction Refinement can be viewed as computing approximations of the bisimulation quotient. Automata-theoretic model checking directly implements the correspondence between temporal logics and automata. Algebraic logic gives lattice-based semantics to logics. The results of static analysis approximate the algebraic semantics of a formula in a logic.

The aim of the unification programme we are proposing is to make such correspondences mathematically precise and, preferably, algorithmic. Such a unification has theoretical and practical applications. The theoretical application is to transfer abstractions and decidability results from one area to another. The practical reward is access to the advantages of different techniques by novel combinations that cannot be achieved if the techniques are treated as black boxes. In the rest of this chapter, we discuss several extensions of our work

8.2 IMMEDIATE EXTENSIONS

The results of this dissertation immediately apply to the following static problems in logic, static analysis and model checking.

PROCEDURE SUMMARISATION FOR NON-DISTRIBUTIVE DOMAINS Computing procedure summaries is a well-known problem in static analysis. Existing summarisation techniques apply to completely additive functions over finite, Boolean lattices. The techniques represent the function as a graph and then compute the transitive closure of such graphs using context-free reachability algorithms. There is no established technique for summarisation in non-Boolean abstractions.

Existing summarisation procedures compute a summary by manipulating the relational representation of a function on a lattice. Discrete duality reveals that such procedures are computing the transitive closure of a transition system. The results of Chapter 3 demonstrate how to derive relational representations of additive functions over non-Boolean lattices. An algorithmic application of these results is to construct procedure summaries for static analyses over non-Boolean domains.

COUNTEREXAMPLE GENERATION IN ABSTRACT DOMAINS Model checkers generate counterexample traces for properties that fail to verify. Such traces have great diagnostic value. Counterexample generation is not standard in static analysis and is difficult to even define.

Counterexample generation in abstract domains can be examined from the viewpoint of Chapter 3. A trace is a sequence of states, which are in turn sequences of join-irreducibles of a perfect Boolean lattice. The notion of a counterexample can be generalised to non-Boolean lattices by considering sequences of join-irreducibles related by the successor transformer. Rather than finding a sequence of states, counterexample generation in static analysis becomes the problem of producing a sequence of join-irreducibles that refutes a correctness property.

Since join-irreducibles form a poset, counterexamples in non-Boolean domains can be ordered. A minimal counterexample represents the maximal level of precision that a static analyser can deliver. If a minimal counterexample is spurious, the analysis can only be improved by modifying the abstract transformers, or by moving to a more precise domain. A maximal counterexample represents the greatest generalisation of an error that preserves irreducibility with respect to abstract elements. Such a notion is useful for eliminating redundant false alarms in static analysers.

LINEAR-TIME BRANCHING-TIME SPECTRUM The Linear-time branching-time spectrum is a classification of various notions of process equivalence. Elements of the spectrum are identified by examining equivalences in the literature and relating them to known equivalences. One question is whether there is a technique to generate new equivalences and their characterisations on demand.

The entire linear-time branching-time spectrum can be viewed as an abstraction of the set of monotone functions between two algebras. The question of generating elements of the spectrum can be viewed as that of constructing an abstraction of the family of monotone functions. Recall that subsumptions and bisubsumptions are parameterised by the signature of a logic. One can generate various subsumptions by applying signature abstraction to a rich logical signature. This method of subsumption generation will generate some, but not all elements of the linear-time branching-time spectrum, because the logics have a very restricted grammatical form. The first issue is to identify which parts of the spectrum can be generated in this manner.

The second issue is to extend subsumption to characterise logical languages not generated by the simple, closed grammar. For example, the literature contains a notion of two-simulation which characterises equivalence between transition systems in a logic where formulae contain at most one negation. The most general answer to address this second issue should develop a framework to generate variants of subsumption for every syntactic abstraction of a logical language.

SATISFIABILITY MODULO THEORIES The work in Chapter 6 covered only two of several important components that contribute to the success of propositional satisfiability solvers. A question is whether the analysis presented in Chapter 6 extends to other satisfiability procedures, particularly to those for reasoning about first-order theories.

We believe that the core data structures in solvers for propositional and first-order logics can all be viewed as abstract domains. Equality graphs used by equality logic solvers, and difference graphs used by difference logic solvers, both form abstractions, in the Galois connection sense, of the space of first-order structures over which these theories are interpreted. Both the congruence closure algorithm used by equality solvers and the Bellman-Ford algorithm implemented by difference logic solvers can naturally be viewed as a fixed point computations. We believe that the abstract satisfaction framework can be used to show that these algorithms are greatest fixed point computations involving abstractions of the model transformer.

EQUALITY INTERPOLATION The work on interpolant strength in the dissertation was published in 2010. Since then, interpolant construction using coloured interpolation systems has been implemented by several researchers. A challenging problem is to develop new interpolation algorithms for quantifier-free first-order theories.

We believe that the techniques used in this dissertation to study propositional interpolation extend to studying interpolation in equality logic and to weak fragments of arithmetic, such as Boolean combinations of inequalities over two variables with unit coefficients. In a propositional resolution refutation, a single proof rule suffices for deduction and interpolants can be constructed using an extension of this rule. An interpolation system for the theory of equality must deal with the transitivity of equality. Interpolants cannot be constructed in a completely local manner because an equality refutation may involve chains of equalities over variables that cannot appear in an interpolant. We believe that coloured interpolation systems can be extended to deal with equality chains and to identify interpolation systems for equality.

8.3 LONGER TERM EXTENSIONS

There are several longer term projects that we envision to extend the work in this dissertation towards a deeper unification of verification and deduction techniques. We sketch these below.

FIRST-ORDER STRUCTURES All the structures considered in Chapter 3 were propositional. The structures generated by programs involve variable and function symbols, and are first-order. A unification of verification techniques for first-order properties must first give algebraic semantics for first-order temporal logics.

The algebraic treatment of first-order structures is highly non-trivial and there are several competing proposals [Németi 1991]. The formulation chosen should lend itself easily to representing existing static analyses techniques. The domain for constant propagation is the simplest first-order structure that first-order algebras should be able to represent. At present, it appears that categorical logic [Pitts 2000] provides the most natural setting in which to formalise static analyses that manipulate variables.

A potential long term extension of the results in Chapter 3 is to extend trace algebras to first-order trace algebras. A first-order trace algebra should support operations for quantification. We anticipate that transition system representations of non-Boolean first-order temporal structures will be non-trivial to derive but also expect that they will have numerous applications.

EHRENFEUCHT-FRAÏSSE GAMES The first-order notion corresponding to simulation and bisimulation is Ehrenfeucht-Fraïsse games. The original structural characterisation of equivalence of first-order structures by Fraïsse was algebraic. A shortcoming of the results in Chapter 5 is that subsumption and bisubsumption cannot be applied to reason about equivalence between first-order structures.

We believe that a first-order extension of subsumption will be indispensable for understanding the algorithmic requirements of static analysis and compiler optimisation procedures. A first-order subsumption for non-distributive abstractions would aid in proving that optimisations such as constant folding and loop-invariant code motion are semantics preserving. To enable automation of such proofs these generalisations should also have fixed point characterisations.

The long term extension of Chapter 5 that we propose is to combine categorical logic with Ehrenfeucht-Fraïsse games to derive first-order extensions of subsumption and bisubsumption. These techniques should then be applied to determine exactly what combination of operations is required to formalise Partial Redundancy Elimination. Algorithms to compute these equivalences between finite procedures should be derived and applied to automatically prove the correctness of compiler optimisations.

A GRAND UNIFIED TOOL When we began this work, we did not envision a unified tool that could be applied to a diverse range of practical problems. This work has convinced us that it is feasible, and in fact desirable, to develop such tools. The problems of satisfiability in a theory, bounds checking for variables that appear in floating point computations or as array indices, and of determining the nullness of pointer variables are among a host of problems that we believe can benefit from a practical marriage of techniques developed by the satisfiability, model checking and program analysis communities.

We believe the results in this dissertation provide the conceptual clarity required to execute the design and implementation of such tools. The first step is to identify the lattice-based structures required for each problem and to implement these structures using a well-defined, uniform interface. Such a programming discipline already exists in the static analysis community but we believe it can also be followed in the implementation of satisfiability solvers. The second step is to formulate satisfiability and verification algorithms as fixed point computations involving abstract transformers. If this formulation is domain agnostic, it is feasible to implement a portfolio of algorithms that only make assumptions about the abstract domain interfaces provided by different tools.

Finally, we believe that elements of discrete duality constructions can be supported by tools and will be useful in practice. The results in Chapter 3 show that dual constructions can be derived by decomposing lattice elements using meet- and join-irreducibles. Abstract domain implementations currently support standard lattice operations and transformers for assignments and conditionals. If these implementations are extended to support meet- and join-irreducible decomposition, it would be possible for tools to support the construction of relational representations, which have applications for counterexample generation and procedure summarisation.

FINAL WORD

This dissertation has argued that logical reasoning and abstraction can and should be treated as first-class citizens of every enterprise for reasoning about programs. The combination of logic and abstraction can be achieved in a systematic manner by combining the algebraic semantics of a logic with abstract interpretation. Such a combination is the gateway to a unification programme that we believe brings great conceptual clarity, uncovers a universe of fascinating symmetries and will have significant practical consequences.

We expect that carrying out such a unification programme will bring mathematical and algorithmic challenges. The route we are proposing moves away from traditional logical structures first to algebraic ones and second to abstract interpretations of algebraic structures. We have found that proofs in this setting are shorter than existing proofs, but have also been more difficult for us to discover. On the flip side, algebraic proofs are known to be more amenable to mechanisation. Designing new algorithms within an abstract, algebraic framework is a challenging task, but implementing such algorithms has led to performance improvements on difficult practical problems.

We expect also that carrying out such a unification programme will bring social challenges. Working with structures that are two-steps removed from the traditional conception of logic has not been and will not be easy. We have found that communicating this work has been a daunting task. Our conclusion stands. We have discovered great beauty, conceptual economy and practical benefits in the rigorous combination of logic and abstract interpretation. It is our hope that the verification community will support and extend these efforts.

Abdulla, P. A., Čerāns, K., Jonsson, B., and Tsay, Y.-K. 2000. Algorithmic analysis of programs with well quasi-ordered domains. *Information and Computation 160*, 1-2, 109–127.

Abramsky, S. 1987. Domain theory and the logic of observable properties. Ph.D. thesis, University of London.

Abramsky, S. 1991a. A domain equation for bisimulation. *Information and Computation 92*, 161–218.

Abramsky, S. 1991b. Domain theory in logical form. *Annals of Pure and Applied Logic 51*, 1-2, 1–77.

Abramsky, S. and Jung, A. 1994. Domain theory. In *Handbook of logic in computer science (vol. 3)*, S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, Eds. Oxford University Press, Oxford, UK, 1–168.

Ball, T., Podelski, A., and Rajamani, S. K. 2001. Boolean and cartesian abstraction for model checking C programs. In *Proc. of Tools and Algorithms for the Construction and Analysis of Systems*. TACAS 2001. Springer-Verlag, Munich, Germany, 268–283.

Basin, D., D'Agostino, M., Gabbay, D., Matthews, S., and Viganò, L., Eds. 2000. *Labelled Deduction*. Kluwer Academic Publishers, Dordrecht.

Beckman, N. E., Nori, A. V., Rajamani, S. K., and Simmons, R. J. 2008. Proofs from tests. In *Proc. of Software Testing and Analysis*. ACM Press, New York, NY, USA, 3–14.

Beyer, D., Henzinger, T. A., and Théoduloz, G. 2007. Configurable software verification: Concretizing the convergence of model checking and program analysis. In *Proc. of Computer Aided Verification*, W. Damm and H. Hermanns, Eds. Lecture Notes in Computer Science. Springer-Verlag, Munich, Germany, 504–518.

Bibel, W. 2007. Early history and perspectives of automated deduction. In *Proceedings of the 30th annual German conference on Advances in Artificial Intelligence*. Springer-Verlag, Berlin, Heidelberg, 2–18.

Birkhoff, G. 1967. Lattice theory. Vol. 25. American Mathematical Society, 420 pp+.

Blackburn, P., Benthem, J. F. A. K. v., and Wolter, F. 2006. *Handbook of Modal Logic, Volume 3 (Studies in Logic and Practical Reasoning)*. Elsevier Science Inc., New York, NY, USA.

Blackburn, P., de Rijke, M., and Venema, Y. 2001. *Modal Logic*. Cambridge University Press, Cambridge.

Blake, A. 1937. Canonical expressions in boolean algebra. Ph.D. thesis, University of Chicago.

Boole, G. 1847. *The Mathematical Analysis of Logic*.

Boole, G. 1854. *An investigation into the Laws of Thought, on Which are founded the Mathematical Theories of Logic and Probabilities*.

Bundy, A. 1999. A survey of automated deduction. In *Artificial intelligence today*, M. J. Wooldridge and M. Veloso, Eds. Springer-Verlag, Berlin, Heidelberg, 153–174.

Burch, J. R., Clarke, E. M., McMillan, K. L., Dill, D. L., and Hwang, L. J. 1992. Symbolic model checking: 10^{20} states and beyond. *Information and Computation 98*, 2, 142–170.

Burris, S. and Sankappanavar, H. P. 2000. *A Course in Universal Algebra*. Distributed electronically.

Carbone, A. 1997. Interpolants, cut elimination and flow graphs for the propositional calculus. *Annals of Pure and Applied Logic 83*, 3, 249–299.

Cattani, G. L. and Winskel, G. 2005. Profunctors, open maps and bisimulation. *Mathematical. Structures in Comp. Sci. 15*, 3, 553–614.

Chomsky, N. and Schutzenberger, M. P. 1963. The algebraic theory of context-free languages. *Computer programming and formal systems 35*, 118–161.

Christensen, A. S., Møller, A., and Schwartzbach, M. I. 2003. Precise analysis of string expressions. In *Proc. of Static Analysis Symposium*. Lecture Notes in Computer Science, vol. 2694. Springer-Verlag, 1–18.

Clarke, E. 1977. Program invariants as fixed points. In *Symposium on Foundations of Computer Science*. IEEE Computer Society, Washington, DC, USA, 18–29.

Clarke, E. M. and Emerson, E. A. 1981. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs*, D. Kozen, Ed. Lecture Notes in Computer Science, vol. 131. Springer-Verlag, Munich, Germany, 52–71.

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

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

Cousot, P. 2005. Abstract interpretation. MIT course 16.399.

Cousot, P. and Cousot, R. 1975. Static verification of dynamic type properties of variables. Research report R.R. 25, Laboratoire IMAG, Université scientifique et médicale de Grenoble, Grenoble, France. Nov.

Cousot, P. and Cousot, R. 1977. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. of Principles of Programming Languages*. ACM Press, New York, NY, USA, 238–252.

Cousot, P. and Cousot, R. 1979. Systematic design of program analysis frameworks. In *Proc. of Principles of Programming Languages*. ACM Press, New York, NY, USA, 269–282.

Cousot, P. and Cousot, R. 1992a. Abstract interpretation and application to logic programs. *Journal of Logic Programming 13*, 2–3, 103–179.

Cousot, P. and Cousot, R. 1992b. Abstract interpretation frameworks. *Journal of Logic and Computation 2*, 4 (Aug.), 511–547.

Cousot, P. and Cousot, R. 1992c. Inductive definitions, semantics and abstract interpretations. In *Proc. of Principles of Programming Languages*. ACM Press, New York, NY, USA, 83–94.

Cousot, P. and Cousot, R. 1995a. Compositional and inductive semantic definitions in fixpoint, equational, constraint, closure-condition, rule-based and game-theoretic form, invited paper. In *Proceedings of the Seventh International Conference on Computer Aided Verification, CAV '95*, P. Wolper, Ed. Springer-Verlag, Berlin, Germany, Liège, Belgium, Lecture Notes in Computer Science 939, 293–308.

Cousot, P. and Cousot, R. 1995b. Formal language, grammar and set-constraint-based program analysis by abstract interpretation. In *Proc. of the Conference on Functional Programming Languages and Computer Architecture*. ACM Press, New York, NY, USA, 170–181.

Cousot, P. and Cousot, R. 2000. Temporal abstract interpretation. In *Proc. of Principles of Programming Languages*. ACM Press, New York, NY, USA, 12–25.

Cousot, P. and Cousot, R. 2007. Grammar analysis and parsing by abstract interpretation. In *Program analysis and compilation, theory and practice*, T. Reps, M. Sagiv, and J. Bauer, Eds. Springer-Verlag, Munich, Germany, 175–200.

Cousot, P. and Cousot, R. 2009. Bi-inductive structural semantics. *Information and Computation 207*, 258–283.

Cousot, P., Cousot, R., and Mauborgne, L. 2011. The reduced product of abstract domains and the combination of decision procedures. In *Proc. of the Conference on Foundations of Software Science and Computation Structures*. Springer-Verlag, Munich, Germany, 456–472.

Craig, W. 1957. Linear reasoning. A new form of the Herbrand-Gentzen theorem. *Journal of Symbolic Logic 22*, 3, 250–268.

Davey, B. A. and Priestley, H. A. 1990. *Introduction to lattices and order*. Cambridge University Press, Cambridge, UK.

Davis, M. 2001. The early history of automated deduction. In *Handbook of Automated Reasoning*, J. A. Robinson and A. Voronkov, Eds. Elsevier and MIT Press, Essex, UK, 3–15.

Davis, M., Logemann, G., and Loveland, D. 1962. A machine program for theorem-proving. *Communications of the ACM 5*, 394–397.

Davis, M. and Putnam, H. 1960. A computing procedure for quantification theory. *Journal of the ACM 7*, 201–215.

Dedekind, R. 1897. Über Zerlegungen von Zahlen durch ihre grössten gemeinsamen Teiler. *Festschrift der Technischen Hochschule zu Braunschweig bei Gelegenheit der 69 Versammlung Deutscher Naturforscher und Ärzte*, 1–40.

D'Silva, V., Kroening, D., Purandare, M., and Weissenbacher, G. 2010. Interpolant strength. In *Proc. of the Conference on Verification, Model Checking, and Abstract Interpretation*, G. Barthe and M. Hermenegildo, Eds. Lecture Notes in Computer Science. Springer-Verlag, Munich, Germany.

Dunn, J. M. 1995. Positive modal logic. *Studia Logica 55*, 2, 301–317.

Dunn, J. M., Gehrke, M., and Palmigiano, A. 2005. Canonical extensions and relational completeness of some substructural logics. *Journal of Symbolic Logic 70*, 3, 713–740.

Emerson, E. A. and Halpern, J. Y. 1985. Decision procedures and expressiveness in the temporal logic of branching time. *Journal of Computer and System Sciences 30*, 1–24.

Filé, G., Giacobazzi, R., and Ranzato, F. 1996. A unifying view of abstract domain design. *ACM Computing Surveys 28*, 2 (June), 333–336.

Finkel, A. and Schnoebelen, P. 2001. Well-structured transition systems everywhere! *Theoretical Computer Science 256*, 1-2, 63–92.

Flon, L. and Suzuki, N. 1978. Consistent and complete proof rules for the total correctness of parallel programs. In *Symposium on Foundations of Computer Science*. IEEE Computer Society, Washington, DC, USA, 184–192.

Gehrke, M. 2006. Generalized Kripke frames. *Studia Logica 84*, 2, 241–275.

Gehrke, M. and Jónsson, B. 1994. Bounded distributive lattices with operators. *Mathematica Japonica 40*, 2, 207–215.

Giacobazzi, R. and Quintarelli, E. 2001. Incompleteness, counterexamples, and refinements in abstract model-checking. In *Symposium on Static Analysis*. Springer-Verlag, London, UK, 356–373.

Giacobazzi, R., Ranzato, F., and Scozzari, F. 2000. Making abstract interpretations complete. *Journal of the ACM 47*, 2, 361–416.

Ginsburg, S. and Rice, H. G. 1962. Two families of languages related to algol. *Journal of the ACM 9*, 350–371.

Glabbeek, R. J. v. 1993. The linear time - branching time spectrum II. In *Proc. of Concurrency Theory*. CONCUR '93. Springer-Verlag, London, UK, 66–81.

Goldblatt, R. 2003. Mathematical modal logic: a view of its evolution. *J. of Applied Logic 1*, 5-6, 309–392.

Granger, P. 1992. Improving the results of static analyses programs by local decreasing iteration. In *Foundations of Software Technology and Theoretical Computer Science*. Springer-Verlag, Munich, Germany, 68–79.

Grätzer, G. 2011. *Lattice Theory: Foundation*. Springer Basel, Basel.

Gulavani, B. S., Chakraborty, S., Nori, A. V., and Rajamani, S. K. 2008. Automatically refining abstract interpretations. In *Proc. of Tools and Algorithms for the Construction and Analysis of Systems*. Lecture Notes in Computer Science, vol. 4963. Springer-Verlag, Munich, Germany, 443–458.

Gulwani, S. and Musuvathi, M. 2008. Cover algorithms and their combination. In *Proc. of the European Symposium on Programming*. Lecture Notes in Computer Science, vol. 4960. Springer-Verlag, Munich, Germany, 193–207.

Guo, S.-y. and Palsberg, J. 2011. The essence of compiling with traces. In *Proc. of Principles of Programming Languages*. ACM Press, New York, NY, USA, 563–574.

Hennessy, M. and Milner, R. 1985. Algebraic laws for nondeterminism and concurrency. *Journal of the ACM 32*, 1 (Jan.), 137–161.

Henzinger, M. R., Henzinger, T. A., and Kopke, P. W. 1995. Computing simulations on finite and infinite graphs. In *Annual Symposium on Foundations of Computer Science*. FOCS '95. IEEE Computer Society, Washington, DC, USA, 453–.

Holley, L. H. and Rosen, B. K. 1980. Qualified data flow problems. In *Proc. of Principles of Programming Languages*. ACM Press, New York, NY, USA, 68–82.

Hopenwasser, A. 1990. Complete distributivity. In *Proc. of the Symposia in Pure Mathematics*. Vol. 51. American Mathematical Society, 285–305.

Huang, G. 1995. Constructing Craig interpolation formulas. In *Proc. of the Conference on Computing and Combinatorics*. Lecture Notes in Computer Science, vol. 959. Springer-Verlag, Munich, Germany, 181–190.

Istrail, S. 1982. Generalization of the ginsburg-rice schützenberger fixed-point theorem for context-sensitive and recursive-enumerable languages. *Theoretical Computer Science 18*, 333–341.

Jensen, T. P. 1991. Strictness analysis in logical form. In *Conference on Functional programming languages and computer architecture*. Springer-Verlag, New York, NY, USA, 352–366.

Johnstone, P. 1986. *Stone Spaces*. Cambridge Studies in Advanced Mathematics. Cambridge University Press.

Jónsson, B. and Tarski, A. 1952. Boolean algebras with operators. *American Journal of Mathematics 74*, 1, 127–162.

Kam, J. B. and Ullman, J. D. 1977. Monotone data flow analysis frameworks. *Acta Informatica 7*, 305–317.

Keller, R. M. 1976. Formal verification of parallel programs. *Communications of the ACM 19*, 371–384.

Kildall, G. A. 1973. A unified approach to global program optimization. In *Proc. of Principles of Programming Languages*. ACM Press, New York, NY, USA, 194–206.

Kovács, L. and Voronkov, A. 2009. Interpolation and symbol elimination. In *Proc. of Conference on Automated Deduction*. Lecture Notes in Computer Science, vol. 5663. Springer-Verlag, Munich, Germany, 199–213.

Kozen, D. 1982. Results on the propositional μ-calculus. In *Proceedings of the 9th Colloquium on Automata, Languages and Programming*. Springer-Verlag, London, UK, 348–359.

Krajíček, J. 1997. Interpolation theorems, lower bounds for proof systems, and independence results for bounded arithmetic. *Journal of Symbolic Logic 62*, 2, 457–486.

Kripke, S. 1959. A completeness theorem in modal logic. *J. Symb. Log. 24*, 1, 1–14.

Kueker, D. W. 1975. Back-and-forth arguments and infinitary logics. In *Infinitary Logic: In Memoriam Carol Karp*. Lecture Notes in Mathematics, vol. 492. Springer Berlin Heidelberg, 17–71.

Kundu, S., Tatlock, Z., and Lerner, S. 2009. Proving optimizations correct using parameterized program equivalence. In *Proc. of Programming Language Design and Implementation*. ACM Press, New York, NY, USA, 327–337.

Lacey, D., Jones, N. D., Van Wyk, E., and Frederiksen, C. C. 2004. Compiler optimization correctness by temporal logic. *Higher Order and Symbolic Computation 17*, 3, 173–206.

Lemmon, E. J. 1966a. Algebraic semantics for modal logics i. *Journal of Symbolic Logic 31*, 1, 46–65.

Lemmon, E. J. 1966b. Algebraic semantics for modal logics ii. *Journal of Symbolic Logic 31*, 2, 191–218.

Loiseaux, C., Graf, S., Sifakis, J., Bouajjani, A., and Bensalem, S. 1995. Property preserving abstractions for the verification of concurrent systems. *Formal Methods in Systems Design 6*, 1 (Jan.), 11–44.

Lynch, N. and Vaandrager, F. 1995. Forward and backward simulations I: untimed systems. *Information and Computation 121*, 2, 214–233.

Maehara, S. 1961. On the interpolation theorem of Craig (in Japanese). *Sûgaku 12*, 235–237.

Malacaria, P. 1995. Studying equivalences of transition systems with algebraic tools. *Theoretical Computer Science 139*, 1-2 (Mar.), 187–205.

Malik, S. and Zhang, L. 2009. Boolean satisfiability from theoretical hardness to practical success. *Communications of the ACM 52*, 76–82.

Mancosu, P., Ed. 2008. *Interpolations. Essays in Honor of William Craig*. Synthese, vol. 164:3. Springer-Verlag.

Manna, Z. and Pnueli, A. 1992. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, Munich, Germany.

McMillan, K. L. 1993. *Symbolic Model Checking*. Kluwer Academic Publishers, Norwell, MA, USA.

McMillan, K. L. 2003. Interpolation and SAT-based model checking. In *Proc. of Computer Aided Verification*. Lecture Notes in Computer Science, vol. 2725. Springer-Verlag, Munich, Germany, 1–13.

McMillan, K. L. 2005. An interpolating theorem prover. *Theoretical Computer Science 345*, 101–121.

McMillan, K. L. 2010. Lazy annotation for program testing and verification. In *Proc. of Computer Aided Verification*. Springer-Verlag, Munich, Germany, 104–118.

Milner, R. 1971. An algebraic definition of simulation between programs. In *International Joint Conference on Artificial Intelligence*, D. C. Cooper, Ed. William Kaufmann, 481–489.

Milner, R. 1989. *Communication and concurrency*. PHI Series in computer science. Prentice Hall.

Möller, B., Höfner, P., and Struth, G. 2006. Quantales and temporal logics. In *Algebraic Methodology and Software Technology*, M. Johnson and V. Vene, Eds. Lecture Notes in Computer Science, vol. 4019. Springer-Verlag, Munich, Germany, 263–277.

Moskewicz, M. W., Madigan, C. F., Zhao, Y., Zhang, L., and Malik, S. 2001. Chaff: engineering an efficient SAT solver. In *Proc. of the Design Automation Conference*. ACM, New York, NY, USA, 530–535.

Muchnick, S. S. 1997. *Advanced Compiler Design and Implementation*. Morgan Kaufmann.

Mundici, D. 1982. Complexity of Craig's interpolation. *Fundamenta Informaticae 5*, 261–278.

Németi, I. 1991. Algebraization of quantifier logics, an introductory overview. *Studia Logica: An International Journal for Symbolic Logic 50*, 3/4, pp. 485–569.

Nielson, F. and Nielson, H. R. 2010. Model checking *is* static analysis of modal logic. In *Foundations of Software Science and Computational Structures*. Lecture Notes in Computer Science, vol. 6014. Springer-Verlag, Munich, Germany, 191–205.

Nieuwenhuis, R., Oliveras, A., and Tinelli, C. 2006. Solving SAT and SAT modulo theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). *Journal of the ACM 53*, 937–977.

Okhotin, A. and Reitwiessner, C. 2010. Conjunctive grammars with restricted disjunction. *Theoretical Computer Science 411*, 2559–2571.

Paulson, L. C. 1994. A fixedpoint approach to implementing (co)inductive definitions. In *Proc. of Conference on Automated Deduction*. Springer-Verlag, Munich, Germany, 148–161.

Pitts, A. M. 2000. Categorical logic. In *Algebraic and Logical Structures*, S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, Eds. Handbook of Logic in Computer Science, vol. 5. Oxford University Press, 39–128.

Pnueli, A. 1977. The temporal logic of programs. In *Foundations of Computer Science*. IEEE Computer Society, Washington, DC, USA, 46–57.

Poizat, B. 2000. *A course in model theory: an introduction to contemporary mathematical logic*. Universitext Series. Springer-Verlag.

Pudlák, P. 1997. Lower bounds for resolution and cutting plane proofs and monotone computations. *Journal of Symbolic Logic 62*, 3, 981–998.

Queille, J.-P. and Sifakis, J. 1982. Specification and verification of concurrent systems in CESAR. In *5th International Symposium on Programming*. Springer-Verlag, Munich, Germany, 337–351.

Raney, G. N. 1952. Completely distributive complete lattices. *Transactions of the American Mathematical Society 3*, 677–680.

Ranzato, F. and Tapparo, F. 2007. Generalized strong preservation by abstract interpretation. *J. of Logic and Computation 17*, 1, 157–197.

Ranzato, F. and Tapparo, F. 2008. Generalizing the paige–tarjan algorithm by abstract interpretation. *Information and Computation 206*, 5, 620–651.

Reynolds, M. 2001. An axiomatization of full computation tree logic. *Journal of Symbolic Logic 66*, 3 (Sept.), pp. 1011–1057.

Rival, X. and Mauborgne, L. 2007. The trace partitioning abstract domain. *ACM Transactions on Programming Languages and Systems 29*, 5, 26.

Robinson, J. A. 1965. A machine-oriented logic based on the resolution principle. *Journal of the ACM 12*, 1 (Jan.), 23–41.

Rutten, J. J. M. M. 2000. Universal coalgebra: a theory of systems. *Theoretical Computer Science 249*, 1, 3–80.

Sangiorgi, D. 2009. On the origins of bisimulation and coinduction. *ACM Transactions on Programming Languages and Systems 31*, 4 (May), 15:1–15:41.

Schachte, P. and Søndergaard, H. 2006. Closure operators for robdds. In *Proc. of the Conference on Verification, Model Checking, and Abstract Interpretation* (2005-12-21), E. A. Emerson and K. S. Namjoshi, Eds. Lecture Notes in Computer Science, vol. 3855. Springer-Verlag, Munich, Germany.

Schmidt, D. A. 1998. Data flow analysis is model checking of abstract interpretations. In *Proc. of Principles of Programming Languages*. ACM Press, New York, NY, USA, 38–48.

Schmidt, D. A. 2008. Internal and external logics of abstract interpretations. In *Verification, model checking, and abstract interpretation*. Springer-Verlag, Berlin, Heidelberg, 263–278.

Schmidt, D. A. 2012. Inverse-limit and topological aspects of abstract interpretation. *Theoretical Computer Science 430*, 23–42.

Schmidt, D. A. and Steffen, B. 1998. Program analysis *s* model checking of abstract interpretations. In *Symposium on Static Analysis*. Lecture Notes in Computer Science, vol. 1503. Springer-Verlag, Munich, Germany, 351–380.

Schmitz, S. 2007. Approximating context-free grammars for parsing and verification. Ph.D. thesis, Université de Nice-Sophia Antipolis, France.

Scott, D. 1969. Lattice of flow diagrams. Programming Research Group PRG 03, Oxford University, Computing Laboratory, Oxford, UK. Nov.

Silva, J. a. P. M. and Sakallah, K. A. 1996a. GRASP – a new search algorithm for satisfiability. In *International Conference on Computer-Aided Design*. IEEE Computer Society Press, 220–227.

Silva, J. P. M. and Sakallah, K. A. 1996b. Conflict analysis in search algorithms for satisfiability. In *Proc. of the International Conference on Tools with Artificial Intelligence*. IEEE Computer Society, Washington, DC, USA, 467–.

Simmonds, J., Davies, J., Gurfinkel, A., and Chechik, M. 2007. Exploiting resolution proofs to speed up LTL vacuity detection for BMC. In *Proc. of Formal Methods in Computer-Aided Design*. IEEE Computer Society, 3–12.

Steffen, B. 1991. Data flow analysis as model checking. In *Theoretical Aspects of Computer Software*. Lecture Notes in Computer Science, vol. 526. Springer-Verlag, Munich, Germany, 346–365.

Steffen, B. 1993. Generating data flow analysis algorithms from modal specifications. *Science of Computer Programming 21*, 2 (Oct.), 115–139.

Stone, M. H. 1936. The Theory of Representation for Boolean Algebras. *Transactions of the American Mathematical Society 40*, 1 (July), 37–111.

Tarski, A. 1935. Zur grundlegung der booleschen algebra, i. *Fundamenta Mathematicae 24*, 177–198.

van Glabbeek, R. 2001. The linear time – branching time spectrum I. the semantics of concrete, sequential processes. In *Handbook of Process Algebra*, J. Bergstra, A. Ponse, and S. Smolka, Eds. Elsevier, Essex, UK, 3 – 99.

von Karger, B. 2002. *Temporal algebra*. Springer-Verlag, New York, NY, USA, 309–385.

Ward, M. 1942. The closure operators of a lattice. *Annals of Mathematics 43*, 191–196.

Wassermann, G., Gould, C., Su, Z., and Devanbu, P. 2007. Static checking of dynamically generated queries in database applications. *ACM Transactions on Software Engineering Methodology 16*, 4, 14.

Wegman, M. N. and Zadeck, F. K. 1991. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems 13*, 181–210.

Yorsh, G. and Musuvathi, M. 2005. A combination method for generating interpolants. In *Proc. of Conference on Automated Deduction*. Lecture Notes in Computer Science, vol. 3632. Springer-Verlag, Munich, Germany, 353–368.

Zhang, L. and Malik, S. 2002. The quest for efficient boolean satisfiability solvers. In *Proc. of Computer Aided Verification*. Springer-Verlag, Munich, Germany, 17–36.

Index of Symbols

Asg, assignments domain, 140
σ, assignment, 140
Table, truth tables, 144
$\mathbb{B}$, Booleans, 26
$\mathbb{N}$, Naturals, 26
$\mathbb{N}_\infty$, Naturals with ∞, 26
$\mathbb{O}$, Ordinals, 26
$\mathbb{Z}$, Integers, 26
$\mathbb{Z}_\infty$, Integers with ∞, 26
ω, 26
$[\cdot]_R$, equivalence class, 26
$\cdot/R$, quotient, 26
$[\cdot]_R$, block, 26
$\bar{s}$, sequence, 27
$len(\bar{s})$, sequence length, 27
$g[\bar{a} \Mapsto \bar{c}]$, sequence substitution, 27
$\bigsqcap$, meet, 28
$\tilde{f}$, De Morgan dual, 30
$\sqcap$, meet, 28
$\sqcup$, join, 28
$\sqsubset$, strict partial order, 27
$\sqsubseteq$, partial order, 27
Fix, fixed points, 29
gfp, greatest fixed point, 29
lfp, least fixed point, 29
$\dot{\sqcap}$, pointwise meet, 29
$\dot{\sqcup}$, pointwise join, 29
$\dot{\sqsubseteq}$, pointwise order, 29
$\mathscr{P}(\cdot)$, powerset, 26
$S\!\downarrow$, downward closure, 34
$\mathscr{D}(\cdot)$, downset, 34
ℓ, lower approximation, 36
u, upper approximation, 36
Atom, 33
CoAtom, 33
$Irr_\sqcap$, meet-irreducibles, 33
$Irr_\sqcup$, join-irreducibles, 33
$S\!\uparrow$, upward closure, 34
$\mathscr{U}(\cdot)$, up-set, 34
Sig, signature, 37
$balg(\cdot)$, relational structure to BAO, 39
$rel(\cdot)$, BAO to relational structure, 39
lset,set to algebra, 50, 52, 55
palg,algebra to set, 52, 54
palg,propositional algebra to set, 50
Prop, 49
post, successor, 58
pre, predecessor, 58
upost, universal successor, 58
upre, universal predecessor, 58
next, next, 75
prev, previous, 75
some, some trace, 75
::=, rule, 91
$root(\cdot)$, root, 91
$sym(\cdot)$, symbol label, 91
PAsg, partial assignments, 146
$aprune_\pi$, abstract pruning, 149
clneg, clausal negation, 149
$ded_{\varphi,\Delta}$, deduction transformer, 148
$prune_\pi$, pruning transformer, 149
bcp, BCP, 148
unit, unit rule, 146
$cmod_\varphi$, countermodels, 141
mod_φ, models, 141
$ucmod_\varphi$, universal countermodels, 141
$umod_\varphi$, universal models, 141
L, colour, 160
R, colour, 160

$\mathscr{G}(\cdot)$, Galois stable set, 36
$R \circ S$, composition, 26

$f \circ g$, function composition, 27

env, syntax environment, 92
$\mathsf{env}_\perp$, syntax environment, 93
In, instantiation, 92
inst, instantiation function, 92

Index of Technical Terms

abstraction
  signature, 95
  syntactic, 95
adjoint, 30
algebra
  Sig-, 37
  Boolean, 28
  conjunctive propositional, 54
  distributive propositional, 51
  propositional, 50
ancestor, 159
anti-symmetric, 26
assignment, 140
assignment transformer
  countermodels, 141
  models, 141
  universal countermodels, 141
  universal models, 141
atom, 33

Boolean algebra with operators, BAO, 37
Boolean Constraint Propagation (BCP), 148

closure
  lower, 32
  upper, 32
coatom, 33
colouring function, 160
complement, 28
completely conjunctive, 98
congruence, 37
conjugate, 37
cover, 27

Dedekind law, 38
Dedekind-MacNeille completion, 36
distributive law, 28
domain
  assignments, 140
  instantiation functions, 93
  syntax environments, 93
  syntax trees, 93
downset, 34
dual
  De Morgan, $\tilde{f}$, 30
  order, 30
  self, 30

equivalence
  class, $[\cdot]_R$, 26
  quotient, A/R, 26
  relation, 26
  representative, 26
expression, 92

fixed point, 29
  extremal, 29
  greatest, 29
  Kleene, 29
  Knaster-Tarski, 29
  least, 29
function
  additive, 28
  bottom-strict, 29
  codomain, 26
  completely additive, 28
  completely multiplicative, 28
  composition, 27
  domain restriction, 27
  extensive, 28
  idempotent, 28
  identity, 26
  image, 27
  involution, 28
  monotone, 28
  multiplicative, 28
  nullary, 27
  range restriction, 27
  reductive, 28
  strict additive, 29
  strict multiplicative, 29
  substitution, 27
  top-strict, 29

Galois
  connection, 30
  injection, 31
  insertion, 31
  isomorphism, 31
  reduction, 31
  stable, $\mathscr{G}(\cdot)$, 36
  surjection, 31
Galois relation, 54
grammar, 91
  closed, 91
  rule, 91

homomorphism, 37

image, 26

instantiation, 92
interpolant, 160
interpolation system, 161
irreducible
  join, 33
  meet, 33
isomorphism, 37

join
  dense, 33
  perfect, 33
Jónsson and Tarski theorem, 38

labelled
  poset, 51
  set, 50
  twoset, 53
language, 92
lattice, 28
  Boolean, 28
  bounded, 28
  complemented, 28
  complete, 28
  distributive, 28
  perfect, 33
  perfect Boolean, 33
  perfect distributive, 34

meet
  dense, 33
  perfect, 33
meta
  symbol, 91
  variable, 91

order
  dual, 30
  embedding, 28
  isomorphism, 28

partial assignments, 146
partial order, 27
partition, 26
  block, 26
pointwise
  extension, 29
  join,$\dot{\sqcup}$, 29
  meet,$\dot{\sqcap}$, 29
  order,$\dot{\sqsubseteq}$, 29
poset, 27
powerset $\mathscr{P}(\cdot)$, 26
powerset extension, 29
predecessor, 58
predecessor algebra, 59
  conjunctive, 70
  distributive , 65
preimage, 26
preorder, 27
principal
  downset, 54
  Galois-stable set, 54

quotient, 37

reflexive, 26
relation
  composition, 26
relational structure, 38
resolution, 158
  ancestor, 159
  ancestor-free, 159
  proof, 159
  refutation, 159
  rule, 158

semi-commutation condition, 63
sequence
  m-termed, 27
  infinite, 27
  length, 27
  substitution, 27
  two-way, 27
signature, 37
  propositional, 49
state algebra, 59
structure
  conjunctive, 53
  monotone, 51
sub-language, 95
sub-signature, 95
substitution, 91
successor, 58
symmetric, 26
syntax
  environment, 92
  meaning, 92
  transformer, 93
  tree, 91
    root, 91
    well formed, 91
  trees domain, 93

trace algebra, 76
trace signature, 75
trace system, 75
transformer
  abstract, 40
  abstract pruning, 149
  best abstract, 41

clausal negation, 149
concrete, 40
deduction, 148
existential predecessor, 58
existential successor, 58
pruning, 149
sound, 40
syntax, 93
universal predecessor, 58
universal successor, 58
transition system, 58
conjunctive, 69
monotone , 62
transitive, 26
translation function, 161
truth table, 144, 145
two-way total, 74
twoset, 36

uniform composition, 91
unit rule, 146
up-set, 34

Ward's theorem, 32