# ANSWER SET PROGRAMMING FOR SYNTHESIS, INFERENCE, AND SEARCH

**ZIYI YANG**

*(B.Eng., Peking University)*

**A THESIS SUBMITTED**

**FOR THE DEGREE OF DOCTOR OF PHILOSOPHY**

**DEPARTMENT OF COMPUTER SCIENCE**

**NATIONAL UNIVERSITY OF SINGAPORE**

**2026**

Supervisor:
Dr. Ilya SERGEY

Examiners:
A/P Chin Wei Ngan
A/P Yap Hock Chuan, Roland

# DECLARATION

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

This thesis has not been submitted for any degree or examination in any other university or institution previously.

### Use of Generative AI

I acknowledge the use of generative artificial intelligence (AI) tools in the preparation of this thesis, solely for language refinement, structuring and proofreading purposes. All AI-generated outputs have been critically reviewed, edited, and verified by me. I confirm that the intellectual content, analysis, and conclusions presented are entirely my own, and I take full responsibility for the work submitted.

Ziyi YANG

Full Name of Candidate

November 7, 2026

Date

## Acknowledgments

Defining completeness of an acknowledgement is hard. This acknowledgement is not meant to be an enumeration. Everything happens at the moment is a consequence of everything in the past. I am grateful for all the people who have been part of my life. I may be pretending to be a person who seldom needs a lot of support, but I did secretly steal that from many people.

Let me start from the thesis. The way to become a mature researcher is bitter, and I'm very grateful to my thesis advisor Ilya Sergey, who continuously provides all kinds of support. On the research side, he is not just a smart and knowledgeable researcher leading me to the diverse research agendas, but also a patient and open-minded mentor allowing me to explore my own research interests with his valuable high-level guidance. And of course is not just about research: VERSE lab is no doubt an amazing team, and everyone can tell how hard it is to build a warm family, but Ilya spent much effort to maintain a friendly and supportive environment for all of us even if he is busy at all times.

I'm also feeling fortunate to have Wei Ngan Chin and Roland Yap as my thesis examiners. Prof. Chin is also my QE examiner, and I keep receiving his valuable feedback and comments in those milestones or seminars. Prof. Yap is an expert in constraint programming and logic programming, so all discussions with him are very inspiring.

All of those projects I have been involved in are not possible without the collaboration with many people. Regardless of whether resulting in published papers, I have learned a lot from all of them. In the chronological order, I would like to thank my collaborators: Yingfei Xiong, Xiaokang Qiu, Yanjun Wang, Zhenjiang Hu, Zhichao Guan, Yushuo Xiao, Ilya Sergey, Alex Potanin, David Young, Matthew Flatt, Kiran Gopinathan, Yunjeong Lee, George Pîrlea. Moreover, other than obtaining valuable feedback from conference reviewers, many friendly and constructive comments from non-collaborators are also very helpful: other than my labmates, I would like to thank: Hila Peleg, Peter Müller, Roland Yap, Matthew Flatt.

The 5 years of PhD life is actually a long journey much more than just research. The lab support is always fruitful from both inside the VERSE lab and the whole PLSE group. I would like to thank all the VERSE Lab members: Andreea Costea, Kiran Gopinathan, Yunjeong Lee, George Pîrlea, Vladimir Gladshtein, Qiyuan Zhao, Yuxi Ling, Valentin Mikhalchuk, Yueyang Feng, Dipesh Kafle, Thibault Dardinier, Linard Arquint, Vitaly Kurin, Vasilii Nesterov; and interns: Gokul Rajiv, Ziyu Mao, Vikram Goyal, Sean Wang, Theodore Leebrant, Mayank Keoliya. My life has been enriched by their fruitful personalities and kindness. The PLSE lab is a big family, and I would like to name Zhendong Ang for his help in many aspects. Hopefully both small and big families keep warm forever, regardless the temperature of AC indoor.

I will also sincerely thank my family. Considering the educational circumstances in East Asia, it's a privilege to have parents who allow me to be myself. Also, my big family is always supportive and encouraging. My grandparents (including passed away grandmothers, both of whom I unfortunately did not manage to meet for the last chance), my parents, my uncle and aunt, my cousins are all loving and caring.

Unlike family, friends are not necessarily lifelong. But I am lucky to have those friends who are always in the heart of each other with me. The oldest one can be traced back to me under 1, and many of recent friends are met during my PhD life. I'm going to list them in the order of when we first meet: Huaijin Zhang, Jialuo Wang, Minghao Zheng, Yingqi Liu, Boning Zhang, Rui Xu, Yuhang Zhou, Zuoer Chen, Zeyu Ma, Yan Li, Chaokai Xu (I have to highlight him since we accompanied each other for the whole university life starting from undergraduate to the end of PhD), Zijing Peng, Qihao Liang. I specially thank my girlfriend Xinyun Yan, with whom I have spent the most amazing time in the past months.

Lastly, I will not forget to thank myself. The five-year journey is like a newborn, and I am brave enough to face all the challenges, with mostly a peaceful mind. There are certain words I hope to leave for myself for the whole life: freeing oneself from the tyranny of external validation is a necessary condition for inner peace.

# Contents

**Declaration** i

**Acknowledgments** ii

**Summary** viii

**List of Tables** x

**List of Figures** xi

**1 Introduction** 1

1.1 The Bottleneck of Formal Specification . . . . . 1

1.2 Answer Set Programming as a Unifying Theme . . . . . 2

1.3 Motivations, Contributions, and Related Work . . . . . 3

1.3.1 Automating Heap Predicate Synthesis . . . . . 3

1.3.2 Inferring First-Order Invariants . . . . . 4

1.3.3 Bridging High-Level Logic and Low-Level Solvers 6

1.4 Thesis Statement . . . . . 8

1.5 Thesis Overview . . . . . 8

**2 A Brief Primer on Answer Set Programming** 10

2.1 Propositional Satisfiability (SAT) . . . . . 10

2.2 Datalog . . . . . 11

2.3 Answer Set Programming . . . . . 11

**3 Sippy: Inductive Synthesis of Heap Predicates** 13

3.1 Introduction . . . . . 13

3.2 Overview and Key Ideas . . . . . 14

3.2.1 Background: Separation Logic . . . . . 14
3.2.2 From Memory Graphs to Heap Predicates . . . . 14
3.2.3 Synthesis without Negative Examples . . . . . 16
3.2.4 Automatically Generating Positive Examples . . . 19
3.2.5 Putting It All Together . . . . . 20
3.3 Positive-Only Predicate Learning . . . . . 21
3.3.1 Normalising the Positive-Only Learning . . . . . 22
3.3.2 An Algorithm for Positive-Only Learning . . . . . 22
3.4 Separation Logic Predicate Synthesis via Sippy . . . . . 25
3.4.1 SL Predicates: Basics and Intricacies . . . . . 26
3.4.2 Ensuring SL Validity in Prolog . . . . . 29
3.4.3 The Sippy Algorithm . . . . . 30
3.5 Automated Heap Graph Generation via Grippy . . . . . 33
3.6 Evaluation and Discussion . . . . . 34
3.6.1 Benchmarking Predicate Synthesis . . . . . 35
3.6.2 The Utility of Sippy . . . . . 40
3.6.3 Failure Modes and Future Work . . . . . 44

**4 FORCE: Inductive First-Order Formula Synthesis 48**
4.1 Introduction . . . . . 48
4.2 Overview . . . . . 49
4.2.1 Background: First-Order Logic and Transition Systems . . . . . 49
4.2.2 Problem Definition . . . . . 50
4.2.3 Taxonomy of Existing Invariant Inference Algorithms 52
4.2.4 FORCE: ASP-based Synthesis + Orthogonal Slices 56
4.3 Static Search Spaces of FO Formulas in ASP . . . . . 57
4.3.1 Encoding the Enumeration . . . . . 57
4.3.2 Encoding Pruning by Redundancy . . . . . 60
4.4 Dynamic Search Spaces via Orthogonal Slices . . . . . 61
4.4.1 Implication Graphs . . . . . 61
4.4.2 From the Formula to the Search Space . . . . . 62
4.4.3 Slicing DNF Modulo Clauses . . . . . 66

4.4.4 The Synthesis Algorithm . . . . . 68
4.5 Experimental Evaluation . . . . . 69
4.5.1 Implementation and Settings . . . . . 69
4.5.2 Results and Analysis . . . . . 70
4.5.3 On Composability of FORCE . . . . . 72

**5 SetLah!: A Language for Combinatorial Search 73**
5.1 Introduction . . . . . 73
5.2 Overview . . . . . 74
5.2.1 A First Look at SetLah!: on Euler's Conjecture . 74
5.2.2 The Design Triangle of SetLah! . . . . . 81
5.3 Language Design of SetLah! . . . . . 82
5.3.1 PCLP: The Kernel Language . . . . . 83
5.3.2 SetLah! as Stratified PCLP . . . . . 86
5.4 Translating SetLah! Programs to ASP . . . . . 88
5.4.1 Translating a Block . . . . . 89
5.4.2 Equivalence Rewriting Rules . . . . . 90
5.4.3 Phase 1: Conjunction Splitting . . . . . 92
5.4.4 Phase 2: Conversion to IC Form . . . . . 92
5.4.5 Correctness . . . . . 94
5.4.6 Extending SetLah! with Set Operations . . . . . 95
5.5 Evaluation . . . . . 96
5.5.1 RQ1: Modelling Common Combinatorial Tasks . 97
5.5.2 RQ2: Performance and Scalability . . . . . 99
5.5.3 RQ3: Qubit Layout Synthesis as a Case Study . . 103
5.5.4 Discussion . . . . . 106
5.6 Related Work . . . . . 107
5.6.1 Transformation of Formulas and Programs . . . . 107
5.6.2 Relevant Paradigms for Constraint Solving . . . . 108
5.6.3 Higher-Order Satisfiability and Constraint Programming . . . . . 109

**6 Discussion and Concluding Remarks** **113**
6.1 The Positions of the Thesis . . . . . 113
6.1.1 The Impact of This Thesis for PL Research . . . 113
6.1.2 The Impact of This Thesis in the LLM Era . . . . 114
6.2 Future Directions . . . . . 115
6.2.1 Combining ASP with E-graphs . . . . . 115
6.2.2 Closing the Loop: From Synthesis to Verification 116
6.2.3 Synthesising Inductive Synthesisers . . . . . 116
6.2.4 SetLah! as a Platform and an Interface for LLMs 116
6.3 Epilogue . . . . . 117

**Bibliography** **119**

**A Deferred Proofs and Model Listings** **141**
A.1 Deferred Proofs . . . . . 141
A.1.1 Translation Correctness . . . . . 141
A.1.2 Finiteness of the Reachable Set . . . . . 143
A.2 Set Operator Extensions . . . . . 149
A.3 Model Listings . . . . . 150
A.3.1 CNF Encoding for Graeco-Latin Squares . . . . . 150
A.3.2 SetLah Model for Q-Synth . . . . . 151

# Summary

Formal specifications are the bedrock of reliable software systems, yet writing them remains notoriously difficult. It requires a dual expertise: deep knowledge of the system being specified and fluency in the rigid logical formalisms of formal methods. This *Specification Bottleneck* manifests in two dimensions: the *Discovery Problem*, where the correct invariants need to be inferred from examples; and the *Modelling Problem*, where even a conceptually understood problem must be encoded in a solver-friendly form. This thesis addresses both dimensions of the Specification Bottleneck: it uses **Answer Set Programming** (ASP) to tackle the Discovery Problem in formal methods, and addresses the Modelling Problem for ASP itself.

Chapter 3 and Chapter 4 address the Discovery Problem through inductive synthesis and inference. **Sippy** automatically synthesises recursive predicates for heap-manipulating data structures directly from memory instances. By encoding the hypothesis space and domain-specific pruning rules as an ASP problem, Sippy efficiently discovers both the structural shape and the data invariants of the target predicate, improving over prior shape analysis tools on the evaluated benchmark data structures. **FORCE** addresses the inference of logical invariants for distributed protocols. It introduces a unified taxonomy of existing inference approaches and a novel *orthogonal slices* technique, which partitions the exponentially large search space into manageable chunks amenable to incremental ASP-based pruning. FORCE demonstrates significant performance improvements over state-of-the-art invariant inference tools on non-trivial distributed protocols.

Chapter 5 addresses the Modelling Problem—an irony that emerges from the preceding two chapters: ASP, our chosen solution, is itself subject to the Specification Bottleneck, since encoding complex constraints directly in ASP requires expertise in stable model semantics and is opaque and error-prone. Crucially, this Modelling Problem is not unique to ASP—the same bottleneck afflicts SAT, where even conceptually simple constraints demand

manual, error-prone compilation to Conjunctive Normal Form. **SetLah!** is a semi-declarative language that resolves this bottleneck for ASP, and in doing so offers a higher-level alternative to direct ASP or CNF encodings. It allows users to express combinatorial search problems using arbitrary First-Order Logic constraints within a structured, block-based syntax, and automatically compiles these into efficient ASP programs, bridging the gap between classical logic and solver-level encodings. Experiments across diverse benchmarks show that, with the efficient ASP system Clingo as the backend, SetLah! programs are competitive with, and often faster than, selected SAT-based tools.

The three contributions are unified by the thesis statement: Chapter 3 and Chapter 4 use ASP to address the Specification Bottleneck in formal methods—the former for heap data structures, the latter for distributed protocol invariants; and Chapter 5 tackles the Specification Bottleneck of ASP itself, making the engine that the first two chapters rely on accessible to non-experts.

# List of Tables


3.1 Statistics on synthesised predicates and tool comparison . . 37
3.2 Statistics on generating input memory graphs . . . . . 41
3.3 Programs synthesised by SuSLik using Sippy predicates . . . 45

4.1 Connection between synthesis and ASP . . . . . 67

5.1 Equivalence rewriting rules for FOL-to-IC translation . . . . 91
5.2 Modelling problems: FOL vs. FOL+Set Operations in SetLah! 97
5.3 PAR-2 Performance of GL Square in seconds . . . . . 100
5.4 Scalability comparison: Alloy vs. SetLah! . . . . . 102

A.1 Set operator translations to ASP . . . . . 149

# List of Figures


3.1 Sorted list heap graph examples . . . . . 15
3.2 Effect of examples on search . . . . . 18
3.3 The workflow of Sippy . . . . . 21
3.4 The grammar of the SL predicates in Backus–Naur form (BNF). 26
3.5 Testing time with different input graphs and node counts . . 39
3.6 Incomplete oracle and invalid DLL graph . . . . . 43
3.7 Input program with inferred sorted-list predicate . . . . . 43
3.8 SuSLik output: sorted DLL to SLL . . . . . 45

4.1 Workflow of FORCE . . . . . 56
4.2 The synthesis loop: imperative vs. ASP . . . . . 58
4.3 Formulas and their implication graph . . . . . 62
4.4 Invariant synthesis results . . . . . 70

5.1 Latin squares and Graeco–Latin square of order 3 . . . . . 75
5.2 The Design Triangle of SetLah!. . . . . 81
5.3 Grammar of PCLP and SetLah! . . . . . 84
5.4 Cactus plots for Q-Synth qubit layout synthesis . . . . . 104

# Chapter 1

# Introduction

## 1.1 The Bottleneck of Formal Specification

Formal specifications are the bedrock of reliable software systems. Whether verifying the memory safety of a pointer-based data structure [123], the correctness of a distributed consensus protocol [158], or the behaviour of hardware designs [51, 125], the entire edifice of formal methods rests on the existence of precise descriptions of system behaviour. However, despite decades of progress in verification engines—from SAT [19]/SMT solvers [42, 41], to interactive theorem provers [34, 43]—the acquisition of these specifications remains a critical bottleneck.

Writing formal specifications is notoriously difficult. It requires a dual expertise: deep knowledge of the system being designed and fluency in the rigid, often non-intuitive logical formalisms required by verification tools. This *Specification Bottleneck* manifests in two distinct dimensions:

1. **The Discovery Problem:** For existing systems or complex algorithms, discovering the correct invariants (e.g., "what defines a valid binary search tree?" or "what is the inductive invariant of Paxos?") is usually a labour-intensive, error-prone manual process.

2. **The Modelling Problem:** Even when the problems to be solved are conceptually understood, encoding them into a solver-friendly format—such as Conjunctive Normal Form (CNF) for SAT solvers or logic programs for answer set programming—is often opaque.

Tackling the Discovery Problem calls for *inductive synthesis*: reasoning from concrete examples to a general formal specification. This inductive step is inherently hard—the hypothesis space of candidate specifications grows exponentially with formula size—and demands an engine that can exhaustively and efficiently search that space under domain-specific constraints. The Modelling Problem requires bridging the semantic gap between high-level intent and low-level constraints.

## 1.2 Answer Set Programming as a Unifying Theme

At the core of this thesis lies **Answer Set Programming** (ASP, [24, 106]), a declarative programming paradigm geared towards difficult combinatorial search problems. Unlike traditional imperative programming, where the programmer specifies *how* to compute a solution, ASP allows the programmer to specify *what* constitutes a solution (an "answer set") and relies on highly optimised solvers to perform the exhaustive search. This makes ASP a natural engine for inductive synthesis: one can directly encode the *shape* of a valid hypothesis and let the solver enumerate all candidates. At the same time, ASP is itself a formal method backend that demands expert knowledge to use effectively—making it a direct, self-contained instance of the Modelling Problem.

In the context of this thesis, ASP serves a dual role:

- **An *engine* for synthesis and inference (Step 1):** In the first half of the thesis, we use ASP to explore the vast space of possible logical formulas and find the "best" specification fitting the given examples (e.g., heap memory graphs or protocol traces).
- **A *target* for combinatorial search (Step 2):** In doing so, we discover that ASP is itself subject to the Specification Bottleneck: encoding synthesis problems directly in ASP is opaque and error-prone. In the second half, we address this by proposing a first-order

modelling language that compiles down to ASP, making the power of combinatorial search accessible to non-experts.

## 1.3 Motivations, Contributions, and Related Work

The core contribution of this thesis is a unified approach to mitigating the Specification Bottleneck using ASP, and then applying the same perspective to ASP itself. We detail the motivations, specific contributions, and positioning against related work for each phase below.

### 1.3.1 Automating Heap Predicate Synthesis

**Motivation.** Separation Logic (SL) [123, 137] has emerged as the premier formalism for specifying and verifying imperative programs that manipulate mutable, pointer-based data structures. Its success, marked by a wide range of applications from automated verification [11, 16, 29] to bug detection [99], relies on its power of compositional reasoning. However, leveraging SL requires writing inductive predicates that describe both the shape of the heap (*e.g.*, a linked list) and the properties of the data it stores (*e.g.*, sortedness). Writing these predicates manually is extremely tedious. While Inductive Logic Programming (ILP) [116, 119, 36] seems a natural fit for synthesising these specifications from concrete memory instances [21, 100, 4], traditional ILP frameworks require both positive and *negative* examples. In the domain of heap-based structures, systematically generating valid negative examples that guide the synthesiser without causing overfitting is notoriously difficult.

**Contributions.** To address this challenge, **Chapter 3** introduces Sippy, an automated framework that synthesises recursive SL predicates strictly from concrete positive data structure instances. Our key insights and contributions are:

- We formulate a novel ILP-based learning algorithm that operates *exclusively on positive examples*. This eliminates the need for the user (or an external generator) to craft negative memory graphs.
- We design a synthesis procedure that simultaneously discovers both the structural shape and the rich data invariants (*e.g.*, binary search trees, $n$-ary trees) of the target heap structures.
- We demonstrate that by encoding the search space and domain-specific pruning rules into ASP, Sippy can efficiently exhaust the hypothesis space and discover specifications that prior shape analysis tools do not infer on the evaluated benchmarks.

**Related Work.** The challenge of inferring SL predicates from data instances has been studied from several angles, none of which achieves Sippy's combination of positive-only learning, shape, and data invariant discovery. ShaPE [21] learns SL predicates from positive memory graphs via meta-interpretive learning [118], but is restricted to structural shape constraints and cannot express data payload invariants. Earlier shape-analysis work [76], as well as DOrder [172] and Evospex [114], extracts recursion patterns or data invariants from data-structure *constructor code*, requiring access to source code that Sippy does not need. Statistical approaches such as Locust [26] and recent deep-learning frameworks [113, 152] bypass constructors but offer no completeness guarantees and demand large corpora; Sippy's ASP-based exhaustive search provides completeness within the hypothesis space. None of these approaches simultaneously captures both memory *shape* and *data invariants* from positive-only heap instances, which is precisely the gap Sippy fills.

### 1.3.2 Inferring First-Order Invariants

**Motivation.** Moving beyond recursive data structures, verifying the correctness of complex distributed systems (*e.g.*, Lamport's Paxos consensus

protocol [97]) using tools like Ivy [125] and Veil [130], which rely on user-supplied inductive invariants, requires those invariants to be expressed in First-Order Logic (FOL). To achieve fully automated verification, recent efforts have attempted to infer these FO formulas directly from execution traces (positive examples) [168, 60, 169]. However, the search space for FOL formulas is astronomically large, and the objective of synthesising the *most specific* conjunction consistent with positive traces demands a fundamentally different approach from traditional ILP and bottom-up generalisation.

**Contributions.** To tackle the immense search space of FOL, **Chapter 4** introduces FORCE, a top-down, "generate-test-prune" framework that delegates the heavy lifting of formula enumeration to ASP. Our key contributions are:

- We present a unified categorisation of existing invariant inference methodologies for distributed systems, demonstrating how various domain-specific pruning techniques can be elegantly encoded as ASP constraints.
- We introduce *orthogonal slices*, a novel formula enumeration technique that partitions the massive FO search space into manageable chunks, enabling highly efficient incremental candidate pruning.
- We evaluate FORCE against state-of-the-art invariant inference tools (*e.g.*, DuoAI), demonstrating significant performance improvements on non-trivial distributed protocols. Furthermore, we show that FORCE's modular, ASP-driven design allows it to compose with and optimise other inference frameworks.

**Related Work.** Synthesising inductive FO invariants for distributed systems has attracted a flurry of recent tools, each with a different strategy that FORCE either subsumes or improves upon. Bottom-up generalisation systems, of which FOL-IC3 [92] is representative, generalise specific protocol traces upward into candidate formulas; this approach scales poorly as the

example set grows because every new trace constrains the generalisation—FORCE avoids this bottleneck entirely via top-down slicing. Popper [37] takes the structurally similar top-down generate-test-prune approach in ASP, but seeks the *most general* hypothesis that satisfies positives while refuting negatives; invariant inference has no negative examples and requires the *most specific* conjunction consistent with all positive traces—an inversion that demands the fundamentally different pruning strategy FORCE provides. Other tools such as DuoAI [168] use implication graphs to prune redundant candidates; FORCE generalises this technique as one instance of its orthogonal-slice framework and composes it with additional pruning.

### 1.3.3 Bridging High-Level Logic and Low-Level Solvers

**Motivation.** The successes of Sippy and FORCE underscore the power of Answer Set Programming (ASP) [121, 24] as a synthesis engine. However, they also expose a striking irony: ASP—our chosen solution to the Specification Bottleneck—is itself subject to the same bottleneck at the modelling level. Encoding complex synthesis rules directly in ASP is a rigid, error-prone process, a direct manifestation of the Modelling Problem. This second-order Specification Bottleneck is not unique to our work; it represents a fundamental challenge in constraint-based formal methods more broadly. Users are often forced into an uncomfortable choice between the expressive, high-level abstractions of First-Order Logic and the raw performance of low-level SAT/ASP solvers [19, 147]. This semantic gap is exacerbated by ASP's reliance on stable model semantics [72], which is powerful but notoriously non-intuitive for non-experts.

**Contributions.** To bridge this gap, **Chapter 5** introduces SetLah!, a novel language designed for "semi-declarative" modelling. SetLah! allows users to embed arbitrary FOL constraints within a structured, block-based

syntax that compiles down to efficient ASP programs. Our key contributions are:

- We define *Positive Choice Logic Programming* (PCLP), a kernel language that takes choice as a primitive, and formalise SetLah! as stratified PCLP. This gives a constructive branching-fixpoint semantics that reconciles the declarative "Guess-and-Check" paradigm of ASP with classical logic.
- We develop a modular two-phase translation pipeline that automatically converts arbitrary FOL constraints into ASP integrity constraints via conjunction splitting and quantifier elimination (with aggregate- or auxiliary-based strategies), and prove its correctness and termination.
- We evaluate SetLah! on benchmarks spanning combinatorial design (Graeco-Latin squares), software modelling (from Alloy [84]), and quantum circuit layout synthesis (Q-Synth), demonstrating competitive performance against selected SAT-based tools.

**Related Work.** Prior attempts to bridge high-level logic and solver backends each sacrifice one or more of the properties SetLah! achieves simultaneously. MiniZinc [120], Enfragmo [1], and Picat [171] offer rich declarative modelling over CP or SAT backends, but target single-shot compilation with no first-class support for multi-shot incremental solving, which is essential for the synthesis tasks in this thesis. FOLASP [45], which compiles IDP [30] programs to ASP, shares the FOL-to-ASP motivation but is based on inductive-definition semantics that diverge from stable models; its bottom-up translation also ignores grounding efficiency, resulting in poor scalability for large domains—two weaknesses SetLah! addresses directly. Alloy [84]/Kodkod [155] provide a mature FOL-like language backed by SAT, but Kodkod applies *equisatisfiable* (not model-preserving) rewrites and relies on automatic symmetry-breaking, introducing a semantic gap that SetLah! avoids by preserving models throughout compilation. Dusa [111] has evaluation semantics close to stable models, but uses its own backtracking

evaluator rather than existing optimised ASP solvers and supports only restricted constraints—SetLah! compiles to Clingo directly, inheriting its full constraint language and solver optimisations. These comparisons are developed in detail in § 5.6, which also situates SetLah! with respect to higher-order satisfiability and higher-order program constraints.

## 1.4 Thesis Statement

> *This thesis addresses both dimensions of the Specification Bottleneck: it uses Answer Set Programming (ASP) to tackle the Discovery Problem in formal methods, and enhances the modelling of ASP-based approaches to address the Modelling Problem of ASP itself.*

The first part (Chapters 3 and 4) uses ASP as an engine against the Specification Bottleneck: Sippy synthesises recursive Separation Logic heap predicates strictly from positive memory graphs, and FORCE synthesises First-Order Logic invariants for distributed protocols, both by encoding the hypothesis search space as an ASP problem. The second part (Chapter 5) turns to the Specification Bottleneck of ASP itself: encoding complex synthesis rules directly in ASP is rigid and error-prone—the same bottleneck, now one level up. To enhance the modelling of ASP-based approaches, we present SetLah!, a semi-declarative language that compiles structured FOL constraints into efficient ASP programs.

## 1.5 Thesis Overview

**Chapter 2 (A Brief Primer on ASP)** introduces Propositional Satisfiability and Datalog as logical foundations, then presents Answer Set Programming, the shared computational engine underlying all three technical contributions.

**Chapter 3 (Sippy)** introduces a method to synthesise Separation Logic predicates for data structures from positive memory graphs.

**Chapter 4 (FORCE)** presents a framework for inferring First-Order Logic invariants for protocols by pruning the search space with ASP.

**Chapter 5 (SetLah!)** details a semi-declarative language that compiles high-level modelling of combinatorial search into efficient ASP.

**Chapter 6 (Conclusion)** discusses the thesis contributions and outlines directions for future research.

# Chapter 2

# A Brief Primer on Answer Set Programming

This chapter provides a brief primer on Answer Set Programming (ASP), the shared computational engine for all technical contributions of this thesis. We first introduce Propositional Satisfiability (SAT) and Datalog as the logical foundations that ASP builds upon, then present ASP itself.

## 2.1 Propositional Satisfiability (SAT)

The SAT problem asks whether there exists an interpretation that satisfies a given propositional formula. Modern SAT solvers typically operate on formulas in *Conjunctive Normal Form* (CNF). A CNF formula $\phi$ is a conjunction of clauses, where each clause is a disjunction of literals (a variable $x$ or its negation $\neg x$).

$$\phi = \bigwedge_i C_i, \quad C_i = \bigvee_j l_{ij}$$

An interpretation $I$ (a truth assignment to variables) satisfies $\phi$, denoted $I \models \phi$, if it satisfies at least one literal in every clause. While SAT is the canonical NP-complete problem, modelling constraints in CNF often results in a loss of structural information, as high-level constraints must be compiled down to flat clauses.

## 2.2 Datalog

A (propositional) *definite logic program* is a finite set of rules of the form

$$a_0 \leftarrow a_1,\ \ldots,\ a_m.$$

where $a_0$ (*head* of the rule) and each $a_i$ ($1 \leq i \leq m$) in the *body* are propositional atoms (in concrete Prolog/ASP syntax, $\leftarrow$ is written `:-`; when rules are lifted to first-order predicates in later chapters, we follow the standard logic programming convention of writing predicate and constant symbols in lower case and variables in upper case). Each rule is a *Horn clause*: logically equivalent to $a_1 \wedge \cdots \wedge a_m \rightarrow a_0$, a clause with at most one positive literal. We restrict attention to ground atoms (*i.e.*, without variables) when discussing models. A rule with $m = 0$ is called a *fact* and is written simply as $a_0$.

An *interpretation* $I$ is a set of ground atoms. We say that the body $a_1, \ldots, a_m$ is satisfied in $I$ iff $\{a_1, \ldots, a_m\} \subseteq I$. An interpretation $I$ is a *model* of a program $P$ when for every rule $a_0 \leftarrow a_1,\ \ldots,\ a_m.$ in $P$, if the body is satisfied in $I$ then $a_0 \in I$.

The semantics of Datalog is based on the *immediate consequence operator* ($T_P$), which maps an interpretation $I$ to the set of all heads $a_0$ of rules in $P$ whose bodies are satisfied in $I$. The *least model* of a definite program $P$ is the least fixed point of $T_P$, obtained by iterating from the empty interpretation.

## 2.3 Answer Set Programming

Unlike imperative programming, where a program specifies the control flow to compute a solution, an ASP program is a declarative specification of the problem. The problem is encoded as a set of logical rules, and the solutions correspond to the *stable models* (or answer sets) of the program. ASP serves as the core computational engine for synthesising specifications (Chapters 3 and 4) and as the compilation target for our semi-declarative language (Chapter 5).

Let $\mathcal{A}$ be a set of propositional atoms. A standard ASP logic program consists of rules of the form:

$$a_0 \leftarrow a_1, \ldots, a_m, \mathit{not}\ a_{m+1}, \ldots, \mathit{not}\ a_n. \tag{2.1}$$

where $a_i \in \mathcal{A}$ are atoms, and *not* denotes default negation (negation as failure).

- $a_0$ is the **head** of the rule.
- $a_1, \ldots, \mathit{not}\ a_n$ constitute the **body**.

If $a_0$ is empty, the rule acts as an *integrity constraint* (i.e., the body cannot be satisfied). If the body is empty, the rule is a *fact* (i.e., $a_0$ is unconditionally true). Modern ASP solvers, such as Clingo, extend this basic syntax with rich constructs like *choice rules* $\{a_1, a_2\} \leftarrow \mathit{Body}$, which allow the solver to non-deterministically choose whether to include $a_1$ and $a_2$ in the answer set.

The semantics of ASP is founded on the stable model semantics. Informally, a set of atoms $M \subseteq \mathcal{A}$ is a stable model of a program $\Pi$ if $M$ is the minimal model of the "reduct" of $\Pi$ with respect to $M$ (obtained by removing rules containing negated atoms that are in $M$, and dropping negative literals from the remaining rules). This non-monotonic reasoning capability is particularly powerful for expressing reachability and fixpoints, which are essential for invariant synthesis.

**Scope of ASP in this thesis.** In this thesis, we primarily use the term *Answer Set Programming* (ASP) to refer to ASP as exemplified in Clingo [63], which is widely regarded as a state-of-the-art ASP language and system [24]. Specifically, the encodings developed in later chapters use only the normal-program fragment illustrated above, together with integrity constraints, choice rules, and aggregates. In particular, they do not rely on disjunctive rule heads. Disjunctive logic programming is an important and well-studied extension—closely related to Disjunctive Datalog [54]—and modern ASP systems may include it as part of their full input language. The goal of this primer, however, is not to survey the entire ASP language family, but to fix the core semantic and modelling notions needed in the rest of the thesis.

# Chapter 3

# Sippy: Inductive Synthesis of Heap Predicates

> *Contrary to the rules of philosophers of science, who advise testing hypotheses by trying to refute them, people (and scientists, quite often) seek data that are likely to be compatible with the beliefs they currently hold.*
>
> —Daniel Kahneman, *Thinking, Fast and Slow* [88]

## 3.1 Introduction

This chapter presents Sippy, an automated framework for synthesising inductive Separation Logic predicates directly from positive memory instances. As motivated in § 1.3.1, the central challenge is the *Discovery Problem*: given concrete heap graphs representing valid data structure instances, infer a recursive predicate that captures both the structural shape (*e.g.*, a linked list or binary tree) and the data invariants (*e.g.*, sortedness) of the structure, without requiring any hand-crafted negative examples. Sippy encodes the synthesis search space and domain-specific pruning rules as an Answer Set Programming problem, enabling exhaustive and efficient hypothesis enumeration.

This chapter is based on a published paper by the author [166].

## 3.2 Overview and Key Ideas

In this section, we first introduce Separation Logic—the specification formalism that Sippy targets—then outline the key ideas of our synthesis approach.

### 3.2.1 Background: Separation Logic

Reasoning about programs that manipulate mutable, pointer-based data structures requires handling potential memory aliasing, which standard logics cannot express directly. Separation Logic (SL) extends traditional Hoare logic to address this. A program state in SL is a pair $(s, h)$, where the *pure part* $s$ maps program variables to values and the *spatial part* $h$ maps finitely many memory addresses to values. The key assertions are:

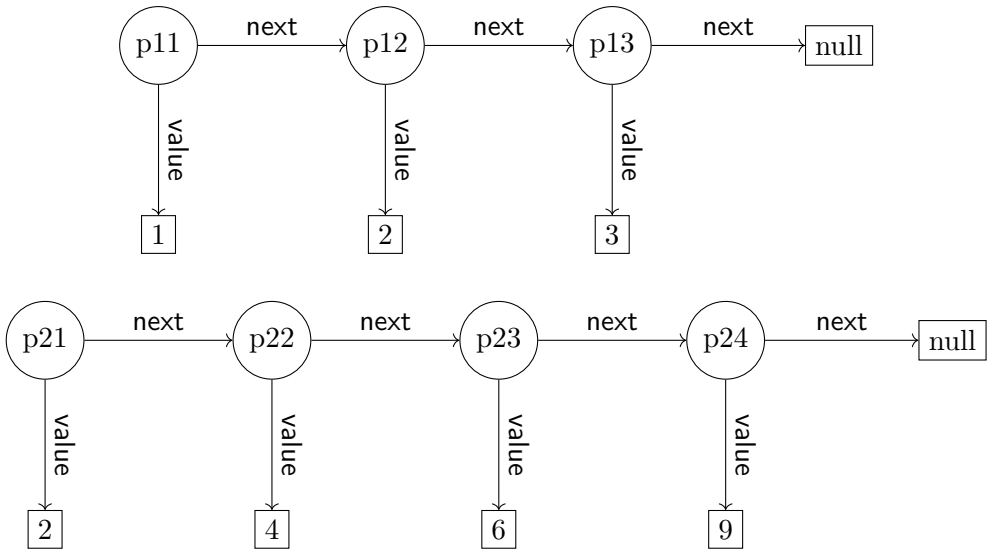
- **Empty heap (emp):** $(s, h) \models \text{emp}$ iff $h$ is empty.
- **Points-to ($x \mapsto v$):** $(s, h) \models x \mapsto v$ iff $h$ contains exactly one location $s(x)$ holding value $v$.
- **Separating Conjunction ($*$):** $(s, h) \models P * Q$ iff $h$ splits into disjoint $h_1 \uplus h_2$ with $(s, h_1) \models P$ and $(s, h_2) \models Q$. The strict disjointness of $\uplus$ prevents aliasing by construction.

To describe recursive data structures of arbitrary size, SL uses *inductive predicates*—the synthesis of which is the primary objective of Sippy.

### 3.2.2 From Memory Graphs to Heap Predicates

Our goal is to synthesise inductive SL predicates from examples of concrete memory graphs. We phrase both SL predicates and the memory graphs that satisfy them in terms of Logic Programming. For example, the Prolog predicate below defines a sorted singly linked list:

```
srtl(X, S) :- empty(S), nullptr(X).
srtl(X, S) :- next(X, Y), value(X, V), srtl(Y, SY),
              min_set(V, S), insert(SY, V, S).
```




```
pos(srtl(p11,[1,2,3])). pos(srtl(p21,[2,4,6,9])).

% Encoding of the first memory graph
next(p11, p12). next(p12, p13). next(p13, null).
value(p11, 1).  value(p12, 2).  value(p13, 3).

% Encoding of the second memory graph
next(p21, p22). next(p22, p23). next(p23, p24). next(p24, null).
value(p21, 2).  value(p22, 4).  value(p23, 6).  value(p24, 9).
```

Figure 3.1: Positive examples of sorted list heap graphs with logic encoding.

The predicate is parameterised by a pointer `X` and a payload set `S`; the constraint `min_set(V, S)` encodes sortedness by stating that `V` is the smallest element of `S`. The predicates `insert` and `empty` correspond to standard set operations.

Figure 3.1 shows two memory graphs of sorted lists and their corresponding logic encoding. The encoding consists of positive examples (*i.e.*, instances of the sought predicate that should hold) and background knowledge (*i.e.*, the memory graph facts). Using program synthesis from input-output pairs as an analogy [75], the background facts (*e.g.*, `next(p11, p12)`) are inputs, the positive examples (*e.g.*, `pos(srtl(p11, [1,2,3]))`) are outputs, and the predicate definition to be synthesised is the program.

An ILP system such as Popper [37] relies on an ASP solver to perform an enumerative search over all possible combinations of literals, finding definitions consistent with the examples.

### 3.2.3 Synthesis without Negative Examples

The classic ILP comes with an important limitation: in general, it requires both *positive* and *negative* examples to learn a predicate. As we explain below, the need for the latter kind of examples makes it challenging to employ ILP *as-is* as a pragmatic approach for synthesising SL predicates.

**Why Negative Examples are Necessary in ILP** Let us get back to our examples with synthesising a sorted singly linked list predicate from positive examples of memory graphs in Figure 3.1. With the conventional ILP, the learned hypothesis by Popper is as follows, and it is not what we need:

```
srtl(X, S) :- empty(S), nullptr(X).
srtl(X, S) :- next(X, Y), value(X, V), insert(SY, V, S),
              srtl(Y, SY).
```

The learned hypothesis does not define a sorted list, but an ordinary (unsorted) singly linked list. The reason is: in the absence of the negative example, this is a consistent hypothesis that is smaller in size than the correct definition of `srtl`. So if we want to learn the correct predicate, we need to provide negative examples that are inconsistent with the incorrect hypothesis, such as

```
neg(srtl(n1, [1,2,3])).
% Encoding of a negative example
next(n1, n2). next(n2, n3). next(n3, null).
value(n1, 2). value(n2, 1). value(n3, 3).
```

which is a singly linked but not sorted list. Thus, when performing synthesis via classic ILP, negative examples are necessary to avoid the predicate being *too general*.

**Challenges in Obtaining Negative Examples.** What makes things worse is that ILP systems rely on *representative* negative examples to correctly prune the generality, which are hard to obtain automatically. The difference between positive and negative examples is that, a good set of positive examples (**Pos**) only needs to guarantee that all instances follow the predicate ($\mathcal{P}$), while a good set of negative examples (**Neg**) needs to be much more elaborated, so it could cover any possible way in which the predicate can be wrong. This difference can be formalised by the following quantifications among the examples:

$$\forall \mathbf{e}^+ \in \mathbf{Pos}, \mathcal{P}(\mathbf{e}^+) \quad \text{vs.} \quad \forall \mathcal{P}' \subset \mathcal{P}, \exists \mathbf{e}^- \in \mathbf{Neg}, \mathcal{P}'(\mathbf{e}^-) \wedge \neg \mathcal{P}(\mathbf{e}^-)$$

That is, unlike a good positive example set, which is only quantified over the examples, a good negative example set is quantified over the *predicates* and the examples, which makes it much harder to achieve. As a concrete example of this phenomenon, imagine learning a predicate for balanced binary trees. A good set of negative examples would contain instances where (a) the height of the left subtree is too large, (b) the height of the right subtree is too large, (c) the imbalance manifests recursively in both left and right subtrees. Without all these rather specific negative instances, it is possible to learn a predicate with a constraint on the subtrees `height(Left) <= height(Right) + 1`, which is not wrong but is imprecise. This is not just an issue for the SL predicate domain, but also for general logical learning, witnessed by the fact that in existing ILP benchmarks [37, 154] and specification mining frameworks [115] high-quality negative examples are often crafted manually.

This state of affairs brings us to the two key novel ideas of this work that enable efficient synthesis of SL predicates only from positive examples.

#### 3.2.3.1 Key Idea 1: Learning with Specificity

As discussed above, without negative examples a solution delivered by Popper, while valid, may not be specific enough. To provide more intuition on the space of possible design choices in finding the best solution, together with the reason why positive-only learning is possible, let us consider the illustration in Figure 3.2. The "up"/"down" in this figure (informally) means

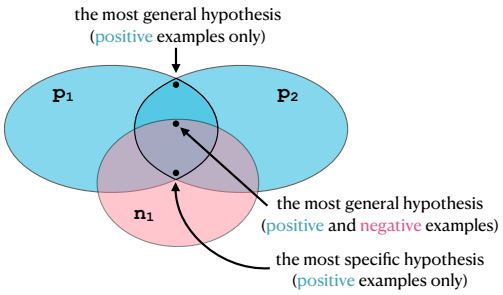



Figure 3.2: The effect of positive and negative examples on search.

"more general/specific", where the top/bottom are constant True/False (*i.e.*, the lattice is defined by subsumption [117]). Providing two positive examples, `p1` and `p2`, restricts the search space for the solution hypothesis to the intersection of their own spaces, with the most general one chosen as the solution. Adding a negative example `n1` provides more restrictions, thus allowing for a more specific most-general solution. From this diagram, one can see that, even without a negative example, we can have a *tighter* [12] solution (*i.e.*, with stronger restrictions given the same number of clauses) if we consider not the most general, but the most specific candidate in the intersection of the search spaces defined by `p1` and `p2` (generally, positive examples only).

Therefore, the basic idea of our positive-only learning is: to learn *the most specific* predicate admitting all provided examples. The only problem is: what is the definition of "specificity"? As the opposite of "generality" (the program with the smallest number of constraints), it is not practical to take the *largest* hypothesis as the most specific, as it would lead to *redundant* constraints. As an example, consider the following valid formulation of the sorted linked list predicate that requires, in its second clause, that `T` $=$ `SY` $\cup\{$`V`$\}$ and `S` $=$ `T` $\cup\{$`V`$\}$:

```
srtl(X, S) :- empty(S), nullptr(X).
srtl(X, S) :- next(X, Y), value(X, V), srtl(Y, SY),
              insert(SY, V, T), insert(T, V, S).
```

Clearly, the last conjunction `insert(T, V, S)` is redundant and can be removed because of the properties of the `insert(...)` predicate.

To eliminate such candidates with redundancies, our approach for positive-only learning encodes intrinsic logical properties of customised predicates to *minimise* the generated SL predicates. Our tool comes with a pre-defined collection of properties of predicates for common arithmetic (*e.g.*, calculation, comparison) and set operations (*e.g.*, insertion, union) that are included into the synthesis automatically. More customised predicates can be added by the user (with additional clause minimisation rules). After performing the minimisation hinted above (detailed in § 3.3.1), we define the *specificity* of a predicate candidate based on its size *w.r.t.* other available candidates (*cf.* § 3.4.3). The solution that is locally-optimal (*i.e.*, the strongest in the search space) will be adopted as the most specific predicate that is implied by all the positive examples.

#### 3.2.3.2 Key Idea 2: Separation Logic-Based Pruning

The domain of our synthesis, *i.e.*, Separation Logic, provides effective ways to prune the search space and accelerate the synthesis process. Postponing the detailed explanation of the optimisations until § 3.4, as an example, consider an important property of the separating conjunction stating that the fact $\mathsf{x} \mapsto \mathsf{a} * \mathsf{y} \mapsto \mathsf{b}$ implies $\mathsf{x} \neq \mathsf{y}$ because of the disjointness assumption enforced by $*$. This property can be encoded as a pruning strategy via ASP integrity constraints that are generated by our tool for each synthesis task. Even for a *doubly linked list*, one of the simplest predicates in our benchmark (*cf.* § 3.6.1), without such optimisations, the synthesis time is increased from 3 to 339 seconds; the synthesis of more complex predicates does not terminate in 20 minutes without SL-specific pruning.

### 3.2.4 Automatically Generating Positive Examples

So far, we assumed that the positive examples are provided by users, in the format of memory graphs. In practice, one may expect that such examples can be obtained in a more automated way, *e.g.*, from the available

programs that manipulate the respective data structures. For instance, an existing work on shape analysis [100] uses a debugger for extracting memory graphs from a program's execution, with the assumption that (1) the user indicates the line of code to extract the memory graph, and (2) an input for the program is provided. Another related line is shape-neutral program analysis, which checks integrity properties of graph-based data structures using constraint solving without committing to a particular graph shape such as lists or trees [50]. This is complementary to Sippy: such tools analyse a program against target data-structure properties, whereas Sippy learns an explicit recursive SL predicate from positive heap instances. Unfortunately, this rules out a large set of programs that *expect* a data structure rather than generate one: without a suitable input we simply cannot run them to obtain a graph, and constructing such input is not much easier than encoding a memory graph manually.

To address this issue, we implemented Grippy—a tool that can automatically generate positive examples in the form of arbitrary valid memory graphs of a data structure from the program that manipulates the structure, without requiring the user to provide any input. The only assumption we make is that the program in question is instrumented with test assertions, which can be used to filter out the invalid memory graphs from the randomly generated ones. We further show in § 3.6.2.1 that Grippy can effectively generate input graphs for synthesising SL predicates from real heap-manipulating programs, reducing the specification burden for proving their correctness.

### 3.2.5 Putting It All Together

We implemented our algorithm for inductive synthesis of SL predicates as the core of our tool called Sippy. Figure 3.3 shows the high-level workflow of Sippy. Starting from the left, the user can provide positive examples (*i.e.*, structure-specific heap graphs) for the synthesiser by either using our graph generator Grippy on programs that expect the data structure instance (*cf.* § 3.5), or by manually writing them (*e.g.*, in the style of Story-

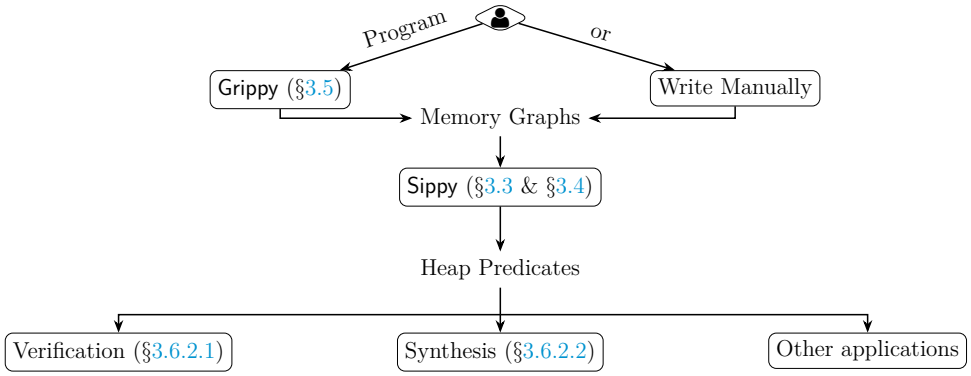



Figure 3.3: The Workflow of Sippy

Board Programming [149]). Given the graph examples, Sippy synthesises an inductive predicate definition, which can be further used for program verification in SL-based verifiers (§ 3.6.2.1), for program synthesis (§ 3.6.2.2), or by any other SL-based tool.

## 3.3 Positive-Only Predicate Learning

In this section, we describe our approach for learning predicates from positive-only examples not specific to SL predicates. Positive-only data has a long history in inductive inference, including classic results on formal-language learning from positive data [10]; our setting is different, but it shares the same basic tension that positive examples alone do not rule out over-general hypotheses. To make the learning effective, we introduce the idea of predicate clause *minimisation*, then explain how our specificity-based positive-only learning works by showing its difference from standard ILP [37, 35]. We note upfront that in general-purpose ILP, one cannot simply dispense with negative examples, as an unconstrained hypothesis space provides no canonical way to rule out over-general programs; rather, our positive-only learning (and our notion of "most specific" hypothesis) should be understood as a domain-motivated heuristic designed for specification and heap predicate synthesis, where examples represent concrete valid

data structures and we bias the search toward the tightest non-redundant specification within a bounded grammar.

### 3.3.1 Normalising the Positive-Only Learning

As discussed in § 3.2.3.1, without negative examples, adopting the smallest-size hypothesis, as done in many program synthesisers [87, 102], can lead to finding solutions that are too imprecise. What about selecting the candidate with the largest size instead? As shown by the sorted list (`srtl(X, S)`) example, more literals in the body do not mean the predicate is more specific if the body contains *redundant* constraints among variables. Therefore, we first define a normalisation procedure to remove the redundancy in the predicate clauses with the help of the logical *entailment* (denoted as $\models$ for Prolog clauses known as *definite clauses* in FOL).

**Definition 1** (Clause Minimisation)**.** Let $\mathbf{numlit}(h)$ denote the number of literals in the body of a clause $h$, and $A \iff B$ denote $A \models B$ and $B \models A$. A predicate clause $h$ is eliminated *iff* $\exists h_0,\ h_0 = \arg\min_{X \in S} \mathbf{numlit}(X) \wedge \mathbf{numlit}(h_0) < \mathbf{numlit}(h)$, where $S = \{X \mid X \iff h\}$.

That is, if for any clause $h$ there is a shorter clause $h_0$ equivalent (defined by entailments) to $h$, $h$ is eliminated in the synthesis. Such minimisation is naturally encoded by ASP. Our experiments show that enumerable (also decidable) entailment rules are effective for the synthesis of SL predicates within finite clause length (§ 3.6.1). Note that the minimisation is essentially a pruning for unnecessary clauses: we do not handle predicate-level redundancy (*a.k.a.* Horn literal minimisation [159]) as it is NP-hard in general [77].

### 3.3.2 An Algorithm for Positive-Only Learning

Our approach to learning predicates from positive-only examples can easily make use of components from the existing ASP-based ILP systems [37, 35], seamlessly. In this section, we describe both the basic ILP learning loop by ASP and the procedure of our positive-only learning.

**Algorithm 1** The positive-only learning (POL) vs. ASP-based ILP

```
Require: Search space: initial constraints in_cons, size limit max_size
Require: User inputs: background knowledge bk, pos/neg examples exs
 1: procedure POL OR ILP (bk, exs, in_cons, max_size)
 2:     cons = in_cons, size = 1, sol = True
 3:     while size≠ max_size do
 4:         h = CLINGO_GEN(cons, size)                          ▷ generate
 5:         if h is None then                                   ▷ no answer set (UNSAT)
 6:             size += 1
 7:         else
 8:             outcome = PROLOG_TEST(h, exs, bk)               ▷ test
 9:             if outcome is complete and consistent then
10:                 sol = h
11:                 break
12:             end if
13:             cons += pruning(h, outcome)                     ▷ prune
14:             if outcome is complete and sol ⊭ h then
15:                 sol = update_sol(h, comparable(h, sol))
16:             end if
17:             cons += new_pruning(h, outcome)                 ▷ prune (updated)
18:         end if
19:     end while
20:     return sol
21: end procedure
```

Algorithm 1 (with the light grey colour) shows the pseudocode of classic ILP learning loop, which follows the "generate-test-constrain" approach. It takes (1) background knowledge `bk`, (2) positive and negative examples `exs`, and (3) learning bias defining parameters of the search `max_size`, such as the maximum size of a predicate to be searched and other customisable parameters (*e.g.*, whether to enable mutually-recursive definitions), summarised as `in_cons`. After encoding the iterative deepening search problem into an answer set program, it first uses the ASP solver Clingo [65] (CLINGO_GEN) to generate a candidate hypothesis from a meta-level encoding of the syntax space (line 5; returning **None** on line 6 when the ASP instance is unsatisfiable and no unpruned candidate of the current `size` remains), and then uses Prolog (PROLOG_TEST) to test the candidate against the provided examples (line 9). Note the distinct roles of Clingo and Prolog in this architecture: Clingo does not execute the object-level predicate on memory graphs, but searches a meta-level ASP program whose

answer sets encode which literals form a candidate clause `h`. Testing `h` is delegated to Prolog because candidate predicates involve recursive definitions paired with rich pure theories (*e.g.*, list/set constructors such as `insert` and `min_set`) as well as dynamic heap-linearity checks (§ 3.4.2); evaluating these in a goal-directed, top-down manner in Prolog avoids the severe grounding blowup (or non-termination over structured set/list terms) that would arise if tested bottom-up in ASP. A hypothesis is *complete* if it entails all positive examples, and *consistent* if it entails no negative examples. ILP returns a hypothesis as the solution if it is complete and consistent (lines 10–12); otherwise, it *prunes* the search space using the test outcome (line 14) and continues the search. The test-based pruning works as follows. Whenever hypothesis testing shows that not all positive examples are true (incomplete), the pruning on *specialisation* is applied; whenever part of the negative examples are true (inconsistent), the pruning on *generalisation* is applied—ideas of the pruning are illustrated in § 3.2.2 and are detailed in [37].

For the case of *positive-only* learning (*cf.* Algorithm 1 with the dark grey colour), we follow the approach from the work on *specification synthesis* [126] to output a set of non-comparable (*i.e.*, the element in the set is not entailed by any other) solutions as a conjunction: there are possibly several solutions satisfying the positive examples where none of them is more specific than the others, so the conjunction of them is the most specific specification. The main difference from classic ILP starts from line 15, when a hypothesis `h` is complete and is not entailed by the existing solution set `sol`, the solution set is updated by adding `h` to the set, together with removing all solutions in `sol` that are comparable (semantically more general in our case) to `h` (line 16). The set of most specific solutions in the search space is returned when the search is exhausted (line 21). The advanced pruning procedure (line 18) is enabled by domain knowledge (*i.e.*, Separation Logic predicates in our case—*cf.* § 3.4.2). We conclude this section by proving the soundness of positive-only learning.

**Theorem 1.** *The hypothesis set returned by the positive-only learning in* Algorithm 1 *contains all non-comparable, most specific predicates that*

*are complete (*i.e.*, entailing all examples) in the search space defined by the algorithm's initial constraints (*`in_cons`*) and the size limit (*`max_size`*) parameters.*

*Proof.* We prove the theorem by the loop invariant maintained by Algorithm 1: after any finite prefix of the search, `sol` contains exactly the non-comparable, most specific complete hypotheses among the hypotheses that have already been generated and tested. Initially no hypothesis has been tested, so the invariant holds trivially. Consider the next generated hypothesis `h`. If `h` is incomplete, it cannot be a member of the returned set, and pruning only removes hypotheses that are specialisations of an already incomplete hypothesis, so no complete most-specific hypothesis is lost. If `h` is complete but is entailed by some existing member of `sol`, then `h` is more general and cannot be most specific. If `h` is complete and not entailed by `sol`, the algorithm inserts `h` and removes precisely those old members that are comparable with it and more general than it. The remaining set is again exactly the non-comparable frontier of the most specific complete hypotheses in the explored space.

The loop terminates only when the ASP generator has exhausted the finite search space defined by `in_cons` and `max_size`; therefore the explored space is the whole search space. By the invariant, the returned `sol` contains all non-comparable, most specific complete predicates in that space. □

## 3.4 Separation Logic Predicate Synthesis via Sippy

Having described the enhanced *general-purpose* predicate synthesis algorithm from positive-only examples, we now show how to instantiate it for synthesis of inductive SL predicates and improve the efficiency of the search algorithm by exploiting domain-specific SL insights. We further discuss the SL-validity of the synthesised predicates and the completeness of the search algorithm.

$$
\begin{aligned}
\langle \text{predicate} \rangle &::= \langle \text{main_pred} \rangle \mid \langle \text{main_pred} \rangle \langle \text{invented_pred} \rangle * \\
\langle \text{main_pred} \rangle &::= \langle \text{base_clause} \rangle (\lceil \text{main_head} \rfloor) \mid \langle \text{rec_clause} \rangle (\lceil \text{main_head} \rfloor) * \\
\langle \text{invented_pred} \rangle &::= \langle \text{base_clause} \rangle (\lceil \text{inv_head} \rfloor) \mid \langle \text{rec_clause} \rangle (\lceil \text{inv_head} \rfloor) * \\
\langle \text{base_clause} \rangle (H) &::= H(\texttt{This}, \langle \text{args} \rangle) \leftarrow \langle \text{base_lit} \rangle *, \langle \text{pure_lit} \rangle * \\
\langle \text{rec_clause} \rangle (H) &::= H(\texttt{This}, \langle \text{args} \rangle) \leftarrow \langle \text{pointer_lit} \rangle *, \langle \text{rec_lit} \rangle *, \langle \text{pure_lit} \rangle * \\
\langle \text{literal} \rangle (R) &::= R(\langle \text{args} \rangle) \\
\langle \text{base_lit} \rangle &::= \langle \text{literal} \rangle (\lceil \text{base_pred} \rfloor) \quad \texttt{\% Pre-defined for spatial relations} \\
\langle \text{pure_lit} \rangle &::= \langle \text{literal} \rangle (\lceil \text{pure_pred} \rfloor) \quad \texttt{\% Pre-defined for pure relations} \\
\langle \text{pointer_lit} \rangle &::= \lceil \text{domain} \rfloor (\texttt{This}, \langle \text{var} \rangle) \quad \texttt{\% Extract from the memory graphs} \\
\langle \text{rec_lit} \rangle &::= \langle \text{literal} \rangle (\langle \text{head} \rangle) \\
\langle \text{args} \rangle &::= \langle \text{var} \rangle \mid \langle \text{var} \rangle, \langle \text{args} \rangle \\
\langle \text{var} \rangle &::= \texttt{X1} \mid \texttt{X2} \mid \ldots \mid \texttt{This} \\
\langle \text{head} \rangle &::= \lceil \text{main_head} \rfloor \mid \lceil \text{inv_head} \rfloor \quad \texttt{\% From task or randomly generated}
\end{aligned}
$$

Figure 3.4: The grammar of the SL predicates, in basic Backus–Naur form (BNF), extended with (1) meta-variables $(\cdot)$ for specialising the symbols, and (2) pre-defined atoms denoted by $\lceil \text{X} \rfloor$ (with comments of their origins).

### 3.4.1 SL Predicates: Basics and Intricacies

We define the space of SL predicates in a way standard for Syntax-Guided Synthesis (SyGuS) [6]. The grammar of the SL predicates is shown in Figure 3.4. An SL predicate is either having a shape with a single main predicate, or shaped by a main predicate together with a set of invented *auxiliary* predicates, which are required in the case of nested linked structures.

Specific to the predicates, both main predicate and invented predicates consist of the base and recursive clauses, where the base clause is the one that does not have any recursive calls, and the recursive clause is the one that has recursive calls. The head literal (*i.e.*, before $\leftarrow$) in each clause has a fixed argument `This` that denotes the base address of the data structure (similar to the *this* reference in object-oriented programming). The body literals (*i.e.*, after $\leftarrow$) in the clauses are defined in terms of different predicates: the base (and pure) predicates are pre-defined, but extensible, to capture the spatial relations among the pointers for the base clause (the pure constraints among variables in clauses, respectively); the domain predicates describing the

points-to relations can be obtained from the memory graphs; the recursive predicates are the recursive calls to the main or invented predicates.

Three aspects in the grammar in Figure 3.4 contribute to the infinite synthesis search space: (1) the length of clauses, (2) the number of sub-clauses for each predicate, (3) the arity of the invented predicates. For our task, we noticed that predicates for real-world structures rarely require more than 10 literals in their bodies; two sub-clauses for each predicate are sufficient to capture the common structures; and the arity of the invented predicates is set to be not more than the arity of the main predicates. Such bounds for hypothesis space are common in almost all synthesis-by-example tools (*e.g.*, [37, 102, 145]), not only to make the synthesis tractable, but also to avoid overfitting [124] (*e.g.*, a predicate disjointing facts of all examples).

Below, we discuss two challenges in making SL predicate synthesis effective and efficient, together with how we address them in Sippy.

#### 3.4.1.1 Semantic-Based Pruning

In most existing syntax-guided synthesisers [37, 6, 145], the search is accelerated by pruning of the hypothesis search space by employing the general *syntax* restrictions. Other than limiting the syntax, we apply the following *semantic* properties' restriction of Separation Logic predicates to the search.

1. *Basic reachability*: no points-to relation appears in the body other than the ones from the `this` pointer. Thus, the clause `p(X, Y) :- next(X, Y), next(Y, Z), ...` is not allowed as a candidate, because we expect all locations in the body to be accessible from `this` via fields.

2. *Basic assumptions*: the base (non-recursive) clause restricts `this` pointer to either be **`null`** or to equal to another pointer parameter variable. *E.g.*, `p(X, Y) :- nullptr(X), ...` is allowed, but `p(X, Y) :- next(X, Y), ...` cannot be a base clause.

3. *Restricted use of* **`null`**: if a variable `X` is a null-pointer (denoted by `nullptr(X)`), no more `X` occurs in the clause. *E.g.*, `p(X, Y) :- nullptr(X), next(X, Y)` is not a valid clause.

4. *Quasi-well-founded recursion of payload*: the pure argument passed to a recursive call should (non-strictly) decrease. For example, in a clause like `p(X, S) :- next(X, Z), p(Z, S1), ...`, the set `S` should contain `S1`. This is a common assumption in recursive program synthesis [5, 102]. Syntactically, because Clingo generates clause structures without evaluating the sets themselves, this property is enforced via pre-declared argument-order annotations on the pure background predicates (*e.g.*, `insert(S1, V, S)` and `union(S1, S2, S)` are annotated as making `S1` a sub-collection of `S`): an ASP integrity constraint requires that in any recursive clause with head pure argument `S` and recursive-call pure argument `S1`, either `S1` is the same variable as `S` or the clause body contains an annotated pure literal that syntactically witnesses `S1` as smaller than `S`.

5. *Heap functionality*: points-to relations of the same field should not target different locations. *E.g.*, a clause like `p(X, Y) :- next(X, Z), next(X, Z1), ...` cannot be a candidate.

This list of search constraints represents a combination of the properties implied by SL semantics (in a Java-style field-based memory model) as well as by common properties of data structures, which are essential for pruning the search.

#### 3.4.1.2 Free Variables and Auxiliary Placeholders

Free variables are common in SL predicates, *e.g.*, the (implicitly existentially-quantified) location `Y` in the base clause of the doubly linked list below:

```
dll(X, Y) :- nullptr(X).
dll(X, Y) :- next(X, Z), prev(X, Y), dll(Z, X).
```

Unfortunately, completeness guarantees of pruning discussed in § 3.3.2 do not hold for predicates with free variables in the sense that a complete (*i.e.*,

valid) hypothesis with free variables might be wrongfully pruned during the search [37, §4.5]. To address this problem, we introduce *auxiliary placeholders* into the search as a way to express predicate clauses with free variables. For example, the following doubly linked list predicate can be regarded the same as the one above with `anypointer()` placeholder, and *can* be synthesised.

```
dll(X, Y) :- nullptr(X), anypointer(Y).
dll(X, Y) :- next(X, Z), prev(X, Y), dll(Z, X).
```

On a technical level, this requires adding an ASP constraint that forces the parameter of the placeholder predicate (`Y` here) to appear *twice* in the whole clause, so it could be later translated into a single occurrence of a free variable.

### 3.4.2 Ensuring SL Validity in Prolog

An astute reader can notice that the validity of the synthesised predicates is not immediate due to our treatment of Prolog clauses as SL assertions: in Separation Logic, the *separating conjunction* ($*$) is a semantic constraint requiring disjointness of sub-heaps, which is not expressed directly in the syntax of Prolog clauses (whose comma operator denotes standard classical conjunction) and therefore is not guaranteed by ordinary Prolog resolution alone. As an example, consider the following simplified Prolog predicate for binary trees:

```
bi_tree(X) :- nullptr(X).
bi_tree(X) :- t1(X, L), t2(X, R), bi_tree(L), bi_tree(R).
```

In this case, an instance of `bi_tree(X)` being evaluated to be *true* in Prolog can imply *false* under SL semantics that enforces heap disjointness: considering a memory graph with two nodes

```
t1(n1, n2). t2(n1, n2). t1(n2, null). t2(n2, null).
```

so that the graph fact `bi_tree(n1)` is provable in Prolog, but the clauses `bi_tree(L)` and `bi_tree(R)` are *non-disjoint*. Notice that, in our induc-

tive synthesis setting, this situation would correspond to having *multiple* occurrences of the same points-to fact in a memory graph representing a positive example for the predicate, but should not be allowed by the definition of separating conjunction.

To avoid this source of unsoundness, we use a straightforward solution that enforces this semantic constraint dynamically via a *linearity execution check* in Prolog rather than encoding separation in the clause syntax: a valid SL predicate is a complete Prolog predicate where the positive examples succeed using each points-to fact *exactly* one time (a semantic property of SL assertions known as *linearity*). This check is not delegated to ordinary Prolog unification alone. During hypothesis testing, the generated predicate is instrumented so that each recursive call returns the set of spatial heap facts it consumes; the tester then rejects a proof if two recursive branches use overlapping points-to facts, or if the accepted proof leaves a required points-to fact unused. In this way, Prolog still performs the logical search, while the extra linearity check enforces the disjointness of the heaplets encoded by the memory graph. For the complete Prolog but invalid SL predicates, we also use the *specialisation* rule in § 3.3.2 to prune them: if a predicate violates the linearity, then a more constrained one will also violate it; this contributes to the new pruning in line 18 of Algorithm 1.

In summary, for the predicates in Sippy's search space (§ 3.4.1), this linearity check bridges Prolog evaluation and Separation Logic semantics: letting `F(h)` denote the memory graph of a heap `h`, for any output predicate `p(X)` of Sippy and any heap `h`, if `F(h)` is provable in Prolog under the linearity check ($\texttt{F(h)} \models_{\text{Prolog+Lin}} \texttt{p(X)}$), then the corresponding heap is valid under the semantics of Separation Logic ($\texttt{h} \models_{\text{SL}} \texttt{p(X)}$).

### 3.4.3 The Sippy Algorithm

The only remaining step before putting all the pieces together is to select the desired predicate from the set of non-comparable solutions of positive-only learning. Even though predicates from POL can be conjuncted in general, the semantics of SL predicates following the definition in § 3.4.1 is more

**Algorithm 2** The Sippy loop for inductive predicate synthesis

```
Require: memory graphs consist of graph_bk, exs
 1: procedure SIPPY(graph_bk, exs)
 2:     graph_cons, shapes = graph_info(graph_bk, exs)
 3:     max_var = max_body = 1
 4:     sol = True                    ▷ The most general solution as initialisation
 5:     for shape in shapes do
 6:         max_size = maxsize(max_body, shape)
 7:         h = MAX_POL(graph_bk, exs, graph_cons, max_size)
 8:         if h ≺ sol then           ▷ A more specific predicate is obtained
 9:             max_var, max_body = (var_of(h), size_of(h)) + δ
10:             sol = h
11:         else if sol == True then  ▷ No predicate is yet learned
12:             if max_var == upper_bound then
13:                 continue                  ▷ Try the next shape
14:             end if
15:             max_var, max_body += (1, 1)
16:         else
17:             break                 ▷ No more specific predicate is found
18:         end if
19:     end for
20:     return sol
21: end procedure
```

restrictive and the conjunction of valid SL predicates may result in an ill-formed or a constantly false one. We found in practice that after the semantics-based normalisation from § 3.3.1, the number of literals can serve as a *good enough* specificity metric among incomparable predicates, since containing more literals is likely to contain more information or constraints about the heap structure. Following this intuition, we define the synthesis algorithm with MAX_POL function, which obtains the largest predicate from POL as per Algorithm 1.

Algorithm 2 summarises the internal workings of Sippy. Our synthesiser takes as inputs memory graphs encoded as sets of logic facts (*e.g.*, `graph_bk`, such as `next(..)` and `value(..)` from Figure 3.1), positive examples of heaps on which a predicate holds (*e.g.*, `exs` as `srtl(..)` from Figure 3.1), so that the shape (matched with pre-defined shapes in § 3.4.1) and a set of ASP constraints (`graph_cons`) describing the information in the graphs (such as the arity and types of the predicates to be learned) are obtained (line 2). Two parameters (line 3) for positive-only learning (MAX_POL),

(1) the maximum number of variables and (2) the maximum size of the body of a predicate clause for restricting the search space, are gradually increased during the search using the following empirical strategy: if no solution is valid (line 12), we either increase both parameters by one to enlarge the space until finding one (line 16), or the attempt on the current predicate shape fails (*i.e.*, the upper bound of the search space is reached), then Sippy will try synthesising using the next shape (line 14, *i.e.*, more auxiliary predicates); when obtaining one new better predicate than the existing, the search parameters are both increased by a parameter $\delta$ to find a possibly more specific predicate (line 9), and the solution is updated (line 10); if the learned predicate in the larger search space is not better than the previous, we stop the search and output (lines 17–18). The role of the parameter $\delta$ is, thus, to provide a "margin" for the completeness of the search: it is not guaranteed that Sippy will find the most specific solution *across all possible search spaces*, but only in the search-space that is bound by the returned output's parameters *plus* $\delta$.[1] Note that line 6 of Algorithm 2 features a function `maxsize()` that calculates the maximum size of the search space based on the maximum body length (`max_body`) and the predicate shape setting.

Finally, we provide a correctness argument for Sippy. The soundness of synthesising *consistent* (*i.e.*, inhabited) and *well-formed* (*i.e.*, finitely provable) SL predicates is guaranteed by the soundness of classic ILP and the linearity check in § 3.4.2. The following 'local" completeness states that, given the output of Sippy, no more specific output can be discovered, *even in* the larger search space obtained by increasing the search parameters *once* by $\delta$ at the line 9 of Algorithm 2. Here the pair $(m, n)$ denotes the two bounds associated with the returned predicate—respectively, the number of variables used by the predicate and the maximum length of a clause body—while $(m', n') = (m, n) + \delta$ denotes the enlarged bounds used by the final unsuccessful search before the algorithm terminates.

[1]We choose it to be (1,2) in our experiment from the natural observation: for our domain, we expect to have one body literal where the predicate is generating a new variable, and one more body literal that uses the new variable.

**Theorem 2** (Local Completeness of Sippy)**.** *If the output of Sippy is a predicate with the maximum number of variables $m$ and the maximum length of the body $n$, then there is no predicate with the maximum number of variables $m'$ and the maximum length of the body $n'$, where $(m',\ n') - (m,\ n) = \delta$, that is more specific than the output predicate.*

*Proof.* Let `sol` be the predicate returned by Sippy, and suppose that `sol` has $m$ variables and maximum body length $n$. When `sol` was first obtained, line 9 of Algorithm 2 set the next search bounds to $(m', n') = (m, n) + \delta$. The algorithm returns `sol` only after running `MAX_POL` under these enlarged bounds and failing to obtain a hypothesis `h` such that $h \prec sol$; otherwise line 8 would hold, line 10 would update `sol`, and the algorithm would continue rather than terminate.

By Theorem 1, the result of `MAX_POL` for the enlarged search space contains the non-comparable most-specific complete predicates in that space. Therefore, if there existed any complete predicate `p` within the bounds $(m', n')$ that was more specific than `sol`, `MAX_POL` would return either `p` itself or another non-comparable frontier predicate at least as specific as `p`. In either case, Sippy would obtain a predicate more specific than `sol`, contradicting the fact that the final enlarged search caused termination. Hence no predicate within the one-$\delta$ larger search space is more specific than the output. □

## 3.5 Automated Heap Graph Generation via Grippy

As presented, Sippy requires input graphs to be provided explicitly. The final technical contribution of this work is Grippy—an auxiliary tool that allows one to *automatically generate valid structure graphs* via programs that take them as inputs and are most likely already available to the user.

Valid memory graphs can be obtained by extracting concrete heap states from the program execution, provided concrete inputs, as, *e.g.*, done by the SLING tool [100]. This approach is, however, only applicable if the

program in question *generates* a data structure instance, and is problematic if it *expects* it as an input. Although symbolic execution could in principle explore execution paths of a structure-expecting program to solve for valid input heaps, symbolic execution over unbounded pointer structures and complex payload theories (*e.g.*, sets and arithmetic) suffers from severe path explosion when the inductive shape predicate is not yet known. Our idea for automated graph generating is inspired by works on fuzz-testing and uses a structure-expecting program as a *validator* for candidate graphs. Given a program with assertions, Grippy generates *arbitrary* input memory graphs, subject to basic validity constraints (*e.g.*, any node should be reachable from its root), and keeps the graphs that pass the assertions together with example facts summarised from those graphs by traversing them and accumulating their payload in a set, so they can be used as inputs for Sippy. The details are given in Algorithm 3.

One can argue that the generator defined this way can also effectively produce *negative* examples, thus removing the need for our positive-only learning. In practice, however, the number of the negative examples can be large due to the randomness of the generator that has no knowledge of the data structure (*cf.* § 3.6.2.1). This makes it impractical to use them for ILP-based synthesis that cannot effectively discriminate good (*i.e.*, informative) negative examples from arbitrary junk.

## 3.6 Evaluation and Discussion

We implemented Sippy by extending Popper with the combined use of Python (∼200 lines for modifying Popper), Prolog (14 rules for specificity of predicates, and 19 supported predicates for pure relations in first-order theories), and ASP (∼200 lines for SL domain knowledge, and ∼300 lines for Sippy's search space, where 46 rules are used to encode the minimisation rules of the 19 pure relations). The prototype of Grippy is implemented by ∼50 lines of ASP (to generate the graphs) together with ∼100 lines of Python. All experiments were conducted on an 8-core M1 MacBook

**Algorithm 3** Generating random memory graphs and examples

**Require:** $p$: Program with assertions as tests
**Require:** $n$: Number of graphs to be generated
Initialize $GraphsAndExamples \leftarrow \emptyset$
**for** $sz \leftarrow 1$ to $max_size$ **do**
    $cnt = 0$
    **while** $cnt < \lceil n/max_size \rceil$ **do**
        $node_1, \ldots, node_{sz} \leftarrow init_node()$
        $node_i.key \leftarrow$ random(int) for $i \in [1, sz]$
        $node_i.pointer \leftarrow$ random(node) for $i \in [1, sz]$
        **if** $p(node_1, \ldots, node_{sz})$ satisfies all assertions **then**
            cnt $\leftarrow$ cnt + 1
            $example \leftarrow$ summarise_graph$(node_1, \ldots, node_{sz})$
            $GraphsAndExamples$.add(graph$(node_1, \ldots, node_{sz})$, $example$)
        **end if**
    **end while**
**end for**
**return** $GraphsAndExamples$

Pro with 16GB RAM. The valid structure-specific input memory graphs were either manually/LLM-written, or automatically generated via Grippy as described in § 3.5.

### 3.6.1 Benchmarking Predicate Synthesis

To assess Sippy's efficacy as a tool for heap predicate synthesis, our evaluation addresses the following research questions:

**RQ 1.1** *How effective is Sippy in synthesising heap predicates?*

**RQ 1.2** *What factors affect the synthesis efficiency?*

**RQ 1.3** *How scalable is Sippy with respect to the input size?*

#### RQ 1.1: Effectiveness and Expressiveness

We have assembled a set of benchmarks for common and complex heap-based data structures. Table 3.1 summarises our (mostly)[2] successful case studies: 19 Separation Logic predicates synthesised by Sippy within the

[2]We will elaborate on the partially-successful binomial heap instance #15 in § 3.6.3.

imposed 20 min time limit, which shows that Sippy can effectively produce SL predicates for a variety of heap-based data structures.

In our evaluation, we restricted the predicates in the search space to feature at *most one pure* theory, since for most predicates, the pure relations on variables do not influence each other's validity. For example, the *AVL tree* predicate should feature (1) a relation between the heights of a node's subtrees and (2) ordering relations on the node payload; any one of these can be encoded independently, leading to separate SL predicate definitions: `balanced(A, B)` for *balanced tree* and `bst(A, B)` for *binary search tree*. One can then *merge* those two SL predicates with identical spatial components into a united *AVL tree* `avl(A, B, C)` as follows:

1. Merge the parameter lists by appending the pure variables;
2. Merge pure relations with the overlapped variable renaming;
3. Adapt the recursion in the spatial parts for the new parameter.

To show that our positive-only learning (in § 3.3) and the SL-based optimisations (in § 3.4.1.1) are effective, we compared the synthesis time of Sippy with the unoptimised classic ILP system Popper, Popper with the SL-based optimisations, and Sippy without the optimisations (from left to right in the table). The results show that (1) our positive-only learning effectively learns the predicates which classic ILP cannot discover, and (2) our optimisations help to reduce the synthesis time significantly for both classic ILP and POL for most cases. The NA entries mark expressiveness limitations rather than failed runs: classic ILP requires negative examples, so it is not applicable when a meaningful negative set is unavailable, while ShaPE cannot express payload constraints or some predicate-shape features used in these benchmarks. They are therefore separated from TO, ERR, and WA, which refer to attempted runs. The only exception, #2, where the unoptimised Sippy is faster, is due to the small size of the predicate, where the cost of adding constraints for reducing the search space is larger than the benefit of the pruning.

We attempted to compare synthesis capabilities of Sippy to those of ShaPE [21], the most closely related work. Like Sippy, ShaPE synthesises SL

Table 3.1: Statistics on synthesised predicates and the comparison with other tools/settings. Columns following the predicate names are split by (1) the properties of the predicates: whether a predicate is invented (*i.e.*, nested data structure, PI), used pure theories Th, the size of the output predicate Size (a triple of arity, the maximal number of literals, the maximal number of variables in the predicate); (2) the stats of Sippy's synthesis: the percentage of run time taken by testing the hypotheses Test%, the synthesis time of finding the expected hypothesis $T_0$, the synthesis time of exhaustive search in **P**ositive-only learning $T_P$, (3) the time $T_I$ it took to synthesise a result when using Popper as the classic **I**LP to synthesise, the time $T_{Io}$ when using classic **I**LP together with our **o**ptimisation to synthesise, and the time $T_{Pd}$ of running **P**OL with SL-based optimisations **d**isabled; finally, (4) we report the runtimes $T_S$ of ShaPE on the same tasks (but without pure relations). All times are in seconds, with TO for time-out, ERR for crashes, WA for wrong answers, and NA for unsupported cases.

| No. | Predicate | PI | Th | Size | Test% | $T_0$ | $T_P$ | $T_I$ | $T_{Io}$ | $T_{Pd}$ | $T_S$ |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | `singly linked list (payload)` | no | set | (2,8,5) | 2% | <1 | 1 | 4 | <1 | 3 | <1 |
| 2 | `singly linked list (length)` | no | int | (2,9,5) | 1% | 1 | 9 | NA | NA | 4 | |
| 3 | `singly linked list segment` | no | set | (3,8,6) | 5% | <1 | 1 | TO | 1 | 31 | 1 |
| 4 | `doubly linked list` | no | set | (3,10,6) | 1% | 2 | 3 | NA | NA | 339 | 1 |
| 5 | `doubly linked list segment` | no | set | (5,10,8) | 1% | 49 | 192 | TO | 10 | TO | TO |
| 6 | `sorted singly linked list` | no | set | (2,9,5) | 3% | <1 | 1 | NA | NA | 4 | NA |
| 7 | `sorted doubly linked list` | no | set | (3,11,6) | 2% | 1 | 3 | NA | NA | 231 | NA |
| 8 | `circular list` | no | set | (3,8,6) | 23% | <1 | 1 | 18 | <1 | 81 | <1 |
| 9 | `lasso list` | no | set | (3,8,6) | 25% | <1 | 1 | TO | 1 | TO | NA |
| 10 | `binary tree` | no | set | (2,11,8) | 8% | 2 | 13 | TO | 2 | 245 | WA |
| 11 | `back-linked tree` | no | set | (3,13,9) | 9% | 33 | 176 | TO | 41 | TO | TO |
| 12 | `binary search tree (set)` | no | set | (2,13,8) | 15% | 11 | 26 | NA | NA | TO | NA |
| 13 | `binary search tree (list)` | no | list | (2,13,8) | 39% | 33 | 433 | NA | NA | TO | |
| 14 | `balanced tree` | no | int | (2,12,7) | 32% | 34 | 458 | NA | NA | TO | NA |
| 15 | `binomial heap (order)` | no | int | (3,14,8) | 33% | 37 | 1087 | NA | NA | TO | NA |
| 16 | `binomial heap (payload)` | no | set | (3,14,9) | 2% | 59 | 101 | NA | NA | TO | NA |
| 17 | `list of lists (set)` | yes | set | (2,17,5) | 1% | 9 | 321 | TO | 33 | TO | ERR |
| 18 | `list of lists (list)` | yes | list | (2,17,5) | 1% | 14 | 759 | TO | 47 | TO | |
| 19 | `rose (n-ary) tree` | yes | set | (2,17,6) | 1% | 9 | 749 | NA | NA | TO | ERR |

predicates for data structures from memory graphs, allowing for positive-only synthesis without negative examples via meta-interpretive learning (MIL) [118]. However, unlike Sippy, ShaPE only allows one to synthesise a structure's shape constraints, without any restriction on the data payload or other pure constraints—this is why three data structures (SLL, BST, and a list of lists) with different pure relations are joined in Table 3.1 into one predicate per structure in the ShaPE column $\mathbf{T_S}$. It has been shown in the prior work that the MIL-based learning cannot define a complete search space for general logic programs [38], which indicates that ShaPE *cannot be extended*, even in principle, to express arbitrary data constraints, such as arithmetic ones. That is, theoretically, 9 (those not marked as NA) out of

16 predicates (with the joining taken into the account) in Table 3.1 can be synthesised by ShaPE. However, in practice, the bugs in ShaPE's learning loop implementation resulted in either (1) the search timing out (TO), (2) ShaPE terminating with an error (ERR), or (3) producing overfitted predicates, *e.g.*, 4 clauses instead of 2 for BST (WA). At the end, we were able to only synthesise 4 predicates with ShaPE.

### RQ 1.2: Efficiency

To understand what affects the synthesis efficiency, let us first look into the case studies with long synthesis times (more than 5 minutes): #13 (BST with list payload), #14 (balanced tree), #15 (binomial heap with order), #18 (list of lists with list payload), and #19 (rose tree with set payload). All these case studies feature large predicate sizes, together with either nested data structures or complex pure theory (integers or lists). As witnessed by the $\mathbf{T}_0$ and $\mathbf{T}_\mathbf{P}$ columns of Table 3.1, the time Sippy takes to obtain the expected predicate is less than 1 minute, and most runtime is spent on the exhaustive search to give the guarantee of local completeness (*cf.* Theorem 2). It is natural for nested data structures to take long time, because the increment of search space is applied to both the synthesis of the auxiliary predicate and the whole predicate.

We also wondered about how the used pure theories affect the synthesis efficiency. To answer that question, we compare two pairs of predicates (#12 v. #13, #17 v. #18) using the same input memory graphs but different theories: sets v. lists. The former is more efficient: this is because the list is more expressive than set (*e.g.*, a set union with itself is eliminated, but appending a list to itself produces a new list), so the search space is larger after the redundancy elimination. The same for integer theories: different combinations of integer operations lead to large search space.

### RQ 1.3: Input and Scalability

Our experiments demonstrate that 2-3 example graphs with tens of nodes in total (less than 20 nodes per case study on average in our benchmark suite) are sufficient to synthesise good predicates (we manually assessed the results'

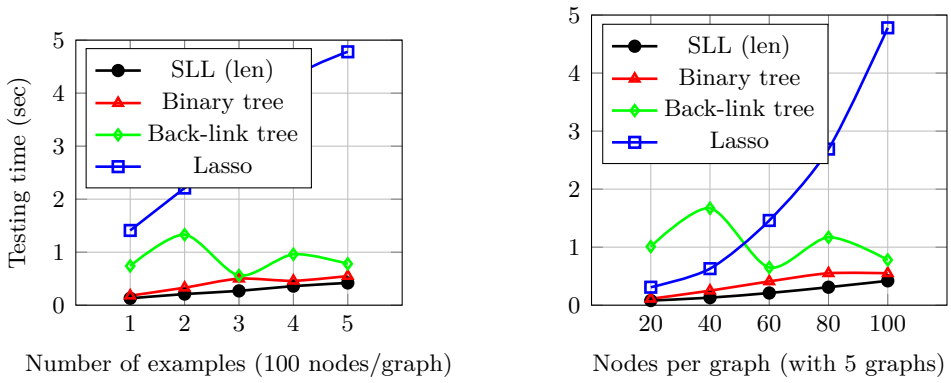



Figure 3.5: Testing time by graph count (left) and nodes per graph (right).

quality). We report the fraction of time it took to test the hypotheses during the search. In our case, it is not negligible for the examples with complex pure relations (*e.g.*, #13-15). In theory the test time should grow *slower than linearly* with the size of inputs. This is because only correct candidates require traversing all nodes by the Prolog unification. Since most candidate predicates do not satisfy all the examples, the testing is terminated when Prolog reaches the node that falsifies the example.

To show scalability of the tool *w.r.t.* the input size, Figure 3.5 shows the testing time by graph number with 100 nodes per graph on its left, and the testing time by node number with 5 graphs on its right. The general trends are as expected: the testing time grows slower than linear in the graph number, and grows linearly in the node number (the percentage of nodes traversed in the examples should be constant with the same topology of the graph, thus linear).

Below we discuss the outliers. The reader can notice the substantial difference between the testing time of the lasso list case study and every other example. The reason for that is our current implementation of the linearity check, which is not optimised for *circular* data structures. Note that the testing of SL predicates consists of two parts: the Prolog validity check and the linearity check (in § 3.4.2), we find that all but circular data structures' linearity checking time is negligible. This inefficiency also explains the faster-than-linear growth of testing times in Figure 3.5, and should be solvable by integrating the checking into Prolog's SLD resolution [95] (left

as future work). Finally, the testing time of back-link tree is not even strictly increasing. The reason is: providing more or larger examples makes it possible to prune earlier, thereby reducing the overall synthesis time.

### 3.6.2 The Utility of Sippy

#### 3.6.2.1 Verification

Let us demonstrate how the combination of Sippy and Grippy facilitates deductive program verification in SL-based provers. Specifically, our goal is to streamline the task of writing inductive predicates, by instead generating them automatically from programs to be verified. We are interested in the following research questions *w.r.t.* the effectiveness of the approach:

**RQ 2.1** *How much human effort is required to infer the predicates?*

**RQ 2.2** *How effective is Grippy for producing positive examples?*

**RQ 2.3** *Are the inferred predicates the same as the expected (human-written) ones, so they can be used directly for the verification?*

**RQ 2.1: Required Human Effort**

For this experiment, we adopted the case studies from benchmark suites of three different deductive verifiers [128, 132, 85], containing heap-manipulating programs for different linked data structures, many of which come with non-trivial data constraints. Our aim is (1) to quantify the human effort for annotating selected program(s) with assertions as oracles for graph generation, and (2) to confirm that Grippy can produce good-quality graphs for Sippy to synthesise the expected predicates within a reasonable time limit (10 min). For simplicity, we only consider the graphs without cycles unless being specified (doubly linked list in our case study).

Table 3.2 shows the statistics for generating up to 100 valid graphs, with up to five nodes, within the time limit of 10 minutes, to infer data structure predicates. In all cases, between 0 and 5 simple assertions (*i.e.*, one

Table 3.2: Statistics on generating input memory graphs for different predicates: the lines of code of the `assert`-annotated function or its test, the number of graph instances (capped by 100) generated in 10 minutes, the number of asserted conjuncts, the ratio of the number of valid graphs to all randomly generated graphs, and whether the original program is verified with the synthesised predicates against its original specification. For VeriFast and GRASShopper, **Y** means verified; we did not manage to compile VCDryad, so = means equivalent to the expected predicate, and **?** means not equivalent and couldn't be checked.

| Predicate | Programs | $LOC_{prog}$ | $LOC_{test}$ | Assert | Num | Ratio | Verified? |
|---|---|---|---|---|---|---|---|
| Singly Linked List | concat[2] | 10 | - | 0 | 100 | 100% | **Y** |
| | reverse[2] | 10 | - | 0 | 100 | 100% | **Y** |
| Sorted List | find[1] | 10 | - | 1 | 100 | 14.4% | = |
| | insert_iter[1] | 28 | - | 2 | 100 | 13.3% | = |
| | copy[2] | 26 | - | 2 | 100 | 13% | **Y** |
| | double_all[2] | 25 | - | 2 | 100 | 12.1% | **Y** |
| Doubly Linked List | append[1] | 11 | - | 2 | 47 | 2.9% | = |
| | dispose[2] | 9 | - | 2 | 41 | 2.5% | **Y** |
| | reverse[3] | 16 | - | 2 | 47 | 2.9% | **Y** |
| | insert_front[1] | - | 10 | 2 | 47 | 2.9% | = |
| Binary Search Tree | find[1] | 16 | - | 3 | 100 | 11.5% | = |
| | insert[1] | 22 | - | 6 | 100 | 8.0% | = |
| | free[3] | 11 | - | 3 | 100 | 7.5% | **Y** |
| | remove[3] | 43 | - | 3 | 100 | 12.3% | **Y** |
| Binomial Heap (order) | find_min[1] merge[1] | - | 26 | 5 | 18 | 1.1% | **?** |

[1] From VCDryad [128] [2] From GRASShopper [132] [3] From VeriFast [85]

single comparison for most cases, except for BST, one of whose assertions is expressed by a comparison function) are enough for capturing the properties (*e.g.*, line 7 in Figure 3.7).

### RQ 2.2: Effectiveness of the Graph Generator

As shown in the table, in each experiment Grippy produced at least 18 valid graphs, which is sufficient for Sippy to infer the expected predicates. Unsurprisingly, the throughput of the generator (*i.e.*, the Num column in Table 3.2) correlates with the complexity of the predicates: *e.g.*, the binomial heap instance needs to satisfy not only the order relation between the nodes but also the heap property, which results in low chances for the generator to produce valid instances; generally, the throughput is similar (ranged by randomness) across programs manipulating the same structure.

As expected, the fraction of valid graphs *w.r.t.* all randomly generated can be very low for complex predicates, which shows the importance of positive-only learning, as large numbers of trivially invalid graphs would significantly slow down the synthesiser.

Even though in principle Grippy could be used for any programs, we found that writing assertions for certain functions ( greyboxed in Table 3.2) is not an effective way to generate the valid instances. This is because the annotated node might simply not traverse "enough" of the structure to perform the validation. As an example, consider inserting a node to the front of a doubly linked list, shown in Figure 3.6. Testing the incorrect DLL heap graph in Figure 3.6 will not violate the assertion at line 9. The reason is: the assertion at line 9 is only checked for node `n1`, but the offending node `n2` is not checked because the function does not traverse the whole list. As a solution, in cases when no functions manipulating the structure traverse the whole structure graph (so their LOCs are not informative, hence "-" in Table 3.2), one can instead write a standalone *traversal function* to be used as an oracle. Sizes of those additional functions are shown as $\text{LOC}_{\text{test}}$ in Table 3.2.

We conclude that Grippy is a useful front-end to Sippy, but its effectiveness depends on the "thoroughness" of the structure traversal done by the function that is used as an oracle.

### RQ 2.3: Quality of Inferred Predicates

With memory graphs obtained automatically, we synthesised the respective predicates with Sippy and translated them into the syntax of the corresponding SL-based verifiers (automatically or manually, based on how complex the concrete verifier's language is). Next, we verified the original programs with the inferred predicates, thus, demonstrating that the inferred predicates are equivalent to the expected ones. An inferred sorted list predicate of VCDryad, corresponding to the example from § 3.2.2, is shown in Figure 3.7. The only failing case is the binomial heap with order constraints because of the limitation of pre-defined predicates in Sippy (further explained in

```
1  DLNode * insert_front(DLNode * x, int k) {
2    if (x == NULL) {
3      DLNode * head = (DLNode *) malloc(sizeof(DLNode));
4      head->key = k;
5      head->next = NULL;
6      head->prev = NULL;
7      return head;
8    } else {
9      if(x->next != NULL) assert(x->next->prev == x);
10     DLNode * head = (DLNode *) malloc(sizeof(DLNode));
11     head->key = k;
12     head->next = x;
13     x->prev = head;
14     return head;
15   } }
```

```
next(n1,n2).   next(n2,n3). next(n3,null).
prev(n1,null). prev(n2,n1). prev(n3,null).
```

Figure 3.6: Top: `insert_front` with an assertion only on the immediate successor (line 9). Bottom: an invalid DLL heap graph that Grippy produces yet does not violate the assertion, because n2 is never checked.

```
1    int sorted_find(SNnode * l, int k){
2      if (l == NULL) {
3        return -1;
4      } else if (l->key == k) {
5        return 1;
6      } else {
7        if (l->next != NULL) assert(l->key <= l->next->key);
8        int res = sorted_find(l->next, k);
9        return res;
10     } }
```

```
define pred sorted^(a):
 ((a l= nil) & emp) |
 ((a |-> loc next: c; int key: e) * sorted^(c) & (e lt-set keys^(c)))
```

Figure 3.7: Input program and inferred sorted-list predicate in VCDryad.

§ 3.6.3), where the synthesised predicates are partially correct but not strong enough, and need to be refined by manually adding the missing constraints.

To summarise, we found the combination of Grippy/Sippy effective for automatically producing SL predicates equivalent to human-written ones

from either modestly-annotated programs to be verified, or with the help of a simple human-written traversal procedure for the data structure.

#### 3.6.2.2 Deductive Synthesis

As another demonstration of Sippy's utility, we employed the synthesised SL predicates to automatically generate correct-by-construction heap-manipulating programs in C using a state-of-the-art deductive synthesiser SuSLik [133, 163]. The goal of this exercise was to demonstrate that, one can use Sippy together with SuSLik to obtain *provably correct* implementations of structure-specific procedures for copying, computing their size, and transformation without knowing how to specify SL predicates, but using heap graphs. Our case study includes 17 synthesis tasks involving the predicates from Table 3.1, producing programs *not* featured in any past works on SuSLik. The average code/spec AST size ratio is 4.1, and the average SuSLik synthesis time is 6.2 sec, which shows that Sippy produces predicates that are *immediately suitable* for proof-driven synthesis. Table 3.3 provides the detailed statistics. An example of the synthesised C program that transforms a sorted DLL into a singly linked list is given in Figure 3.8.

Compared to the existing example-based heap-manipulating program synthesisers, SPT [149] and Synbad [141], the joint Sippy/SuSLik workflow does not require "fold/unfold" functions (as does SPT) or a template (as needed by Synbad) for the intended programs.

### 3.6.3 Failure Modes and Future Work

In its current version, Sippy failed to synthesise predicates for several intricate linked structures. The reasons for the failed tasks fall into one of the following three categories:

1. Sippy's default settings cannot fully capture the structure's properties. Consider #15 from Table 3.1: the root and leaf nodes of binomial trees have different order relations with their siblings, but our search space cannot express this distinction, so the synthesised predicate is *the best*

| No. | Category | Program | Code/Spec | Time |
|---|---|---|---|---|
| 1 | Deallocate | sll | 5.5x | 0.2s |
| 2 | | bst | 8.0x | 0.2s |
| 3 | | dll_seg | 2.4x | 0.2s |
| 4 | | multilist | 16.0x | 0.3s |
| 5 | Copy | lseg | 2.0x | 0.8s |
| 6 | | bst | 3.5x | 3.3s |
| 7 | | balanced tree | 3.0x | 1.9s |
| 8 | Size | sll_len | 2.1x | 0.4s |
| 9 | | balanced tree | 3.5x | 0.6s |
| 10 | Transform | sll → dlseg | 2.5x | 0.4s |
| 11 | | srt_dll → sll | 3.1x | 7.4s |
| 12 | | dll → bst | 15.0x | 42.8s |
| 13 | | btree → bktree | 13.6x | 11.8s |
| 14 | | multilist → sll | 5.0x | 8.8s |
| 15 | | btree → dll | 9.6x | 7.1s |
| 16 | | bst → srtl | 11.6x | 10.3s |
| 17 | | dll → srt_dll | 7.3x | 9.3s |

Table 3.3: Programs synthesised by SuSLik from Sippy-inferred predicates.

```
// pre:  {f :-> x ** sorted_dll(x, z, s)}
// post: {f :-> y ** sll(y, s)}
void srt_dll_to_sll (loc f) {
loc x1 = READ_LOC(f, 0);
if (x1 == 0)  {
  WRITE_INT(f, 0, 0);
  return;
} else {
  int vx11 = READ_INT(x1, 0);
  loc nxtx11 = READ_LOC(x1, 1);
  loc z1 = READ_LOC(x1, 2);
  WRITE_LOC(f, 0, nxtx11);
  srt_dll_to_sll(f);
  loc y11 = READ_LOC(f, 0);
  loc y2 = (loc)malloc(2 * sizeof(loc));
  free(x1);
  WRITE_LOC(f, 0, y2);
  WRITE_LOC(y2, 1, y11);
  WRITE_INT(y2, 0, (int)vx11);
  return;
}}
```

Figure 3.8: An example SuSLik output (#11): a C program for converting a sorted DLL to SLL. `loc`, `READ_LOC`, *etc.* are macro-definitions around ordinary C types and operations, automatically generated by SuSLik [163].

*in this setting* (capturing nodes' ordering *w.r.t.* their children) but not the expected one.

2. Nested data structures with multiple arguments or complex pure relations. For example, the needed search space of the *braced list segment* predicate [138] is too large to be fully explored within our time limits.

3. An instance of a predicate cannot be proven by *top-down evaluation*, so that Prolog cannot evaluate them under the SL semantics.

In the first case, the solution is naturally to allow a richer search space: we might either extend the search space with more predicates (*e.g.*, judging whether a node is a root or not), or enable larger parameters in § 3.4.1 to enrich the expressiveness of Sippy. This is in line with a common synthesis trade-off between the expressiveness and the efficiency; we leave it to future work to find general search space settings to enrich the expressiveness without much loss of performance.

In the second case, it is possible to optimise the synthesis of nested predicates using problem-specific knowledge. For instance, we can assume the inner data structure is not mutually recursive (as it is in the case of the braced list segment), reducing the search space by splitting the synthesis of the auxiliary predicate and the whole predicate. We leave this optimisation to the future work.

To explain the last issue, consider the following Prolog predicate:

```
p(X, Y) :- X == Y.
p(X, Y) :- next(X, Z), p(Z, Y), p(X, Z).
```

In plain words: if `X` and `Y` have the same location, then `p(X, Y)` is true; if not, then `p(X, Y)` holds if both segments `p(X, Z)` and `p(Z, Y)` are true, where `Z` is the next node of `X`. This predicate is valid in Separation Logic, but Prolog rejects it, because of its validity checking algorithm. To understand the difference that causes the rejection, consider the literal `p(a, b)`, in the case when `next(a, b)` is a fact that holds. In SL, the literal is true, because it is checked in a *bottom-up* way, *i.e.*, "whether the predicate is consistent if it is true". Therefore, `p(a, b)` holds because it is consistent with `next(a,`

`b), p(b, b), p(a, b)`. However, the test of `p(a, b)` in Prolog is done in a top-down way, *i.e.*, "whether there is a variable unification that makes `next(a, Z), p(Z, b), p(a, Z)` true", which triggers a recursive test on the inner `p(a, b)`. It will be interesting to see whether replacing Prolog with the solvers from SL-COMP [146] can directly solve it without other overhead.

# Chapter 4

# FORCE: Inductive First-Order Formula Synthesis

*What we observe is not nature itself but nature exposed to our method of questioning.*

—Werner Heisenberg, *Physics and Philosophy* [81]

## 4.1 Introduction

This chapter presents FORCE, a top-down, generate-test-prune framework for synthesising inductive First-Order Logic invariants for distributed protocols. As motivated in § 1.3.2, the challenge is the *Discovery Problem* at the scale of First-Order Logic: given positive execution traces of a distributed system (*e.g.*, the Paxos consensus protocol), find the most specific conjunction of FOL formulas consistent with all traces that also constitutes an inductive invariant certifying system safety. The search space is exponentially large, and FORCE tames it by delegating formula enumeration and pruning to Answer Set Programming, introducing *orthogonal slices* as a novel technique for partitioning the space into manageable chunks. While our primary motivation and evaluation focus on invariant inference for distributed protocols, the core formulation and techniques developed in this chapter—synthesising strongest closed FOL formulas over finite sets of first-order structures via ASP-based enumeration and orthogonal

This chapter is based on a published paper by the author [165].

slicing—are general and broadly applicable to other specification mining and relational learning domains.

## 4.2 Overview

We start with the necessary background on First-Order Logic and transition systems, then introduce the synthesis problem, survey nine existing approaches to invariant inference, and conclude with an overview of our ASP-based FORCE framework and its orthogonal slices technique.

### 4.2.1 Background: First-Order Logic and Transition Systems

Invariant inference for distributed protocols requires two components: a logical language for specifying states and properties, and a formal model of protocol execution as a state machine.

**First-Order Logic.** Let $\mathcal{V}$ be a set of variables, $\mathcal{C}$ a set of constants, and $\mathcal{P}$ a set of uninterpreted predicate symbols over a vocabulary $\Sigma$. Formulas are built from atoms $p(t_1, \ldots, t_n)$ using connectives ($\wedge$, $\vee$, $\neg$, $\rightarrow$) and quantifiers ($\forall$, $\exists$). A *first-order structure* $M$ over $\Sigma$ interprets each predicate symbol as a concrete relation; $M \models \varphi$ denotes that formula $\varphi$ holds in $M$ under the standard Tarskian semantics. In the context of distributed protocols, the vocabulary $\Sigma$ captures entities such as nodes, quorums, and messages, and each protocol state is a first-order structure over $\Sigma$.

**Transition Systems and Inductive Invariants.** A transition system $\mathcal{T} = \langle S, I, T \rangle$ models a protocol as a state machine, where $S$ is the (potentially infinite) set of FO states over $\Sigma$; $I(V) \subseteq S$ is the set of initial states characterised by a formula over state variables $V$; and $T(V, V') \subseteq S \times S$ is the transition relation, expressing how the system evolves from a pre-state $V$ to a post-state $V'$. A *trace* $\tau = \langle s_0, s_1, \ldots, s_m \rangle$ is a finite, contiguous sequence of states generated by a sequence of atomic protocol events

$\langle e_1, \ldots, e_m \rangle$ starting from an initial state $s_0 \in I$, where each consecutive pair $(s_{i-1}, s_i) \in T$ (for $1 \leq i \leq m$) represents the immediate transition induced by $e_i$ with no intermediate events omitted between $e_{i-1}$ and $e_i$.

A *safety property* $P \subseteq S$ specifies the set of safe states; the system is safe if every reachable state lies in $P$. Because the exact reachable-state set is generally undecidable for infinite-state systems, safety is proved via an *inductive invariant* $Inv(V)$—a formula that over-approximates reachable states while excluding unsafe ones. Formally, $Inv$ must satisfy:

$$\textbf{Initiation:} \quad \forall V.\ I(V) \rightarrow Inv(V) \tag{4.1}$$

$$\textbf{Consecution:} \quad \forall V, V'.\ \Big(Inv(V) \wedge T(V, V')\Big) \rightarrow Inv(V') \tag{4.2}$$

$$\textbf{Safety:} \quad \forall V.\ Inv(V) \rightarrow P(V) \tag{4.3}$$

Discovering $Inv$ automatically is the central challenge FORCE addresses.

### 4.2.2 Problem Definition

**Definition 2** (First-Order Synthesis Problem)**.** Given a set of formulas $\Omega_0$ over signature $\Sigma$ and a set of FO structures $\sigma = \{M_1, \ldots, M_k\}$, where each $M_i$ is a model over $\Sigma$, find a set of formulas $\Phi = \{\phi_1, \ldots, \phi_n\} \subseteq \Omega_0$ s.t. $\forall \phi \in \Phi$:

1. $\forall M \in \sigma . M \models \phi$, (satisfies all input FO structures),
2. $\text{FreeVars}(\phi) = \emptyset$, (closed formula),
3. $\exists M \notin \sigma . M \not\models \phi$, (non tautology),
4. $\forall \phi' \in \Phi . \phi \neq \phi' \wedge \phi \not\models \phi'$, (no formula entails another).

In other words, the problem is to find in search space $\Omega_0$ a conjuncted set of well-formed formulas $\Phi$ that satisfy all the given first-order structures in $\sigma$. Such a conjunction describes the "most precise" formula that satisfies all the given structures, because there is no satisfied formula which is entailed by any $\phi_i$. The problem can be seen as an instance of the general specification synthesis problem [126] in the setting of positive-only learning [166]. Notably, Definition 2 depends only on a vocabulary $\Sigma$ and a finite collection of static FO structures $\sigma = \{M_1, \ldots, M_k\}$, rather than on the transition relation $T$ or

the sequential ordering of protocol traces: in distributed protocol verification, $\sigma$ is obtained simply by collecting the individual states visited along sampled execution traces, since any inductive invariant must hold as a state property at every reachable state. Consequently, although this chapter focuses on distributed protocol verification, the synthesis formulation and pruning techniques developed below are general and apply broadly to synthesising closed FOL specifications from finite FO structures.

It is worth noting that we make several assumptions and simplifications to the problem definition above: (1) a disjunction of all input FO structures is always a valid (though overfit) solution, but in practice, meaningful formulas are to be found in a size-restricted search space; (2) search spaces with function symbols and constants can be easily handled by introducing new literals [169, §4], so we will avoid discussing them in detail; (3) we assume users want to synthesise prenex DNF formulas, as distributed system (DS) invariants are commonly expressed in this form; our techniques can be extended to other FO formulas.

**Connection to Chapter 3 (Sippy).** While Chapters 3 and 4 target different verification domains, FORCE builds directly upon the core philosophy of Sippy while adapting it to a richer logical setting. First, both chapters operate in a *positive-only* learning regime: just as Sippy learns from positive memory graphs without negative heap instances, FORCE's synthesis phase (Definition 2) takes only positive protocol states $\sigma = \{M_1, \ldots, M_k\}$ sampled from valid traces (since *True* trivially satisfies all positive states) and searches for the *most specific* specification consistent with $\sigma$. Second, the two chapters differ in how specificity and hypothesis composition are realised due to their target logics: in Chapter 3, candidate hypotheses are recursive Horn clauses interpreted under Separation Logic, which cannot simply be conjoined without altering spatial disjointness, so Sippy uses clause normalisation and a size-based heuristic (`MAX_POL`) to select a single locally-optimal predicate; in Chapter 4, candidates are closed FOL sentences with universal and existential quantifiers ($\forall, \exists$) and negation, which can be freely conjoined ($\bigwedge_{\phi \in \Phi} \phi$), allowing FORCE to define specificity directly via

logical entailment ($\phi \not\models \phi'$) and compute the full Pareto frontier of strongest non-entailed conjuncts. Third, while Sippy delegates hypothesis testing to Prolog with a linearity check, FORCE evaluates candidate FOL formulas over finite FO structures and subsequently uses an SMT solver over the transition system $\mathcal{T}$ to filter the learned conjuncts into a sound inductive invariant.

**Example 1** (A tiny example)**.** Considering a FO signature $\Sigma = \langle C, R, F, S \rangle$ with $C$ and $F$ empty, $R = \{p, q, r\}$, and $S = \{X\}$. The synthesis problem is defined with the inputs:

- the set of FO formulas is restricted to the form of $\forall X\colon lit_1$ or $\forall X\colon lit_1 \vee lit_2$, a disjunction of one or two literals with one universal quantifier. Without pruning, $\Omega_0$ contains $(2{\times}3) + (2{\times}3)^2 = 42$ formulas.
- the set of FO structures is $\sigma = \{M_1, M_2\}$, where $M_1$ and $M_2$ are models over $\Sigma$ with universe $X = \{x_0, x_1, x_2\}$ and the interpretations:
  1. $M_1$: $[p^{M_1} = \{x_0,\ x_1\},\ q^{M_1} = \{x_1,\ x_2\},\ r^{M_1} = \{x_0,\ x_1\}]$
  2. $M_2$: $[p^{M_2} = \{x_0,\ x_1\},\ q^{M_2} = \{x_0,\ x_2\},\ r^{M_2} = \{x_0\}]$

And the output of the synthesis problem based on Definition 2 is a set of formulas

$$\Phi = \{\forall X.\, p(X) \vee q(X),\ \forall X.\, p(X) \vee \neg r(X)\}.$$

As a simple case of the sliced-template, our algorithm will first find satisfied formulas in $\forall X : lit_1$ (and fail), then check the satisfied formulas in $\forall X : lit_1 \vee lit_2$ (with the two formulas successfully found). The inputs of FORCE, the bounded FO search space and the real traces of distributed system protocols, are much more complex. An interested reader can refer to `configs/` and `traces/` folders in https://github.com/verse-lab/FORCE for the real instances.

### 4.2.3 Taxonomy of Existing Invariant Inference Algorithms

Existing invariant inference methods use different formula enumeration techniques and employ a variety of optimisations to effectively reduce

the search space. We summarise nine representative existing approaches: I4 [109], FOL-IC3 [92], IC3PO [73], SWISS [78], DistAI [169], P-FOL-IC3 [93], DuoAI [168], Scimitar [143], and Flyvy [60] (in the order of their publication dates). The categories are based on five detailed sub-aspects. The first two sub-aspects are "system-level", capturing the high-level design of a synthesis framework, while the remaining three are "algorithm-level": the pruning techniques applied to the brute-force enumeration of the search space.

### System-Level Aspects

**Inference mode.** This aspect determines the way the overall inference procedure uses FO formula synthesis. In particular, *one-shot* synthesis [109, 169, 168] generates all satisfied formulas in the search space given fixed input examples (*e.g.*, sampled traces of a distributed protocol), while *multi-shot* synthesis [143, 60] generates a set of candidate formulas incrementally, based on examples obtained incrementally (*e.g.*, counter-examples of current invariants). Combined approaches [92, 73, 78, 93] use system traces or counter examples to guide the multi-shot synthesis, calling the synthesis procedure multiple times.

Historically, the majority of multi-shot synthesis algorithms can be seen as extensions of IC3 [23], while one-shot synthesis can be considered as extensions of Houdini [58]. Crucially, both modes share the underlying FO synthesis problem similar to the definition of our synthesis problem, which means our FORCE framework can be used to improve most existing approaches.

**Language restriction.** To make the synthesis problem tractable, it is frequently defined for a subclass of first-order logic. The most common one is the Effectively Propositional Logic (EPR [131, 55]) fragment, a subset of FOL in which formulas can be transformed into equivalent propositional formulas, allowing for provably decidable verification.

While EPR and its extensions are applied in most existing approaches with a theoretical decidability guarantee, in practice, synthesis approaches

often impose additional syntactic constraints. For example, the k-pseudo-DNF (proposed in P-FOL-IC3 [93], and further applied to [60]) is a syntax restriction for practical efficiency: it is based on the observation that the invariant formulas written in such form are smaller than standard DNF, reducing the search space that needs to be explored. Specifically, a k-pseudo-DNF formula has the matrix in the form of $c_1 \rightarrow (c_2 \vee \ldots \vee c_k)$, where $c_i$ is a conjunction of literals. This is essentially a heuristic of the search, which makes sense because implications are commonly used to express invariants.

DistAI [169] proposed sub-templates (further applied to [168]) for efficiency, exploiting the following property of first-order logic with equality:

$$\forall X_1, X_2 : T.mat(X_1, X_2) \equiv \forall X : T.mat(X, X) \wedge \forall X_1 \neq X_2 : T.mat(X_1, X_2) \tag{4.4}$$

With such a property, formula synthesis can avoid enumerating formulas on the LHS of Equation 4.4 because they are equivalent to the conjunction of the two RHS formulas. Therefore, only enumerations among the two sub-templates on the right are required to synthesise the satisfied formulas on the left, resulting in an accelerated enumeration on the original search space.

More than syntactic constraints above, specific tools [92, 78, 143, 60] also define their own syntactic customisations. However, upon close examination of existing invariant inference implementations, we find that syntactic constraints are not extensible in many tools. For example, extending k-pseudo-DNF with sub-templates would require modifying any algorithm that manipulates formulas in P-FOL-IC3.

### Algorithm-Level Aspects

**Redundancy elimination.** The most common pruning to apply is redundancy elimination, which is used in many synthesis tasks. The idea is to simply eliminate the formulas that are equivalent to each other. Approximations as "pruning by symmetry" (to be detailed in § 4.3) are often used, where the symmetry of formulas under quantification is identified to prune away redundant ones. More than the syntactic-based equivalence, the redundancy can also be introduced by semantics (*e.g.*, tautology and

contradiction), where many case-by-case rules are used in DistAI [169] and DuoAI [168]. An example of formulas eliminated by this kind of redundancy is the one containing $p(X) \wedge \neg p(X)$ or $p(X) \vee \neg p(X)$ as sub-expressions. Another approach to redundancy is elimination *canonicalisation*, which is implemented by Flyvy [60]: it also uses symmetry breaking, but defines a partial order of formula sets (instead of the individual formulas) to eliminate redundancy.

A more complex but very effective pruning strategy of this category is proposed in DuoAI [168], where a large number of DNF formulas are shown to be redundant by their *decomposition*: $A \equiv B \wedge C$. Similar to Equation 4.4, the formula $A$ can be decomposed into a conjunction of smaller formulas, so the original formula can be pruned if all the smaller formulas are in the search space. However, the authors of Flyvy (*cf.* [60, Appendix D]) found that the decomposition in DuoAI is unsound in certain cases (*i.e.*, it leads to over-pruning), and proposed an amended version.

**Incremental pruning.** Another inherent part of nearly all efficient approaches to FOL formula synthesis is incremental pruning. The idea is to test formulas against the set of input FO structures in a specific order, exploiting the entailment relation between formulas to eliminate the need to test some of them.

The *implication graph* to be detailed in § 4.4.1 is an effective technique of this aspect from DuoAI [168]. More than this, DuoAI also identifies another incremental pruning technique based on *co-implication*, which combines the intermediate results of FO model checking with the redundancy elimination to prune the search space. For example, if we find that formula $\forall x.P(x) \Rightarrow R(x)$ satisfies all examples, we do not need to test any formulas of the form *prefix*. $(P(x) \wedge R(x) \wedge F) \vee G$, since they are co-implied by the original formula and *prefix*. $(P(x) \wedge F) \vee G$.

**Other techniques.** Several pruning techniques are introduced in existing invariant inference tools [92, 73, 143], but not generally applicable to all FO synthesis tasks.

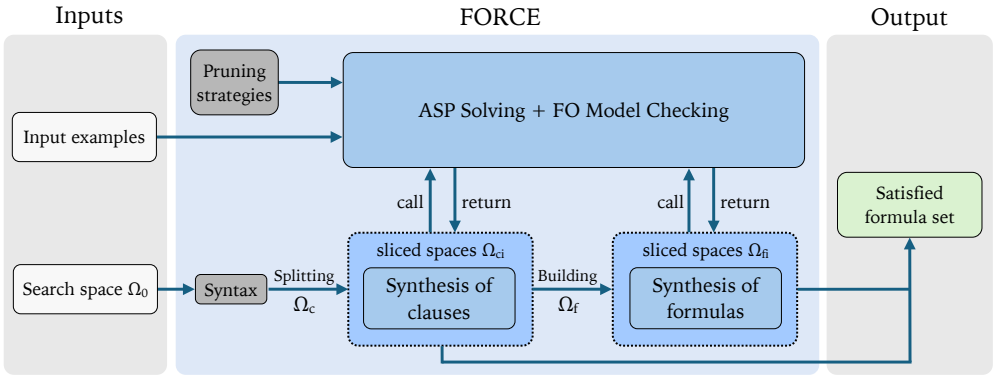



Figure 4.1: The workflow of FORCE.

To sum up the taxonomy of existing approaches, it suggests a possibility of a unified framework to capture the existing techniques for FO synthesis and propose new ones. In particular, we find that the **Language** and **Redundancy** aspects together form a *static* search space, while the **Incrementality** aspect defines the *dynamic* pruning of the search space during the synthesis. These aspects are applicable to any FO synthesis problem, though this chapter focuses on the synthesis of invariants for distributed protocols.

### 4.2.4 FORCE: ASP-based Synthesis + Orthogonal Slices

Before the technical details, we give a high-level overview of our ASP-based synthesiser FORCE, whose workflow is shown in Figure 4.1, by combining pruning strategies with the new *orthogonal slices* technique. The framework is customised by the grey parts: (1) the FO language (DNF by default) and (2) a set of pruning rules (predefined but extensible). Given the initial search space $\Omega_0$ and a set of FO structures as input, FORCE starts by splitting the search space of clauses $\Omega_c$ from $\Omega_0$. For further slices of $\Omega_c$ (called $\Omega_{c_1}$, $\Omega_{c_2}$, ...), the algorithm generates the formulas in $\Omega_{c_i}$ (by ASP solving), tests them on the examples (by model checking), and prunes the later slices ($\Omega_{c_j},\ j > i$) based on the results. Then using the output of $\Omega_c$, the algorithm further builds the search space of formulas *other than clauses* $\Omega_f$ into $\Omega_{f_1}$, $\Omega_{f_2}$, ..., reusing the same process as in $\Omega_c$ to generate-test-prune. Finally, FORCE

outputs both the clauses and the non-clause formulas that satisfy the inputs. It is called *orthogonal* because (1) slicing $\Omega_0$ into $\Omega_c$ and $\Omega_f$ and (2) slicing of $\Omega_c$ and $\Omega_f$ are using different pruning strategies and work together.

## 4.3 Static Search Spaces of FO Formulas in ASP

We assume the reader is familiar with the basics of ASP, such as rules, choices, aggregates, and refer to the literature for more details [64]. This section demonstrates how ASP is suitable for encoding the static search space of FO formulas.

Our high-level approach follows the "generate-and-test" workflow, similar to other invariant synthesis systems, but differs in how the search space and pruning techniques are encoded. ASP is a paradigm suitable for such exhaustive enumeration with restrictions (search space customisations and pruning techniques in our case). As shown in Figure 4.2, existing imperative approaches (demonstrated on the left) require manual integration of pruning steps (lines 3–6), which can lead to brittle and hard-to-maintain code due to dependencies on intermediate results and evaluation order. In contrast, our ASP-based approach on the right modularises the search space and pruning strategies directly into the solver (line 2). This ASP-based "generate-and-test" loop improves extensibility and maintainability, by allowing easier integration of new pruning rules (as new domain knowledge) and automatic handling of dependencies. In the remainder of this section, we first demonstrate how to encode a basic search for FO formulas and then show how the search is customised with different knowledge of pruning.

### 4.3.1 Encoding the Enumeration

Let us show how to encode FO formula enumeration in ASP. As an illustration, we use DuoAI's [168] configuration of FO search space for synthesising invariants of the `lockserv` distributed protocol.

```
var: node: n1, n2; lock: l1
```

```
1: tmp ← init()
2: for all x ∈ Ω0 do
3:     if pruning1(x, tmp) then
4:         continue
5:     else if pruning2(x, tmp) then
6:         update2(tmp, x)
7:     else
8:         if Satisfied(x) then
9:             tmp.update(x)
10:        end if
11:    end if
12: end for
13: return tmp.result
```

**Algorithm 4** Definition 2 with Pruning

```
1: function FOSyn(Ω0, σ, prunings)
2:     solver ← init(Ω0, prunings)
3:     while solver.solve() do
4:         x ← solver.current()
5:         if Satisfied(x) then
6:             yield x
7:         end if
8:     end while
9: end function
```

Figure 4.2: The synthesis loop (imperative programming vs. ASP solving).

```
relations: lock_msg:          node, lock;  grant_msg:  node, lock;
           unlock_msg:        node, lock;  holds_lock: node, lock;
           server_holds_lock: lock
max-literal: 4 max-or: 3 max-and: 3 max-exists: 1
```

The first two lines, "var" and "relations", together specify the variables (in `node` and `lock` sorts) and atoms (made out of five relations and valid variables) that can be used in the formulas. Note that the order of variables in formulas' prefix is usually fixed in EPR formulas, *i.e.*, the variable `l1` cannot appear before any node variable (`n1`, `n2`) in the prefix. The search space of formulas in DNF is then constrained by the problem-specific customisations: the maximum number of literals in the formula, the maximum number of disjunctions, the maximum number of cubes (*i.e.*, conjunctions of literals), and the maximum number of existential quantifiers. They together define the $\Omega_0$ in Definition 2.

Thanks to the simplicity of FO formulas' prenex normal form, the enumeration can be easily encoded in ASP. The enumeration of formulas requires generating their two parts: prefixes and matrices, within the restrictions of the search space. The ASP encoding of the basic search space is illustrated as follows:

```
var(node, n1). var(node, n2). var(lock, l1).          % variables
```

```
0{exists(V): var(V, _)}1.                                    % prefix
rel(lock_msg, (node, lock)). ...                             % relations
vars((node, lock), (n1, l1)). ...                            % var tuples
atom(P, A) :- rel(P, T), vars(T, A).                         % atoms
pos(0..1). cube(1..3).                                       % sign, cubes
0{lit_in_C(P,A,Pos,C): atom(P,A), pos(Pos)}3:-cube(C).% matrix
:- #count{P,A,Pos,C: lit_in_C(P,A,Pos,C)} > 4.               % max-literal
```

In the program above, the prefix and matrix generation of formulas are achieved by the choice construct `0{...}n` on `exists()` and `lit_in_C()` predicates, which are restricted by the parameters in the configuration. The last line then eliminates the answer sets where more than four literals are in the corresponding formula. *E.g.*, the answer set

```
{ lit_in_C(lock_msg, (n1,l1), 0, 1), lit_in_C(grant_msg, (n2,l1), 1, 1),
  lit_in_C(unlock_msg, (n1,l1), 0, 2), exists(l1) }
```

corresponds to the FOL formula

$$\forall \texttt{n1}\ \texttt{n2},\ \exists \texttt{l1}.$$
$$(\neg \texttt{lock_msg(n1,l1)} \wedge \texttt{grant_msg(n2,l1)}) \vee \neg \texttt{unlock_msg(n1,l1)}.$$

We should also note that the encoding provided here is to illustrate the basic idea of the enumeration; for different syntactic customisations (*e.g.*, the variable order in the prefix) of FOL formulas, certain predicates and rules are required to be added to the encoding.

This gives a concrete view of how invariant inference is instantiated for a distributed protocol. Given sampled execution traces of `lockserv` (each being a contiguous sequence of states $\langle s_0, s_1, \ldots, s_m \rangle$ produced by protocol transitions), each visited state in a trace is translated to a FO structure over the relations above, such as `lock_msg`, `grant_msg`, and `holds_lock`. The configuration is then compiled into ASP facts defining the typed variables, admissible atoms, prefix choices, and matrix choices. A stable model of this enumeration program represents one candidate formula: the `exists` atoms determine the quantified prefix, and the `lit_in_C` atoms determine the signed literals in each DNF cube. The candidate is reconstructed as a FO formula and evaluated on the trace structures. If it holds on all

of them and passes the well-formedness and non-tautology checks from Definition 2, it is kept as a potential invariant conjunct. Otherwise, the failed test is translated back into ASP constraints, so subsequent solver calls avoid the candidate and the related formulas ruled out by the pruning techniques in § 4.3.2 and § 4.4. Thus the `lockserv` example follows the same generate–test–prune loop as the larger protocols in the evaluation; the main contribution of the rest of the chapter is to make this loop efficient when the formula space becomes too large for direct enumeration.

### 4.3.2 Encoding Pruning by Redundancy

A particular encoding of a search space can output many answer sets whose corresponding formulas do not necessarily satisfy the examples or even basic well-formedness constraints. For instance, the formula $\forall n_1.\, \texttt{lock_msg}(n_2, l_1)$ should be ignored because $n_2$ is not in the prefix. As another example, the formula $\mathit{prefix}.\, \texttt{lock_msg}(n_1, l_1) \vee \texttt{lock_msg}(n_2, l_1)$ is equivalent to $\mathit{prefix}.\, \texttt{lock_msg}(n_2, l_1) \vee \texttt{lock_msg}(n_1, l_1)$ but can be featured twice as two different answer sets. Our next step is, therefore, to encode those **Redundancy** techniques from § 4.2.3 on the search space to only output well-formed formulas.

Let us illustrate the *symmetry-based normalisation* technique allowing us to exploit equivalence using ASP; the remaining encoding of redundancy elimination can be found in our implementation. The normalisations are done by building a partial order on the formulas. That is, we can make sure that (1) the number of literals in cubes (from left to right in a DNF) is non-decreasing, (2) the minimal (in alphabetic order) literal in a cube $i$ is less than the minimal literal in cube $j > i$ if their numbers of literals are the same, and (3) if two variables of one sort $v_i < v_j$, then the minimum predicate where $v_i$ appears should be less or equal to the minimum predicate where $v_j$ appears. The following ASP program encodes these pruning rules by constraining the orders in the formula.

```
lit_no_in(No, C) :- lit_in_C(P,A,Pos,C), lit_no(P,A,Pos,No).
num_lit(C, N) :- cube(C), #count{P,A,Pos: lit_in_C(P,A,Pos,C)} = N.
:- num_lit(C1,N1), num_lit(C2,N2), C1 < C2, N1 > N2.
```

```
min_lit(C, M) :- cube(C), #min{No: lit_no_in(No,C)} = M.
:- num_lit(C1,N), num_lit(C2,N), min_lit(C1,M1), min_lit(C2,M2),
   C1<C2, M1>M2.
min_lit_var(V, M) :- used_var(V),
   #min{P: lit_in_C(P,A,_,_), var_in(V,A)} = M.
:- min_lit_var(V1,M1), min_lit_var(V2,M2), V1 < V2, M1 > M2.
```

To summarise this section, we have shown that for the synthesis of FO formulas, the existing techniques can be easily encoded in ASP, which works as the backbone of our synthesiser. The expressive power of ASP allows us to encode existing techniques shown in § 4.2.3 concisely and make them work together efficiently, being further optimised with our new pruning technique, described in the next section.

## 4.4 Dynamic Search Spaces via Orthogonal Slices

Armed with the ASP-based framework for formula enumeration in the previous section, which allows one to combine pruning techniques, we propose *orthogonal slices* to handle the dynamic search space, which (1) generalises the state-of-the-art incremental pruning technique *implication graph* (IG), and (2) introduces a novel complementary pruning to resolve the bottleneck of IG. Both are achieved by slicing the FO search space into smaller ordered parts, and easily implemented by ASP's incremental solving.

### 4.4.1 Implication Graphs

Amongst the proposed pruning techniques for distributed system invariant synthesis in § 4.2.3, *implication graph* of FO formulas is a prominent one. An IG is a directed graph where each node represents a formula, and an edge from $A$ to $B$ indicates that $A$ implies $B$. The pruning is processed by removing formulas that are implied by already satisfied formulas from the search space, dynamically, to accelerate the synthesis process. That said, we can call it "pruning by satisfied formulas". The IG-based tool

$$\forall x.P(x) \vee R(x) \quad (4.5)$$
$$\forall x.P(x) \vee S(x) \quad (4.6)$$
$$\forall x.P(x) \vee (R(x) \wedge S(x)) \quad (4.7)$$

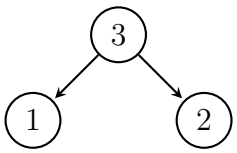



Figure 4.3: Left: example formulas. Right: their implication graph.

DuoAI [168] can infer inductive invariants for many complex protocols where no other existing tools can.

**Example 2** (An Illustration of the Implication Graph Shown in Figure 4.3)**.** Formula Equation 4.7 implies both formula Equation 4.5 and formula Equation 4.6. If formula Equation 4.7 is satisfied, then formulas Equation 4.5 and Equation 4.6 are automatically satisfied and can be pruned from the search space. The numbers in the right-hand graph are the indices of the formulas on the left: node 3 denotes Formula Equation 4.7, and its outgoing edges to nodes 1 and 2 denote the two entailment relations used for pruning.

### 4.4.2 From the Formula to the Search Space

The first step of the orthogonal slices is to *abstract* the entailment relation from formulas to the level of the search spaces, then synthesise formulas following the partial order on the search spaces to achieve incremental pruning. We represent a FO formula search space by means of imposing syntactic constraints of the candidates, with the sets of possible values as arguments. As such, the search space is the Cartesian product of those sets, where each element describes a sub search space.

**Definition 3** (Template of First-Order Formulas and Its Slicing)**.** A template $T$ of first-order formulas is defined by a tuple of parameter sets $T = (P_1, \ldots, P_n)$, where each $P_i$ represents a set of possible values for a parameter. The search space $\Omega(T)$ defined by $T$ is the Cartesian product of these parameter sets. A valid slicing of $T$ (called $\texttt{SL}(T)$) is defined as a partitioning of $T$ into sliced-templates $\{T_1, \ldots, T_j\}$, where each sliced-template

$T_i = (P_{i1}, \ldots, P_{in})$ corresponds to a subset of the parameter sets. Then $\Omega(T)$ is partitioned into subsets $\{\Omega(T_1), \ldots, \Omega(T_j)\}$, where:

$$\Omega(T) = \prod_{i=1}^{n} P_i. \qquad \bigcup_{i=1}^{j} \Omega(T_i) = \Omega(T), \quad \text{and} \quad \Omega_m \cap \Omega_n = \emptyset \quad \text{for all } m \neq n.$$

This ensures that the entire search space is covered, without overlap between slices, and each slice $\Omega_i$ corresponds to a sliced-template $T_i$.

**Example 3** (A Template of `lockserv` in § 4.3.1)**.**

- The number of existential quantifiers $ne$ in the prenex prefix (bounded by `max-exists: 1`),
- The tuple $tv = (n_{\text{node}}, n_{\text{lock}})$ recording how many variables of each sort (`node` and `lock`) appear in the formula (where the `lockserv` configuration declares at most 2 `node` variables `n1, n2` and 1 `lock` variable `l1`, and at least one variable must be used),
- The tuple $tl = (n_{l1}, \ldots, n_{lk})$ recording the number of literals in each DNF cube, sorted in non-decreasing order by the symmetry-breaking rules of § 4.3.2 (where the number of cubes $k$ is bounded by `max-or: 3`, each cube size $n_{li}$ is bounded by `max-and: 3`, and the total literal count $\sum_i n_{li}$ is bounded by `max-literal: 4`).

The template parameters (after the redundancy pruning in § 4.3.2) are:

- $P_{ne} = \{0, 1\}$
- $P_{tv} = \{(0,1), (1,0), (1,1), (2,0), (2,1)\}$
- $P_{tl} = \{(1), (2), (3), (1,1), (1,2), (1,3), (2,2), (1,1,2)\}$

Now we describe the slicing for "pruning by satisfied formulas", which is defined by the partial order of the sliced-templates (and their parameters).

**Definition 4** (Partial Order for Parameters of Templates)**.** Given two parameters $P_{ix}$ and $P_{iy}$ as subsets of $P_i$ in a template $T = (P_1, \ldots, P_i, \ldots, P_n)$,

we say that $P_{ix}$ is less or equal than $P_{iy}$ (denoted as $P_{ix} \preceq P_{iy}$; $P_{ix} \prec P_{iy}$ in case $P_{ix} \neq P_{iy}$) *w.r.t.* $\texttt{SL}(T)$ if, for all

$$T_1 = (P'_1, \ldots, P_{ix}, \ldots, P'_n),\ T_2 = (P'_1, \ldots, P_{iy}, \ldots, P'_n) \in \texttt{SL}(T),$$

every formula $\phi \in \Omega(T_1)$ can be weakened (via the syntactic weakening steps of Definition 6, or following the sub-template decomposition in Equation 4.4) to some formula $\phi' \in \Omega(T_2)$ (in particular, $\phi \models \phi'$ for entailment-based weakening). Because the redundancy constraints of § 4.3.2 exclude vacuous quantifiers and the weakening steps change the discrete template parameters monotonically, $\preceq$ is antisymmetric and thus forms a partial order on parameter slices (whereas semantic entailment alone on unnormalised slices would only induce a preorder).

**Definition 5** (Partial Order for Sliced-templates)**.** Given two sliced-templates $T_i = (P_{i1}, \ldots, P_{in})$ and $T_j = (P_{j1}, \ldots, P_{jn})$ sliced from $\texttt{SL}(T)$, we say that $T_i$ is less or equal than $T_j$ (denoted as $T_i \preceq T_j$) if $\forall k \in [1, n], P_{ik} \preceq P_{jk}$, where the equality holds if and only if $P_{ik} = P_{jk}$ for all $k \in [1, n]$.

With two definitions above, the whole search space of FO formulas is sliced (and ordered) by the partial order; we call this procedure SplitTem. Note that possibly the search space's parameters are not as regular as in Example 3, but the worst case of the partial order is exactly the implication graph: the set of possible formula candidates is the only parameter, and the partial order is defined by FO entailment (modulo logical equivalence). This says, the parameterisation of FO search space by syntactic constraints is "always" possible.

**Example 4** (Partial Order of the Templates in Example 3)**.** The partial order of the templates' parameters of `lockserv` is defined as follows:

- For $n_e$, it follows the integer order: $\{0\} \prec \{1\}$.
- For $t_v$, the set $\{(n_{v1}, n_{v2}, \ldots, n_{vi}, \ldots)\} \prec \{(n_{v1}, n_{v2}, \ldots, n_{vi} + 1, \ldots)\}$.
- For $t_l$, the set $\{(n_{l1}, n_{l2}, \ldots, n_{li}, \ldots)\} \prec \{(n_{l1}, n_{l2}, \ldots, n_{li} - 1, \ldots)\}$ and
  $\{(n_{l1}, n_{l2}, \ldots, n_{li}, \ldots)\} \prec \{(n_{l1} + 1, n_{l2}, \ldots, n_{li}, 1)\}$.

The partial order on template parameters is defined consistently with formula weakening and sub-template decomposition: by swapping $\forall$ into $\exists$, a formula becomes more general ($\forall x.\, M \models \exists x.\, M$); when increasing the variable count $n_{vi}$ of a sort by one, (i) replacing a repeated occurrence of a variable $x$ with a fresh existential variable $\exists y$ bound inside the scope of $x$ weakens the formula ($\forall x.\, M(x,x) \models \forall x \exists y.\, M(x,y)$), (ii) adding a fresh universally quantified variable $\forall y$ together with extra disjunctive cubes containing $y$ weakens a satisfied formula from the $n_{vi}$-variable slice ($\forall x.\, M(x) \models \forall x, y.\, M(x) \vee C(x,y)$), and (iii) for repeated universal variables, the formula decomposition of Equation 4.4 (proven in [169, §4]) splits the universal search space into the diagonal sub-template with $n_{vi}$ variables and the distinct-variable sub-template with $n_{vi}+1$ variables; finally, by deleting a literal from a cube or adding a new cube, a formula becomes more general. Therefore, by obtaining the partial order of a slicing of the search space, the "pruning by satisfied formulas" is naturally achieved.

**Definition 6** (Prefix and Matrix Weakening)**.** Consider two prenex DNF formulas

$$\phi = Q\bar{x}.\, M \qquad \text{and} \qquad \psi = Q'\bar{x'}.\, M',$$

where the matrices are disjunctions of cubes. We say that $Q'\bar{x'}$ is a *prefix weakening* of $Q\bar{x}$, written $Q\bar{x} \preceq_{pre} Q'\bar{x'}$, if $Q'\bar{x'}$ is obtained by one or more of the following sort-preserving relaxations in SplitTem: (i) replacing a universal quantifier $\forall$ by an existential quantifier $\exists$; (ii) replacing a repeated occurrence of a variable $x$ by a fresh existential variable $\exists y$ of the same sort bound within the scope of $x$ (which holds automatically in FORCE's fixed $\forall^*\exists^*$ prefix order whenever $x$ is universally quantified); or (iii) introducing a fresh universal variable $\forall y$ that does not occur in $M$ (to be used in additional cubes of $M'$). We say that $M'$ is a *matrix weakening* of $M$, written $M \preceq_{mat} M'$, if under a sort-preserving variable substitution consistent with $\preceq_{pre}$, each cube of $M$ is weakened by deleting zero or more literals, and $M'$ may additionally contain extra cubes.

By construction, both relations follow the entailment direction used by the implication graph: if $Q\bar{x} \preceq_{pre} Q'\bar{x'}$ and $M \preceq_{mat} M'$, then a formula

with prefix $Q\bar{x}$ and matrix $M$ entails the corresponding weakened formula with prefix $Q'\bar{x'}$ and matrix $M'$ (*i.e.*, $\phi \models \psi$).

The ASP encoding of "pruning by satisfied formulas" is done by multi-shot solving [66], which is standard to achieve incremental solving in ASP. The sketch of its encoding is as follows:

```
#program inv(pre, mat).
:- output(Pre, Mat), pre_weaken(Pre, pre), mat_weaken(Mat, mat).
```

which says that the formula entailed by the `inv` (*i.e.*, a satisfied formula) should not be generated. The entailment checking is implemented by variable substitution together with the weakening relations from Definition 6. These relations are finite for each sliced template, so they can be materialised by ASP facts such as `pre_weaken` and `mat_weaken`.

### 4.4.3 Slicing DNF Modulo Clauses

The problem of implication graph (or "pruning by satisfied formulas" in general) is its scalability: the search starts from the root to leaves of the graph (top to bottom in Figure 4.3), but the number of root nodes is still exponential in the search space's size. To see the issue, let us take the formula *prefix*. $(lit_{11} \wedge lit_{12} \wedge lit_{13}) \vee (lit_{21} \wedge lit_{22})$ as an example: it can be a root node used to prune the formula *prefix*. $lit_{1i} \vee lit_{2j}$ because of the entailment. However, checking all root formulas has complexity of $O(n^5)$ ($n$ is the number of literals), but this effort prunes only formulas using $O(n^2)$.

Intuitively, if "pruning by satisfied formulas" is costly when using the result of a larger search space to prune the smaller one, its dual version—"pruning by unsatisfied formulas"—should solve it. This idea, however, is not immediately applicable for two reasons: (1) the two pruning strategies have different directions (general-to-specific v. specific-to-general), which means an algorithm needs to deal with both, and (2) for complex problems, the majority of formulas are unsatisfied (they do not cover all examples), which means reducing the search space too many times incurs a large performance overhead.

| **Synthesis** | **ASP** | **Connection** |
|---|---|---|
| Parameterised search space | choice constructs | set Cartesian product |
| Slicing | external statement | set partition |
| Static pruning | integrity constraint | set comprehension |
| Dynamic pruning | multi-shot solving | set union, difference |

Table 4.1: The connection between synthesis and ASP.

Our solution to the first issue is simply another slicing, which is orthogonal to the slicing of templates: first synthesising clauses (disjunctions of literals) by slicing them from the whole search space (named SplitDNF), and then pruning the search of DNF synthesis based on the unsatisfied clauses. To further address the second issue, we avoid the high-cost of "pruning by unsatisfied clauses" by *constructing* the DNF search space from the satisfied clauses. The definition of DNF search space construction given below describes the pruning of this slicing.

**Definition 7** (Possibly satisfied DNF modulo clauses)**.** Given a set of satisfied (*w.r.t.* input examples) clauses $\Phi_c$, a formula in DNF of the form *prefix.* $(lit_{11} \wedge \ldots \wedge lit_{1k_1}) \vee \ldots \vee (lit_{m1} \wedge \ldots \wedge lit_{mk_m})$ is possibly satisfied only if

$$\forall (l_1, \ldots, l_m) \in \prod_{i=1}^{m} \{lit_{i1}, \ldots, lit_{ik_i}\},\ \textit{prefix.}\ l_1 \vee \ldots \vee l_m$$

is satisfied *w.r.t.* $\Phi_c$. The function constructing DNFs from satisfied clauses is denoted as BuildfromClauses.

In plain words, a formula in DNF is added into the search space only when all clauses it entails satisfy all input FO structures. The reason we resolve the bottleneck of "pruning by satisfied formulas" is evident: the search space of clauses, which is exponential in the number of cubes, is much smaller than the search space of DNF (exponential in the number of literals) for most cases. Moreover, since this search space construction is also naturally encoded by ASP, all existing techniques to formulas in § 4.2.3 are directly applied to further refine this search space built from clauses.

**Algorithm 5** The Core Algorithm of FORCE

1: **function** FORCE($\Omega_0$, $\sigma$, *prunings* = *pre_def*)
2: $\Phi_c,\ \Phi_f \leftarrow \emptyset$
3: $\Omega_c \leftarrow$ SplitDNF($\Omega_0$) ▷ slice into clause templates
4: **for** $\Omega_{ci} \in$ SplitTem($\Omega_c$) **do** ▷ Phase 1: clause synthesis
5: $\Phi_i \leftarrow$ FOSyn($\Omega_{ci},\ \sigma,\ prunings$)
6: $\Phi_c \leftarrow \Phi_c \cup \Phi_i$ ▷ collect satisfied clauses
7: $prunings \leftarrow prunings.update(\Phi_i)$ ▷ incremental pruning
8: **end for**
9: $\Omega_f \leftarrow$ BuildfromClauses($\Phi_c,\ \Omega_0$) ▷ DNF space from clauses
10: **for** $\Omega_{fi} \in$ SplitTem($\Omega_f$) **do** ▷ Phase 2: DNF synthesis
11: $\Phi_i \leftarrow$ FOSyn($\Omega_{fi},\ \sigma,\ prunings$)
12: $\Phi_f \leftarrow \Phi_f \cup \Phi_i$
13: $prunings \leftarrow prunings.update(\Phi_i)$
14: **end for**
15: $\Phi_c \leftarrow$ FilterImplied($\Phi_c,\ \Phi_f$) ▷ remove redundant clauses
16: **return** $\Phi_c \cup \Phi_f$
17: **end function**

### 4.4.4 The Synthesis Algorithm

Algorithm 5 combines the two slicing strategies into the complete FORCE procedure. It operates in two phases: the first phase (lines 4–8) synthesises satisfied clauses by iterating over sliced clause templates; the second phase (lines 10–14) uses those clauses to build a restricted DNF search space via BuildfromClauses and synthesises the remaining formulas. Throughout both phases, the pruning rules are updated incrementally so that formulas found earlier constrain the search in later iterations.

Given the formula synthesis algorithm in Algorithm 4 for an input search space, and the two approaches to slice the search space, the procedure for synthesising formulas is given by Algorithm 5: first synthesise clauses by its sliced templates (lines 4 to 8), then synthesise other formulas by the sliced templates of DNF modulo clauses (lines 10 to 14), finally normalising the satisfied formulas and outputting them (lines 15 and 16). The input *pruning* rules are customisable, but we pre-defined existing redundant and incremental rules in § 4.2.3. The soundness of the algorithm *w.r.t.* orthogonal slices (*i.e.*, it does not over-prune) is guaranteed by the fact that if a formula $A$ is pruned, $A$ is either unsatisfied, having been pruned

by unsatisfied clauses in DNF modulo clauses, or satisfied but more general than a satisfied formula which prunes $A$ in a sliced template.

Looking back to the whole synthesis process, both static and dynamic search spaces we discussed are essentially achieved by different set operations, which is the algorithmic reason for us to use ASP. In Table 4.1, we summarise the connection between the synthesis problem and ASP by unifying them as set operations, which hopefully helps the readers from either synthesis or ASP background to understand "why and what synthesis task is suitable for ASP".

## 4.5 Experimental Evaluation

### 4.5.1 Implementation and Settings

Our implementation of FORCE combines the use of ASP (250 lines for the enumeration plus 80 lines for pruning, which is extremely concise to encode the different existing and new pruning techniques), and C++ to call APIs of Clingo [64] (400 lines for the synthesiser call functions, and 200 lines to integrate with other systems). The benchmarks are run on a MacBook Pro with 8-core M1 Pro CPU, 16GB RAM. The current version of FORCE is available at `https://zenodo.org/records/15654427`.

We ran FORCE on a set of benchmarks from the work on DuoAI [168], the most efficient state-of-the-art DS invariant inference tool. Briefly speaking, DuoAI's algorithm is an enumeration-based inductive synthesis using sampled traces of the protocols as examples, followed by an optimised Houdini [58]. Since the formula enumeration component has non-negligible run time (detailed later), reducing the overhead of the enumeration part can significantly improve the overall performance.

Note that (at least) two unsound optimisations were made in DuoAI: the first is an *over-pruning*, found and rectified by [60]: DuoAI identifies formulas that can be decomposed into smaller formulas, then not testing the original formula to reduce the search space; however, a subset of those

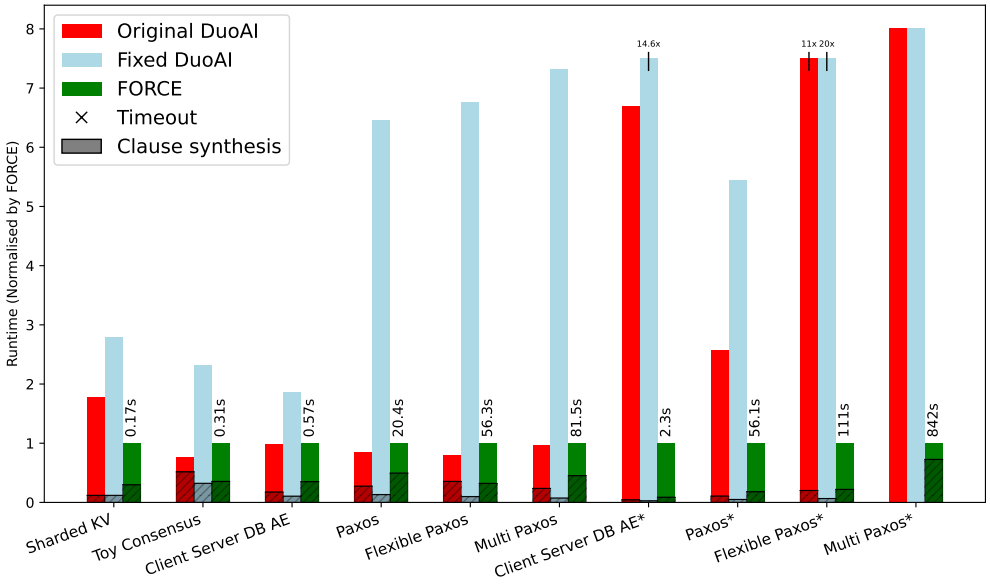



Figure 4.4: Invariant synthesis results, normalised *w.r.t.* FORCE runtimes.

should not be decomposable, which leads to unsoundness by missing possibly satisfied formulas. The second unsound optimisation is the use of *restricted quantifiers* of formulas; specifically, the default setting of DuoAI only allows arbitrary quantifiers over the last three variables in a template (*e.g.*, for $Q_1X_1.Q_2X_2.Q_3X_3.Q_4X_4$, $Q_1$ can only be $\forall$), which results in an incomplete search in general, but practically works for their benchmarks. We treat the first issue as a bug, and the second observation as domain-specific knowledge that encodes an extra constraint on the search space.

### 4.5.2 Results and Analysis

Our performance statistics are shown in Figure 4.4. We selected six complex distributed protocols from the DuoAI suite for evaluation on synthesising inductive invariants. FORCE was given the same input as DuoAI: traces as input examples and a search space configuration which contains the inductive invariant. For each protocol, we benchmark with two configurations: one with the restricted quantifier limitation from DuoAI and one without it. The unrestricted quantifier setting is indicated by a $*$ in the figure when enabled. The reason for showing only 10 data points, rather than 12, is

that the difference of quantifier restriction did not affect the small search space for the first two smaller protocols. We fixed the over-pruning bug in DuoAI (referring to the result as "fixed DuoAI") with our best effort for a fair comparison. That said, we also provide results of the "original" DuoAI version as a reference point in the figure.

The results show that FORCE significantly outperforms fixed DuoAI in all benchmarks, and also beats the original (unsound) DuoAI in most cases. The difference in performance is larger without the quantifier restriction. As a reference, the original DuoAI's runtimes to synthesise invariants for Paxos, FlexiblePaxos and MultiPaxos are 60.4s, 78.7s and 1,549s, *resp.*; this means the enumeration in DuoAI is the main bottleneck for complex protocols, so the improvement is effective for overall runtime.

The improvement is mainly sourced from the effective pruning by DNF modulo clauses (§ 4.4.3): as the *clause synthesis* part in the bars shows, FORCE takes a (slightly) longer time on the clause synthesis than DuoAI, and results in a much shorter time for the remaining synthesis by reducing the search space. The huge difference for cases without quantifier restriction is also explained for the same reason: taking Paxos as an example, the number of satisfied clauses increased from 134 to 141 when disabling the restriction in FORCE, which means the increased search space for DNF is not much (with 36s difference of time); but for DuoAI, the enumeration explores the whole new search space (thus 76s difference in total). Note that abstracting the formula into the search space as in § 4.4.2 allows parallelism of the formula synthesis in FORCE, which also contributes to the efficiency, but it is specific to (1) the implementation of parallelism, and (2) the multi-threaded solving in Clingo, which is not essential to discuss here. The overall extra time when disabling the parallelism in our system varies from 50% to 100%, which shows potential improvement with a better implementation of parallelism.

Another interesting comparison *w.r.t.* the unsoundness in DuoAI is shown between the red and blue bars: the fixed version takes a longer time to explore the (now larger) search space, but the clause synthesis time is lower, because additional discovered satisfied formulas prune the clauses

entailed (which also implies the unsoundness of original DuoAI). By analogy with our orthogonal slices, they are using results from "larger slices" to prune the "smaller slices", which results in inefficiency, as expected (see § 4.4.3). From a high-level aspect, this comparison illustrates the common trade-off between completeness and efficiency in synthesis tasks, but FORCE achieves complete search with better efficiency compared to the unsound version (original) of DuoAI by spending minor extra time (in the green bars) on synthesising the clauses.

### 4.5.3 On Composability of FORCE

More than the performance benefits of the orthogonal slices, the "solver-aided" feature of FORCE is also promising: it works as a general framework to combine different approaches by providing a common interface for FO synthesis. To illustrate this, we built the bridges for both DuoAI and Flyvy [60] (about 400 lines of code), and used the clause output (*i.e.*, part of Algorithm 5 before line 8) to optimise Flyvy.

The high-level task of Flyvy is to output the *strongest* inductive invariant (that contains a set of FOL formulas) given a distributed system protocol and a bounded FO language. Without describing the details of Flyvy's algorithm, it is clear that if Flyvy is obtaining formulas, those formulas can be bounded by DNF modulo clauses. We use the traces obtained from DuoAI together with the bounded language to synthesise satisfied clauses and output them as an ASP program by FORCE, then restrict the formulas in Flyvy by calling the ASP program to filter out the unnecessary formulas. The results show that, for the two most complex protocols that can be synthesised in Flyvy, Paxos and FlexiblePaxos, the sizes of the output invariants are reduced by about 13% (165/1260) and 7% (61/816) respectively on average, and they remain strongest in the sense that unnecessary formulas are removed during the intermediate steps of Flyvy. We believe it is a promising direction to explore combinations of existing approaches with the power of FORCE.

# Chapter 5

# SetLah!: A Language for Combinatorial Search

*Logic matters. Evidence matters too, but logic can be powerful all on its own.*

—Steven E. Landsburg, *The Armchair Economist* [98]

## 5.1 Introduction

This chapter presents SetLah!, a semi-declarative language for modelling combinatorial search problems. As motivated in § 1.3.3, the preceding two chapters reveal an irony: ASP—our chosen solution to the Specification Bottleneck—is itself subject to the same bottleneck at the modelling level. Writing ASP encodings directly requires expertise in stable model semantics; the constraints are rigid and error-prone. SetLah! resolves this second-order bottleneck by allowing users to express combinatorial search problems using arbitrary First-Order Logic constraints within a structured, block-based syntax, and automatically compiles these into efficient ASP programs via a two-phase translation pipeline.

The chapter assumes only standard first-order logic and the ASP background from Chapter 2. It is organised as follows: § 5.2 motivates and overviews SetLah! through running examples; § 5.3 defines the language formally; § 5.4 presents the translation pipeline and its correctness; § 5.5

This chapter is based on a published paper by the author [167].

evaluates SetLah! on diverse benchmarks; and § 5.6 gives a detailed comparison with related modelling languages and constraint-solving paradigms, including higher-order satisfiability.

## 5.2 Overview

In this section, we provide a high-level overview of the key concepts and motivations behind SetLah!, illustrated with the *orthogonal Latin squares* problem. To show the advantages of modelling in SetLah! in contrast to SAT and ASP, we first refer to Chapter 2 for brief introductions to these two paradigms.

### 5.2.1 A First Look at SetLah!: on Euler's Conjecture

A *Latin square* of order $n$ is an $n \times n$ array filled with integers from 1 to $n$, each occurring exactly once in each row and in each column. A *Graeco–Latin square* (equivalently, a pair of *orthogonal* Latin squares) of order $n$ can be viewed as two Latin squares $g$ (for Graeco) and $l$ (for Latin) on the same $n \times n$ grid such that the superposition $(l(R, C), g(R, C))$ yields a unique ordered pair in every cell (no pair repeats). Determining whether such objects exist is a classic combinatorial-design problem: the constraints are simple to state but tightly coupled, and the search space grows rapidly with $n$. A Latin square and a Graeco–Latin square of order 3 are illustrated in Figure 5.1.

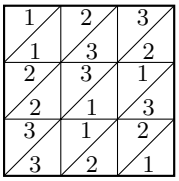
The great mathematician Euler conjectured in 1782 a sharp non-existence pattern for Graeco–Latin squares of orders $n \equiv 2 \pmod 4$, which was disproved only after nearly two centuries [91]. A general order $n$ will be used to illustrate the SetLah! modelling in comparison to SAT and ASP. We start with a single Latin square, then add the orthogonality constraints between two squares.

#### A Single Latin Square

Searching for a Latin square can be viewed as enumeration on atom $l(R, C, A)$ $(1 \leq R, C, A \leq n)$, meaning that cell $(R, C)$ contains num-

| | | |
|---|---|---|
| 1 | 2 | 3 |
| 2 | 3 | 1 |
| 3 | 1 | 2 |

(a) A Latin square of order 3




(b) A Graeco–Latin square

Figure 5.1: Order-3 Latin and Graeco–Latin squares (diagonally split cells).

ber $A$, where all three indices range from 1 to $n$. At the solver level, the constraint for the square to be Latin can be encoded in CNF as follows.

$$\bigwedge_{1\le R,C\le n}\Big(\bigvee_{1\le A\le n} l(R,C,A)\Big)\wedge \bigwedge_{\substack{1\le R,C\le n\\ 1\le A_1<A_2\le n}}\big(\neg l(R,C,A_1)\vee\neg l(R,C,A_2)\big) \quad \text{(one num/cell)}$$

$$\wedge \bigwedge_{\substack{1\le R\le n\\ 1\le A\le n}}\Big(\bigvee_{1\le C\le n} l(R,C,A)\Big)\wedge \bigwedge_{\substack{1\le R\le n\\ 1\le C_1<C_2\le n\\ 1\le A\le n}}\big(\neg l(R,C_1,A)\vee\neg l(R,C_2,A)\big) \quad \text{(one col/row, num)}$$

$$\wedge \bigwedge_{\substack{1\le C\le n\\ 1\le A\le n}}\Big(\bigvee_{1\le R\le n} l(R,C,A)\Big)\wedge \bigwedge_{\substack{1\le C\le n\\ 1\le R_1<R_2\le n\\ 1\le A\le n}}\big(\neg l(R_1,C,A)\vee\neg l(R_2,C,A)\big) \quad \text{(one row/col, num)}$$

Each line in the formulas above encodes a permutation constraint, using *at least one* clause (a disjunction of atoms) and *at most one* clause (a disjunction of two negated atoms). The greyed-out parts are redundant: the requirement that "each cell has exactly one symbol" already ensures that "each symbol appears at least once in each row and column." From a FO perspective, these conjunctions and disjunctions correspond to universal and existential quantification over finite domains (*i.e.*, the universe of all models). To clarify, the at least can be expressed as: $\forall R, C.\exists A.\ l(R,C,A)$. And the at most part is more non-trivial because of the $A_1 < A_2$: it is expressed as $\forall R, C, A_1, A_2.\ (A_1 < A_2 \rightarrow \neg(l(R,C,A_1)\wedge l(R,C,A_2)))$, which is a common *symmetry breaking* technique that halves the number of clauses compared to the more naive encoding using $A_1 \neq A_2$ instead.

While for the ASP encoding, the difficulties for at-most and at-least are different from SAT. Because the $\forall \bar{X}.(a_1(\bar{X}) \vee \cdots \vee a_k(\bar{X}))$ part is easily translated to $\nexists \bar{X}.(\neg a_1(\bar{X}) \wedge \cdots \wedge \neg a_k(\bar{X}))$, the at-most one part is straightforward in ASP as integrity constraints. The at-least one part is not directly expressible in the restricted FOL, since $\forall\exists$ cannot be translated

into $\nexists$ in general. Therefore, a possible practice is to introduce auxiliary predicates to capture the existential quantification. The whole ASP encoding is as follows (`dom/1` defines the shared domains $\{1, \ldots, n\}$):

```
dom(1). dom(2). dom(3). ... dom(n).
{ l(R, C, A) } :- dom(R), dom(C), dom(A).

latin_aux(R,C):- l(R,C,A).
:- dom(R), dom(C), not latin_aux(R,C).
:- dom(R), dom(C), l(R, C, A1), l(R, C, A2), A1 < A2.
:- dom(R), dom(A), l(R, C1, A), l(R, C2, A), C1 < C2.
:- dom(C), dom(A), l(R1, C, A), l(R2, C, A), R1 < R2.
```

With the stable model semantics, the predicate `latin_aux(R, C)` is derived when there exists some $A$ such that `l(R, C, A)` holds, then the following integrity constraint captures the at-least one condition with the help of the `latin_aux` predicate. The example stable model corresponding to the Latin square in Figure 5.1(a) contains the following atoms (with `..` omitting the enumeration for brevity):

```
dom(1), dom(2), dom(3),
latin_aux(1,1), latin_aux(1,2), latin_aux(1,3),
latin_aux(2,1), latin_aux(2,2), latin_aux(2,3),
latin_aux(3,1), latin_aux(3,2), latin_aux(3,3),
l(1,1,1), l(1,2,2), l(1,3,3),
l(2,1,2), l(2,2,3), l(2,3,1),
l(3,1,3), l(3,2,1), l(3,3,2)
```

### Defining a Latin Square in SetLah!

So, both SAT and ASP can model the Latin square problem, but with certain non-straightforward encodings for the constraints. Why not general FOL? This motivates our SetLah! language. Below is the SetLah! encoding for a Latin square:

```
[dom(1). dom(2). dom(3). ... dom(n).] % block 1
[
```

```
l(R, C, A) ?:- dom(R), dom(C), dom(A).
```
$\forall R, C.\ \big(\exists A.\ \mathrm{l}(R, C, A) \wedge$ `% one number per cell`
$\quad \forall A_1, A_2.\ (\mathrm{l}(R, C, A_1) \wedge \mathrm{l}(R, C, A_2) \rightarrow A_1 = A_2)\big)$
$\forall R, A, C_1, C_2.\ (\mathrm{l}(R, C_1, A) \wedge \mathrm{l}(R, C_2, A) \rightarrow C_1 = C_2)$
$\forall C, A, R_1, R_2.\ (\mathrm{l}(R_1, C, A) \wedge \mathrm{l}(R_2, C, A) \rightarrow R_1 = R_2)$
```
] % block 2
```

The program has two blocks (denoted in square brackets), starting from the domain definition and followed by the Latin square *rule* and *constraints*. The three constraints state that each cell has exactly one number, each number appears at most once per row, and each number appears at most once per column. These correspond exactly to the three lines in the CNF encoding above, with the first constraint combining both the "at least one" and "at most one" conditions into a single FOL formula.

Informally, the semantics of each block is a "guess-and-check" process built upon the output of previous blocks (detailed in § 5.3.2.2). A rule is either deterministic (`:-`) or non-deterministic (`?:-`) in the head-body form, which is similar to ASP's normal and choice rules, but restricted to positive atoms in the body. That is, a fixpoint computation similar to Datalog's immediate consequence operator $T_P$ can be generalized to operate on the *set of interpretations* instead of a single interpretation. The steps below illustrate the computational semantics of the program above.

**Initialization (Block 1):** This block takes the model set of the empty program $IS_0 = \{\emptyset\}$ as input. It applies all deterministic rules until reaching a fixpoint. For example,

- Starting from the empty model, `dom(1)` is added, yielding $IS_{1,1} = \{\{dom(1)\}\}$.
- Next, `dom(2)` is added to each interpretation in $IS_{1,1}$, resulting in $IS_{1,2} = \{\{dom(1), dom(2)\}\}$.
- After several iterations, all `dom/1` atoms are added, resulting in $IS_1 = \{\{dom(1), \ldots, dom(n)\}\}$.

**Guessing (Block 2):** Taking $IS_1$ as input, the non-deterministic rule explores all possible interpretations by branching on each choice atom in the block:

- **Branching:** for each interpretation $I \in IS_1$, the non-deterministic rule "guesses" whether to add the atom `l(R,C,A)` or not, for every combination of $R, C, A$ in the domain. For example, `l(1,1,1)` is either added or not to the only interpretation in $IS_1$, resulting in two interpretations in $IS_{2,1}$: $\{\{dom(1), \ldots, dom(n), l(1,1,1)\}, \{dom(1), \ldots, dom(n)\}\}$. Continuing this on the choice rule of `l(1,3,1)` for example, each of these two interpretations branches into two more, resulting in four interpretations.
- **Fixpoint:** This branching continues for all combinations of $R, C, A$ until reaching $IS_{2,k}$, where any further application of the rules on any interpretation in $IS_{2,k}$ yields no new interpretations. The order of branching does not matter because all combinations are eventually considered.

**Constraint Checking:** Finally, all FOL constraints are checked against each interpretation in $IS_{2,k}$, filtering out those that violate any constraint. The remaining set $IS_2$ forms the final **answer sets**—precisely the set of all satisfiable interpretations that satisfy every constraint, *e.g.*, the square in Figure 5.1(a) is in it.

The computational semantics naturally unifies the deterministic and non-deterministic rules (whether branching or not). The constraints are essentially *set operations* on the interpretations (the set of atoms) formalised in FOL, which is why SetLah! is a set-constraint language.

For making the combinatorial search work in practice, the computational semantics will not scale. The good news is that the semantics is consistent with stable model semantics when the FOL constraints are translated into ASP integrity constraints with equivalence. The translation process contains several phases, operating on the *block level* of the SetLah! program:

1. **FOL constraint decomposition:** Each FOL constraint is transformed into an equivalent normalized form using the logical equivalence rewrite rules of Table 5.1, then decomposed into simpler conjunctive components when possible. For example, the "one number per cell" constraint in block 2 corresponds to two separate CNF clauses in the SAT encoding, but appears as a single FOL formula in SetLah!. By applying the FOL equivalences from Table 5.1, we decompose it into $\forall R, C.(\exists A.l(R, C, A))$ and $\forall R, C, A_1, A_2.((l(R, C, A_1) \wedge l(R, C, A_2)) \rightarrow A_1 = A_2)$. This decomposition is crucial for grounding: a single formula with 5 variables requires $n^5$ ground instantiations, whereas the decomposed constraints require only $n^3 + n^4$ instantiations. While ASP programmers must manually perform such decomposition for efficiency, SetLah! automatically splits conjunctions (which might be hidden by complex FOL structures), significantly simplifying the modelling.

2. **Existential quantifier elimination (with possible auxiliary predicates):** Following standard logic programming techniques, existential quantifications are eliminated by introducing auxiliary predicates (akin to Skolemization). Alternatively, aggregate functions can express existential quantification without introducing new predicates. For instance, the "at least one" constraint translates to `:- dom(R), dom(C), #count{A: l(R, C, A)}=0.`, which forbids zero occurrences of `l(R, C, A)` for each pair $(R, C)$. Each FOL constraint thus translates to a set of ASP integrity constraints: by default we output the aggregated version which is smallest in size, but in case it is necessary (*i.e.*, cannot be expressed without auxiliary predicates) or required by the users, SetLah! can output the ASP programs with auxiliary predicates.

3. **Set operation desugaring:** Beyond FOL, SetLah! supports high-level set operations (subset, partition, and cardinality constraints) that are desugared into standard ASP aggregates or FOL formulas *before* the translation pipeline runs. This extension covers problems—such

as N-clique—that cannot be expressed in pure FOL with a fixed-arity formula.

### Orthogonality

From a single Latin square, we obtain a Graeco–Latin square by pairing two Latin squares $g$, $l$ of the same order and requiring orthogonality, *i.e.*, each pair $(A, B)$ to appear exactly once. In FOL, such constraints can be expressed naturally as: $\forall A, B.\exists R, C.(l(R, C, A) \wedge g(R, C, B))$.

For modelling the property in CNF, auxiliary atoms $p(R, C, A, B)$ for $l(R, C, A) \wedge g(R, C, B)$ are required[1], leading to the following encoding where three sub-CNFs are used for the bi-implication. The full CNF encoding used in the experiments is included in Appendix A.

$$\bigwedge_{1\leq A,B\leq n}\Big(\bigvee_{1\leq R,C\leq n} p(R,C,A,B)\Big) \wedge \bigwedge_{1\leq R,C,A,B\leq n}\big(\neg l(R,C,A)\vee\neg g(R,C,B)\vee p(R,C,A,B)\big)$$
$$\wedge \bigwedge_{1\leq R,C,A,B\leq n}\big(\neg p(R,C,A,B)\vee l(R,C,A)\big) \wedge \bigwedge_{1\leq R,C,A,B\leq n}\big(\neg p(R,C,A,B)\vee g(R,C,B)\big)$$

And for expressing the orthogonality in ASP, there can be different ways for the inner existential quantification as shown previously.

```
% With auxiliary predicate as in SAT
product(A, B) :- g(R, C, A), l(R, C, B).
:- dom(A), dom(B), not product(A, B).

% Or alternatively with Aggregates
:- dom(A), dom(B), #count {R, C : g(R, C, A), l(R, C, B) } = 0.
```

For SetLah!, the FO constraint can be directly added in the block for defining the second Latin square; with the translation, both ASP encodings above can be generated. In other words: each FOL formula encodes exactly one constraint.

One might argue that SMT (Satisfiability modulo theories, [42]) can be used for expressing all these FOL constraints, since the modern solvers (*e.g.*, Z3 [41], CVC5 [13]) support unrestricted quantifiers on theories like integer arithmetic and BitVector. However, SMT rarely scales well for

[1]It is possible to project out the existential quantifications of `R,C`, but the corresponding CNF becomes more complex.

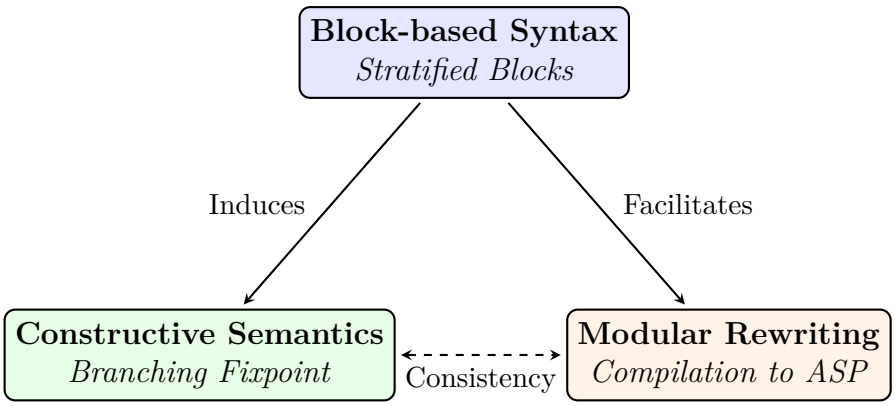



Figure 5.2: The Design Triangle of SetLah!: The stratified syntax induces a user-friendly constructive semantics while simultaneously facilitating efficient, modular rewriting to ASP.

difficult combinatorial problems with non-trivial quantifications due to lazy quantifier instantiation [136, 62]. Since the atom number for a single Latin square is $n^3$, the search space for "guessing" two squares is $2^{2n^3}$. We find SMT solvers struggle with $n \geq 6$ (unless the atoms are manually grounded), while SAT/ASP solvers can handle the case up to $n = 10$ in reasonable time with the same propositional search space (*cf.*, § 5.5). This should not be read as a claim that users normally write raw SAT: in practice, CNF is often the low-level target of a higher-level modelling language, much like assembly code is the target of a programming language. The point of SetLah! is to provide such a higher-level source language while preserving an ASP-level representation that remains inspectable and reusable.

### 5.2.2 The Design Triangle of SetLah!

The progression of the examples above illustrates the core design philosophy of SetLah!, which can be summarised in the **Design Triangle** shown in Figure 5.2. The three corners represent the key aspects of SetLah!'s design: *block-based syntax*, *constructive semantics*, and *modular rewriting*. Each aspect complements the others, forming a cohesive framework for declarative problem solving.

- **Block-based Syntax:** The blocks in SetLah! enable a clear separation between sub-problems, with the requirement of stratification ensuring that all atoms used in a block are defined in previous blocks. And in each block, the rules and constraints are written in a straightforward manner, essentially replacing ASP's unstratified negation with choice rules plus FOL constraints.
- **Constructive Semantics:** The stratified blocks induce a *Branching Fixpoint* semantics, which generalises Datalog's fixpoint computation to sets of interpretations. This semantics is intuitive and aligns with the "guess-and-check" model in combinatorial problems, and is consistent with stable model semantics when translated to ASP.
- **Modular Rewriting:** The clear structure of blocks allows for a modular translation process to ASP, where each block can be independently transformed. This modularity not only simplifies the translation but also ensures that the resulting ASP programs facilitate efficient solving.

## 5.3 Language Design of SetLah!

We present the syntax and the formal semantics of SetLah! in two stages. We first introduce *Positive Choice Logic Programming* (PCLP), a small kernel language that takes *choice* as a primitive—in contrast to ASP, which derives non-determinism from negation-as-failure. A program in PCLP takes a single answer set as its input and produces a set of answer sets via a least fixpoint computation. We then define SetLah! as *stratified PCLP*: a program is a sequence of PCLP blocks whose answer sets are chained together block by block.

### 5.3.1 PCLP: The Kernel Language

#### 5.3.1.1 Syntax

Figure 5.3 gives the grammar of PCLP and SetLah!. Variables are in upper case $(X, Y, \dots)$ and constants in lower case $(a, b, \dots)$; predicate symbols are lower case $(p, q, \dots)$. A PCLP program is a triple $P = (\mathcal{R}, \mathcal{C}, \Phi)$, where *definite rules* $\mathcal{R}$ are Horn clauses that deterministically derive atoms, *choice rules* $\mathcal{C}$ are called *choice Horn clauses*—clauses whose head is optional—that introduce non-deterministic branching, and *constraints* $\Phi$ are closed FOL sentences (no free variables) that filter the search space. The implication $\varphi_1 \rightarrow \varphi_2$ is syntactic sugar for $\neg\varphi_1 \vee \varphi_2$. All rule bodies are *positive* (contain no negations besides built-in comparisons); this restriction ensures monotonicity of the rule operators defined below, which in turn guarantees a well-defined fixpoint. Note that the comparison operators $op \in \{=, \neq, <, \leq, >, \geq\}$ in Figure 5.3 denote built-in relational comparisons over finite ordered domain constants (*e.g.*, bounded integers in a declared finite domain), rather than unbounded arithmetic expressions; because every variable in SetLah! ranges over a finite declared domain (§ 5.3.2) and rule heads do not construct unbounded arithmetic terms, SetLah! avoids the undecidability and safety pitfalls associated with unrestricted arithmetic in ASP. We write $\leftarrow$ in formal definitions and `:-` (resp. `?:-`) in the concrete syntax for definite (resp. choice) rules.

#### 5.3.1.2 Semantics

We define the semantics of $P$ relative to a single *input answer set* $I$ (a set of ground atoms). Let $\mathcal{A}$ be the *Herbrand base* of $P$, *i.e.*, the set of all ground atoms constructible from the predicates and constants in $P$ and $I$. We write $\sigma : \mathcal{V} \rightarrow \mathcal{T}_G$ for a *ground substitution* that maps every variable in $\mathcal{V}$ to a ground term in the Herbrand universe $\mathcal{T}_G$ of $P$ and $I$ (substituting away all variables); $\sigma$ lifts homomorphically to terms $\sigma(t)$ and atoms $\sigma(a)$, and for a rule body $\bar{L} = L_1, \dots, L_m$, we write $\sigma(\bar{L})$ for the set of ground literals $\{\sigma(L_1), \dots, \sigma(L_m)\}$.

**PCLP (kernel language)**

| | | |
|---|---|---|
| $t$ | $::= X \mid c \mid f(t_1, \ldots, t_k)$ | term |
| $a$ | $::= p(t_1, \ldots, t_k)$ | atom |
| $L$ | $::= a \mid t_1 \; op \; t_2$ | body literal |
| $\varphi$ | $::= a \mid \top \mid \bot \mid \neg\varphi$ | FOL formula |
| | $\mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2$ | |
| | $\mid \varphi_1 \to \varphi_2$ | |
| | $\mid \forall X.\, \varphi \mid \exists X.\, \varphi$ | |
| | $\mid t_1 \; op \; t_2$ | comparison |
| $r_d$ | $::= H \leftarrow L_1, \ldots, L_m$ | definite rule |
| $r_c$ | $::= \{H\} \leftarrow L_1, \ldots, L_m$ | choice rule |
| $P$ | $::= (\mathcal{R},\ \mathcal{C},\ \Phi)$ | program |

where $op \in \{=, \neq, <, \leq, >, \geq\}$.

**SetLah! (stratified PCLP)**

| | | |
|---|---|---|
| $d$ | $::=$ `ExtDOM` $D : \tau$ | external domain |
| | $\mid$ `DefDOM` $D : \tau$ $-\{c_1, \ldots, c_k\}$ | domain definition |
| | $\mid$ `ExtPred` $p : \tau_1 \times \cdots \times \tau_k$ | external predicate |
| | $\mid$ `Const` $c : \tau$ | constant |
| $B$ | $::= (\Sigma,\ P)$ | block |
| $\Pi$ | $::= (\mathcal{D},\ \langle B_1, \ldots, B_n \rangle)$ | program |

Figure 5.3: Grammar of PCLP and SetLah!.

**Rule and closure operators.** For the definite rules $\mathcal{R}$, let $\mathrm{Cl}_{\mathcal{R}}(J)$ denote the least fixpoint of the standard immediate-consequence operator $T_{\mathcal{R}}(K) = K \cup \{\sigma(H) \mid (H \leftarrow \bar{L}) \in \mathcal{R},\ \sigma(\bar{L}) \subseteq K\}$ starting from $J$, *i.e.*, the unique minimal $\mathcal{R}$-closed superset of $J$ (mandatory derivation). For a choice rule $r = (\{H\} \leftarrow \bar{L}) \in \mathcal{C}$, the *choice rule operator* $T_r$ (and the collective choice operator $T_c = \bigcup_{r \in \mathcal{C}} T_r$) grows the candidate set by adding the $\mathcal{R}$-closure of each optional branch alongside the existing candidates:

$$T_r(\mathcal{S}) = \mathcal{S} \cup \Big\{ \mathrm{Cl}_{\mathcal{R}}(J \cup \{\sigma(H)\}) \;\Big|\; J \in \mathcal{S},\ \sigma(\bar{L}) \subseteq J \Big\}, \quad T_c(\mathcal{S}) = \bigcup_{r \in \mathcal{C}} T_r(\mathcal{S}).$$

Built-in comparisons in $\bar{L}$ are evaluated directly over $\sigma$ rather than checked against $J$, and do not affect monotonicity of $T_{\mathcal{R}}$ or $T_r$.

**Candidate set.** Starting from $\{\mathrm{Cl}_{\mathcal{R}}(I)\}$, we apply choice rule operators until closure. The *candidate set* of $P$ from $I$ is the least $T_c$-closed set containing $\mathrm{Cl}_{\mathcal{R}}(I)$:

$$\mathcal{S}^*(P, I) = \mathrm{lfp}\Big(T_c,\ \{\mathrm{Cl}_{\mathcal{R}}(I)\}\Big).$$

Since $T_c$ is inflationary ($\mathcal{S} \subseteq T_c(\mathcal{S})$) and monotone on $2^{2^{\mathcal{A}}}$, the least fixpoint is well-defined, and by construction every element of $\mathcal{S}^*(P, I)$ is $\mathcal{R}$-*closed*, *i.e.*, closed under all definite rules in $\mathcal{R}$.

**Confluence and convergence.** The fixpoint is independent of the order in which choice rules are applied, so the semantics is deterministic. Formally, this follows from a diamond property: for any $r_1, r_2 \in \mathcal{C}$,

$$T_{r_1}(\mathcal{S}) \subseteq T_{r_2}(T_{r_1}(\mathcal{S})) \quad \text{and} \quad T_{r_2}(\mathcal{S}) \subseteq T_{r_1}(T_{r_2}(\mathcal{S}))$$

since every $T_r$ only adds $\mathcal{R}$-closed candidates to $\mathcal{S}$ and never removes them. When $\mathcal{A}$ is finite, convergence in finitely many steps is guaranteed; SetLah! enforces finiteness via domain declarations (§ 5.3.2).

**Answer sets of $P$.** Since every $M \in \mathcal{S}^*(P, I)$ is already $\mathcal{R}$-closed, the *answer sets* of $P$ from $I$ are those candidates satisfying every FOL constraint:

$$\mathcal{M}(P, I) = \Big\{ M \in \mathcal{S}^*(P, I) \;\Big|\; \forall \varphi \in \Phi,\ M \models \varphi \Big\}.$$

For a set of input answer sets $\mathcal{I}$, define $\mathcal{M}(P, \mathcal{I}) = \bigcup_{I \in \mathcal{I}} \mathcal{M}(P, I)$.

When $\mathcal{C} = \emptyset$, the fixpoint $\mathcal{S}^*(P, I)$ is the singleton $\{\mathrm{Cl}_{\mathcal{R}}(I)\}$ containing the unique minimal $\mathcal{R}$-closure of $I$, recovering Datalog semantics seeded from $I$. Conversely, PCLP captures NP: each ground choice rule $\{a\} \leftarrow$ corresponds to a Boolean variable, and FOL constraints subsume CNF clauses in SAT.

**Example 5.** Consider $P$ with the following components and input:

$$\mathcal{R} = \{b(X) \leftarrow a(X)\},\quad \mathcal{C} = \{\{a(X)\} \leftarrow d(X)\},\quad \Phi = \emptyset,\quad I = \{d(1),\, d(2)\}.$$

Starting from $\{\mathrm{Cl}_{\mathcal{R}}(I)\} = \{I\}$ (since no $a(\cdot)$ is in $I$): (1) the first application of $T_c$ branches on $a(1)$ and $a(2)$ and closes each under $\mathcal{R}$, adding $\mathrm{Cl}_{\mathcal{R}}(I \cup \{a(1)\}) = I \cup \{a(1), b(1)\}$ and $\mathrm{Cl}_{\mathcal{R}}(I \cup \{a(2)\}) = I \cup \{a(2), b(2)\}$; (2) applying $T_c$ again adds $\mathrm{Cl}_{\mathcal{R}}(I \cup \{a(1), b(1), a(2)\}) = I \cup \{a(1), a(2), b(1), b(2)\}$. Further applications of $T_c$ yield no new sets, so the fixpoint is $\mathcal{S}^*(P, I) = \Big\{ I,\ \ I \cup \{a(1), b(1)\},\ \ I \cup \{a(2), b(2)\},\ \ I \cup \{a(1), a(2), b(1), b(2)\} \Big\}$. Since $\Phi = \emptyset$, all four candidates are answer sets.

In summary, PCLP provides deterministic derivation, non-deterministic branching, and FOL filtering—but leaves open how to bound the search space and decompose multi-stage problems. SetLah! addresses both.

### 5.3.2 SetLah! as Stratified PCLP

SetLah! extends PCLP with two features: (i) *declarations* that specify domains, external input predicates, and constants, bounding the Herbrand base finitely so the fixpoint always terminates, and (ii) *stratification* into an ordered sequence of blocks, so that a multi-stage problem decomposes into dependent sub-problems solved one after another.

#### 5.3.2.1 Syntax

A SetLah! program $\Pi = (\mathcal{D}, \langle B_1, \ldots, B_n \rangle)$ consists of *domain declarations* $\mathcal{D}$ followed by a sequence of *blocks* (Figure 5.3, right).

**Declarations.** Each declaration in $\mathcal{D}$ introduces a named domain, predicate, or constant with an associated type $\tau$:

- `ExtDOM` $D : \tau$. declares an *external* domain whose constants are supplied as input (*e.g.*, node identifiers of a graph).
- `DefDOM` $D : \tau - \{c_1, \ldots, c_k\}$. declares an *internal* domain with explicitly listed constants (*e.g.*, `Set` : 1..9 for Sudoku digits).
- `ExtPred` $p : \tau_1 \times \cdots \times \tau_k$. declares an *external predicate* whose extension is supplied as input (*e.g.*, an adjacency relation).
- `Const` $c : \tau$. declares a named constant of type $\tau$ (*e.g.*, a parameter such as the square size $n$).

Every variable is typed by a declared domain, and each domain contains finitely many constants, so the set of possible ground atoms is finite. This bounds the Herbrand base $\mathcal{A}_{\mathcal{D}}$ and guarantees that the fixpoint always terminates.

**Blocks.** Each block $B_i = (\Sigma_i, P_i)$ pairs a set of *output predicates* $\Sigma_i$ with a PCLP program $P_i = (\mathcal{R}_i, \mathcal{C}_i, \Phi_i)$, where $\Phi_i$ contains arbitrary FOL sentences over the predicates visible to $B_i$.

**Stratification of a SetLah! program.** $\Pi$ is *well-formed* if for every block $B_i$:

- every predicate in the body of a definite rule in $\mathcal{R}_i$ belongs to $\Sigma_1 \cup \cdots \cup \Sigma_{i-1}$ or to $\Sigma_i$ (definite rules may be recursive within a block);
- every predicate in the body of a choice rule in $\mathcal{C}_i$ belongs strictly to $\Sigma_1 \cup \cdots \cup \Sigma_{i-1}$ (no self-referential guessing—a choice can only branch over facts already established);
- every predicate in a constraint in $\Phi_i$ belongs to $\Sigma_1 \cup \cdots \cup \Sigma_i$.

#### 5.3.2.2 Semantics

The semantics of $\Pi$ processes blocks left to right, feeding each block's output into the next. Let $I_0$ denote the set of ground facts supplied by external predicates and constants, and let $\mathcal{M}_0 = \{I_0\}$. Each successive block $B_i = (\Sigma_i, P_i)$ computes:

$$\mathcal{M}_i = \mathcal{M}(P_i, \mathcal{M}_{i-1}) = \bigcup_{I \in \mathcal{M}_{i-1}} \mathcal{M}(P_i, I),$$

using the finite Herbrand base $\mathcal{A}_{\mathcal{D}}$. The *answer sets* of $\Pi$ are $\mathcal{M}(\Pi) = \mathcal{M}_n$. In other words, each block generates and filters its own candidates, then passes the survivors as input to the next block.

**Expressive relationship with ASP.** While Section 5.4 focuses on compiling SetLah! into ASP, it is also natural to ask the converse question: how easily can an existing ASP program over a finite domain be expressed in SetLah!? Any *stratified* normal ASP program—or any program written in the standard "generate–define–test" methodology—maps directly into SetLah!: choice rules become PCLP choice rules in $\mathcal{C}_i$, positive and stratified definitions become definite rules in $\mathcal{R}_i$ ordered across blocks, and integrity constraints or aggregates become FOL/set constraints in $\Phi_i$. By contrast, ASP programs that rely on *unstratified negation cycles* (*e.g.*, mutual default negation `a :- not b.  b :- not a.` or odd-length negative loops) cannot be transcribed rule-for-rule into PCLP's strictly positive rule bodies. Even

so, because both SetLah! and normal ASP over finite domains capture NP search problems, such unstratified guesses can be reformulated in SetLah! by expressing the non-determinism via explicit choice rules $\{a\} \leftarrow$ in $\mathcal{C}_i$ and stating the stability or loop-free requirements as FOL constraints in $\Phi_i$.

**Relationship to ASP.** Three structural differences between SetLah! and ASP are worth noting: (1) PCLP takes choice as a primitive instead of deriving non-determinism from negation-as-failure, but the two mechanisms are equally expressive [122, 57, 160]; (2) PCLP's FOL constraints $\Phi$ admit arbitrary quantifiers and connectives, whereas ASP's integrity constraints are restricted to denials, but both equivalently filter the answer sets; and (3) SetLah!'s block structure is less monolithic than ASP's flat rule set, while the chaining semantics $\mathcal{M}_i = \mathcal{M}(P_i, \mathcal{M}_{i-1})$ remains consistent with stable model semantics. The latter two points are established by the translation from SetLah! to ASP in § 5.4.

## 5.4 Translating SetLah! Programs to ASP

One of the main technical contributions of this work is a compiler that translates SetLah! programs into efficient ASP encodings, automatically bridging the gap between arbitrary FOL constraints and the restricted integrity-constraint form that ASP solvers require. Without this translation, users would have to decompose every constraint by hand—precisely the modelling burden that SetLah! is designed to eliminate. We describe how the compiler translates a SetLah! program $\Pi$ into an equivalent Clingo ASP program. The translation operates block by block on each $P_i = (\mathcal{R}_i, \mathcal{C}_i, \Phi_i)$. It is worth distinguishing the standard (vanilla) logical transformations in this pipeline—such as propositional De Morgan/distributivity rewrites and classical Skolemisation—from the design choices specific to SetLah!'s semi-declarative modelling philosophy and ASP backend: (i) preserving the stratified block structure and user-defined rule vocabulary $(\mathcal{R}_i, \mathcal{C}_i)$ directly as ASP rules rather than flattening the entire model into opaque

SAT clauses; (ii) grounding-aware conjunction splitting via equality saturation, which targets the variable-arity bottleneck of ASP grounders; and (iii) model-preserving dual quantifier elimination (QE) that exploits native ASP `#count` aggregates alongside conservative-extension auxiliary rules. For space reasons, all proofs in this section are deferred to Appendix A.1.

### 5.4.1 Translating a Block

**Rules** The definite and choice rules translate directly into Clingo syntax: $H \leftarrow \bar{L}$ becomes `H :- L`$_1$`, ..., L`$_m$`.` and $\{H\} \leftarrow \bar{L}$ becomes `{H} :- L`$_1$`, ..., L`$_m$`.`. Since all bodies are positive and the Herbrand base $\mathcal{A}_\mathcal{D}$ is finite, no further transformation is needed.

**FOL constraints** Each $\varphi \in \Phi_i$ must be translated into a set of ASP *integrity constraints* (ICs), possibly introducing auxiliary predicates. An IC has the form:

$$\texttt{:-}\ \ell_1, \ldots, \ell_m.$$

where each $\ell_j$ is an *ASP body element*: an atom, the default negation of an atom (`not` $a$), a built-in comparison, or an aggregate expression—strictly more general than the body literals $L$ in PCLP (§ 5.3.1), which are restricted to atoms and comparisons. Semantically, the IC forbids any answer set from satisfying $\ell_1 \wedge \cdots \wedge \ell_m$. When the constraint contains variables $\bar{X}$, grounding replaces them with all possible constant tuples, so the IC corresponds to the negated existential:

$$\neg\exists\bar{X}.\ (\ell_1 \wedge \cdots \wedge \ell_m)$$

We call this the *IC form*: a negated existential over a conjunction of ASP body elements. The goal is to convert each $\varphi \in \Phi_i$ into a set of ICs equivalent over the finite Herbrand base $\mathcal{A}_\mathcal{D}$. Since arbitrary FOL sentences are not directly in IC form—for instance, a formula like $\forall X.\, \exists Y.\, p(X, Y)$ contains a quantifier alternation that has no direct IC encoding—the compiler performs

**Algorithm 6** TRANSLATECONSTRAINTS($\Phi_i$) for block $B_i$

**Require:** FOL constraints $\Phi_i$ from block $B_i$
**Ensure:** Auxiliary rules $\mathcal{R}_{\text{aux}}$ and integrity constraints $\mathcal{IC}$

$\mathcal{IC} \leftarrow \emptyset$; $\mathcal{R}_{\text{aux}} \leftarrow \emptyset$
**for** each $\varphi \in \Phi_i$ **do**
  *splits* $\leftarrow$ SPLIT($\varphi$) ▷ Phase 1: conjunction splitting
  **for** each $\varphi' \in$ *splits* **do**
    $E \leftarrow$ NORMALIZE($\varphi'$) ▷ Set of logically equivalent formulas
    **if** $\exists\, \psi \in E$ in IC form **then**
      $\mathcal{IC} \leftarrow \mathcal{IC} \cup \{\text{TOIC}(\psi)\}$ ▷ Phase 2: direct conversion
    **else**
      $(\psi, R) \leftarrow$ ELIMINATEQUANTIFIERS($E$) ▷ Phase 2: auxiliary QE
      $\mathcal{R}_{\text{aux}} \leftarrow \mathcal{R}_{\text{aux}} \cup R$
      $\mathcal{IC} \leftarrow \mathcal{IC} \cup \{\text{TOIC}(\psi)\}$
    **end if**
  **end for**
**end for**
**return** ($\mathcal{R}_{\text{aux}}$, $\mathcal{IC}$)

a series of equivalence-preserving rewrites to bridge this gap (§ 5.4.2). Since the cost of grounding in ASP depends on the domain of each variable, a high-level goal of the translation is to reduce the number of variables in each IC.

The translation proceeds in two phases. *Phase 1* (conjunction splitting, § 5.4.3) decomposes each $\varphi \in \Phi_i$ into independent conjuncts, reducing grounding cost. *Phase 2* (conversion to IC form, § 5.4.4) rewrites each conjunct into one or more ICs by applying the equivalences in Table 5.1 and, when necessary, quantifier elimination (QE) via auxiliary predicates or aggregates, before invoking TOIC($\psi$) to emit each resulting IC-form formula $\psi$ as a Clingo integrity constraint. Together with the directly translated rules from $\mathcal{R}_i$ and $\mathcal{C}_i$, the output forms a complete ASP encoding of the block $P_i$. Algorithm 6 summarises the procedure.

### 5.4.2 Equivalence Rewriting Rules

Both phases rely on a fixed set of equivalence rewriting rules (Table 5.1) applied to FOL formulas and their subformulas. The rules comprise *one-way reductions* ($\rightarrow$), *structural equivalences* ($\leftrightarrow$, such as commutativity,

Table 5.1: Equivalence rewriting rules. One-way rules ($\to$) strictly reduce formula size; bidirectional rules ($\leftrightarrow$) rearrange structure or are oriented by phase (toward CNF in Phase 1, toward DNF in Phase 2). Meta-variables $\varphi, \psi, \chi$ range over arbitrary formulas.

| Scope | Category | **Rule** (dual means swapping $\vee$ and $\wedge$) | **Effect on formula** |
|---|---|---|---|
| Propositional | Identity | $\varphi \wedge \top \to \varphi, \quad \varphi \vee \bot \to \varphi$ | Reduces size |
| | Annihilation | $\varphi \wedge \bot \to \bot, \quad \varphi \vee \top \to \top$ | Reduces size |
| | Idempotence | $\varphi \wedge \varphi \to \varphi$, dual | Reduces size |
| | Double negation | $\neg\neg\varphi \to \varphi$ | Reduces size |
| | Commutativity | $\varphi \wedge \psi \leftrightarrow \psi \wedge \varphi, \quad \varphi \vee \psi \leftrightarrow \psi \vee \varphi$ | Reorders operands |
| | Associativity | $\varphi \wedge (\psi \wedge \chi) \leftrightarrow (\varphi \wedge \psi) \wedge \chi$, dual | Regroups operands |
| | De Morgan | $\neg(\varphi \wedge \psi) \leftrightarrow \neg\varphi \vee \neg\psi$, dual | Redistributes $\neg$ |
| | Distribution | $\varphi \wedge (\psi \vee \chi) \leftrightarrow (\varphi \wedge \psi) \vee (\varphi \wedge \chi)$, dual | Duplicates/factors (oriented by phase) |
| First-order | Quantifier negation | $\neg\forall x\, \varphi \leftrightarrow \exists x\, \neg\varphi$ | Moves $\neg$ |
| | Quantifier commutation | $Q\,x.\, Q\,y.\, \varphi \leftrightarrow Q\,y.\, Q\,x.\, \varphi \quad (Q \in \{\forall, \exists\})$ | Reorders quantifiers |
| | Quantifier distribution | $(\forall x\, \varphi) \wedge (\forall x\, \psi) \leftrightarrow \forall x\, (\varphi \wedge \psi)$<br>$(\exists x\, \varphi) \vee (\exists x\, \psi) \leftrightarrow \exists x\, (\varphi \vee \psi)$ | Scopes/pulls quantifier (oriented by phase) |
| | Prenex conversion | $(\forall x\, \varphi) \vee \psi \leftrightarrow \forall x\, (\varphi \vee \psi) \quad (x \notin \mathrm{FV}(\psi))$<br>$(\exists x\, \varphi) \wedge \psi \leftrightarrow \exists x\, (\varphi \wedge \psi) \quad (x \notin \mathrm{FV}(\psi))$ | Scopes/pulls quantifier (oriented by phase) |

associativity, De Morgan, and quantifier negation/commutation), and *phase-oriented rules* (distribution and quantifier/prenex scoping, which are oriented toward CNF in Phase 1 and dually toward DNF in Phase 2 so that conflicting distribution directions are never mixed within a single phase). A key invariant is immediate from inspection of the rules:

**Observation 1** (Atom and variable invariance)**.** *No rule in Table 5.1 introduces fresh atoms or variables. For every formula $\psi$ reachable from $\varphi_0$,* $\mathrm{Atoms}(\psi) \subseteq \mathrm{Atoms}(\varphi_0)$ *and* $\mathrm{Vars}(\psi) \subseteq \mathrm{Vars}(\varphi_0)$.

This invariant, together with phase orientation, ensures that the rewriting system explores only finitely many formulas:

**Theorem 3** (Finiteness of the reachable set)**.** *For every formula $\varphi_0$ of finite size, the set of formulas reachable from $\varphi_0$ in each phase by applying the phase-oriented rules from Table 5.1 is finite.*

By Observation 1, every reachable formula is built from the same atoms and variables as $\varphi_0$. Orienting distribution and quantifier scoping by phase—toward CNF in Phase 1 and dually toward DNF in Phase 2—admits an AC-compatible polynomial interpretation [32] on the negation-normal form (specifically, $[\![A \vee B]\!] = [\![A]\!] \cdot [\![B]\!]$, $[\![A \wedge B]\!] = [\![A]\!] + [\![B]\!] + 1$, and

$[\![Qx.\, A]\!] = 3[\![A]\!]+1$ with leaf weight $\geq 2$ toward CNF, and dually with $\wedge$ and $\vee$ swapped toward DNF). Under this measure, commutativity, associativity, and quantifier commutation are invariant, while every other step on the NNF image—including both idempotences, distribution, factoring, and quantifier/prenex scoping—strictly decreases $[\![\cdot]\!]$. Combined with the finite bound on non-NNF negation placements via De Morgan and double negation, the set of reachable formulas in each phase is finite.

### 5.4.3 Phase 1: Conjunction Splitting

Given $\varphi \in \Phi_i$, the splitting phase finds a logically equivalent $\varphi'$ such that $\varphi' = \bigwedge_j \psi_j$, and then processes each conjunct independently. Splitting under universal quantifiers—$\forall X.\, (\psi_1 \wedge \psi_2) \equiv (\forall X.\, \psi_1) \wedge (\forall X.\, \psi_2)$—and through implications—$\varphi_1 \to (\psi_1 \wedge \psi_2) \equiv (\varphi_1 \to \psi_1) \wedge (\varphi_1 \to \psi_2)$—are obtained directly from the reachable set (Theorem 3). In principle, if splitting is not confluent (*i.e.*, if two irreducible forms are not bijectively equivalent), the compiler can explore all possible splits and choose the one that minimises grounding cost. In practice, the splitting is always confluent in our case studies. The general confluence of the rewriting system remains an open research question [31], and we leave a formal proof of this to future work.

By Observation 1, every conjunct $\psi_j$ satisfies $\mathrm{Vars}(\psi_j) \subseteq \mathrm{Vars}(\varphi)$, so splitting never increases grounding cost per conjunct. In practice, splitting can reduce grounding cost substantially (as in the Graeco-Latin squares of § 5.2.1) and can also simplify the subsequent conversion to IC form.

### 5.4.4 Phase 2: Conversion to IC Form

While Phase 1 rewrites within pure FOL, Phase 2 targets ASP: the rewriting rules are extended with ASP-specific constructs—predicates, negations and aggregate expressions—to convert each conjunct into IC form. As Algorithm 6 shows, each irreducible formula $\varphi'$ from Phase 1 is normalised again by applying the equivalences in Table 5.1 to obtain a finite set $E$ of logically equivalent formulas. If any $\psi \in E$ is already in IC form $(\neg\exists \bar{X}.\, (\ell_1 \wedge \cdots \wedge \ell_m))$, the sub-procedure ToIC($\psi$) (lines 7 and 11 of

Algorithm 6) strips the outer $\neg\exists\bar{X}$ prefix and emits the body directly as the ASP integrity constraint `:-` $\ell_1$`, ...,` $\ell_m$`.` (adding domain atoms $\texttt{dom_T}_i(X_i)$ for any variable $X_i$ that requires range restriction).

Otherwise, the formula contains a quantifier alternation that cannot be resolved by propositional rewriting alone, and the classic quantifier elimination (QE) is needed (to be further discussed in § 5.6.1). We provide two QE strategies.

**Auxiliary-based QE** The general QE strategy is essentially Skolemisation: given $\forall\bar{X}.\,\exists\bar{Y}.\,\psi(\bar{X},\bar{Y})$, the compiler introduces a fresh auxiliary predicate $\mathsf{aux}_k(\bar{X})$ (with a fresh $k$) and rewrites the body $\psi$ to DNF, $\bigvee_i(L_{i,1}\wedge\cdots\wedge L_{i,n_i})$, where each $L_{i,j}$ is a FOL literal. Each disjunct (called a *cube*) becomes a separate defining rule:

$$\mathsf{aux}_k(\bar{X})\ \texttt{:-}\ L_{i,1},\ \ldots,\ L_{i,n_i}. \qquad \text{(one rule per cube)}$$

The existential variables $\bar{Y}$ are absorbed into the rule bodies, and the original formula reduces to $\forall\bar{X}.\,\mathsf{aux}_k(\bar{X})$, encoded as the IC:

$$\texttt{:- not aux_k(}\mathsf{X}_1\texttt{, ...,}\ \mathsf{X}_j\texttt{), dom_}\mathsf{T}_1\texttt{(}\mathsf{X}_1\texttt{), ...}$$

The domain atoms $\texttt{dom_T}_i$ are ASP-specific to restrict each variable $X_i$ to its declared type $T_i$, inferred from the declarations (§ 5.3.2.1). The auxiliary rules' bodies may themselves contain the auxiliary predicate in cases of nested quantifier alternations, so the conversion *recurses* through the same pipeline (EliminateQuantifiers in Algorithm 6).

Note that auxiliary predicates extend the program's vocabulary: the resulting ASP program is a *conservative extension* (*a.k.a. quasi-equivalence* [96]), so its answer sets, projected onto the original signature $\Sigma$, coincide with those of the source SetLah! program—but the full answer sets (over $\Sigma$ plus auxiliaries) contain extra atoms not present in the SetLah! answer sets. This is harmless for correctness, yet it can complicate debugging since the solver's output includes auxiliary predicates the user did not write. When only a single layer of quantifier elimination is needed, an alternative strategy avoids auxiliary predicates entirely by using ASP aggregates, as described next.

**Aggregate-based QE** When a single existential quantifier block remains (no further alternation underneath), the compiler can instead eliminate it using an ASP aggregate, preserving the answer sets exactly. As above, the body under the quantifier is rewritten to DNF, $\bigvee_i (L_{i,1} \wedge \cdots \wedge L_{i,n_i})$. This DNF maps elegantly into ASP: each disjunctive cube becomes a condition joined by `;` inside the aggregate. For example, $\forall \bar{X}.\, \exists \bar{Y}.\, (L_1 \wedge L_2) \vee L_3$ becomes:

$$\texttt{:- \#count\{}\bar{Y}\texttt{:}\ \ L_1\texttt{,}\ L_2\ \texttt{;}\ \bar{Y}\texttt{:}\ \ L_3\texttt{\} = 0, dom_T}_1\texttt{(X}_1\texttt{), ...}$$

No new predicates are introduced, so the generated ASP program has exactly the same stable models over the original vocabulary. Since aggregate bodies cannot contain nested aggregates, this strategy cannot be applied recursively for deeper quantifier alternations.

**Strategy.** By default, the compiler prefers aggregate-based QE to minimise auxiliary predicates, keeping the generated program as close to the input specification as possible (not cluttered by fresh predicates). However, aggregates can make the grounder enumerate a large condition space (*cf.*, [71]). When grounding dominates runtime, SetLah! can instead prefer auxiliary-based QE: the existential search is materialised as ordinary rules, enabling the ASP grounder to share intermediate predicates and apply its usual rule-level optimisations. This is the grounding-scalability option used in the Alloy benchmarks (§ 5.5.2).

Regardless of which strategy is chosen, the translation is correct and terminates:

**Theorem 4** (Correctness and Termination of IC conversion)**.** *For every closed FOL formula $\varphi$ over a finite Herbrand base $\mathcal{A}_\mathcal{D}$, the translation procedure terminates and produces a finite set of ICs and auxiliary rules that is equivalent to $\varphi$ over $\mathcal{A}_\mathcal{D}$.*

### 5.4.5 Correctness

We show that the compiled ASP program admits exactly the same solutions as the source SetLah! program: every SetLah! answer set appears as a stable

model of the ASP output (after projecting away auxiliaries), and vice versa. The argument proceeds in two levels: first at the PCLP block level, then lifted to the stratified SetLah! level. Let $\Sigma$ denote the original signature (predicates declared in $\Pi$), and let $\hat{\Sigma} \supseteq \Sigma$ be the extended signature that includes auxiliary predicates introduced by QE. For an interpretation $M$ over $\hat{\Sigma}$, write $M \restriction_\Sigma$ for its projection onto $\Sigma$.

**Theorem 5** (Block-level preservation)**.** *Let $P_i = (\mathcal{R}_i, \mathcal{C}_i, \Phi_i)$ be a PCLP block, $I$ an input answer set over $\Sigma$, and $\hat{P}_i$ the ASP program produced by the translation. Then:*

1. *When $\Phi_i = \emptyset$, $\mathrm{AS}(\hat{P}_i \cup I) = \mathcal{S}^*(P_i, I)$.*
2. *When $\Phi_i \neq \emptyset$, $\left\{ M \restriction_\Sigma \;\middle|\; M \in \mathrm{AS}(\hat{P}_i \cup I) \right\} = \mathcal{M}(P_i, I)$.*

**Theorem 6** (SetLah! program correctness)**.** *Let $\Pi = B_1, \ldots, B_n$ be a well-formed SetLah! program, and let $\hat{\Pi}$ be the ASP program produced by translating each block $B_i$. Then*

$$\mathcal{M}(\Pi) \;=\; \left\{ M \restriction_\Sigma \;\middle|\; M \in \mathrm{AS}(\hat{\Pi}) \right\}.$$

**Implementation.** The translation of Phases 1 and 2 is implemented using EGGLOG [170], an equality saturation engine that explores equivalent formula representations over PCLP blocks. The extraction cost first favours formulas whose top-level conjunctions have been split, because this reduces the maximum number of variables per emitted IC. It then applies the QE preference described above: aggregate-based QE is cheaper by default, while the grounding-scalability option assigns lower cost to auxiliary-based QE. After translation, the resulting blocks are rewritten into ASP syntax and merged into a single Clingo program.

### 5.4.6 Extending SetLah! with Set Operations

Beyond FOL constraints, SetLah! supports a small collection of set-level operations. These operations do not extend the core PCLP semantics: they are surface constructs that are desugared into ordinary FOL formulas or

ASP aggregates *before* the translation pipeline of the previous subsections runs. Their purpose is pragmatic: they let users write common modelling idioms such as subset, partition, and cardinality constraints directly, while still compiling to the same FOL-to-ASP pipeline and to standard Clingo aggregate syntax when appropriate.

The idea is straightforward: the extension of a predicate can be named as a set of tuples, and operations such as subset or cardinality checks can be applied to that set. The compiler then expands each set atom locally. For example, subset becomes an implication under universal quantification, while cardinality and partition constraints become ASP aggregate constraints. Thus, set operations are best understood as notation for common finite-domain constraints, not as an additional semantic foundation for SetLah!. They complement the core FOL constraints when the natural specification is set-level—for instance, when a model needs cardinality over a parameterised relation. Details are provided in Appendix A.2.

## 5.5 Evaluation

In this section, we evaluate SetLah! on combinatorial search problems spanning constraint satisfaction, model checking, and quantum circuit synthesis. Our experiments are designed to answer the following research questions:

- *RQ1 (Expressivity & usability):* Can SetLah! effectively model common combinatorial tasks with high-level abstractions? Are the translation techniques—splitting and quantifier elimination—necessary for different problems?
- *RQ2 (Performance):* Given the remarkable advances in SAT solving over decades [59, 18], how does the SetLah! +Clingo workflow perform on different tasks, compared to SAT-based approaches?
- *RQ3 (Practical utility):* Can SetLah! help to solve real-world combinatorial problems, where advanced techniques like incremental solving in SAT are usually required?

Table 5.2: Comparison of SetLah! models written with FOL only and with FOL plus set operations. **#Block** gives the number of PCLP blocks; **#FO** and **#(FO+Set)** report the number of constraints in the pure-FOL model and in the model using set operations, respectively; **Split** and **QE** indicate whether conjunction splitting and quantifier elimination are used.

| **Problem** | **#Block** | **#FO** | **Split** | **QE** | **#(FO+Set)** | **Split** | **QE** |
|---|---|---|---|---|---|---|---|
| 15-puzzle | 1 | 8 | ✓ | ✓ | 8 | × | ✓ |
| Einstein riddle | 5 | 25 | ✓ | ✓ | 25 | × | ✓ |
| Graeco-Latin Square | 3 | 10 | ✓ | ✓ | 10 | ✓ | × |
| N-Queens | 1 | 3 | ✓ | ✓ | 3 | ✓ | × |
| N-clique | 1 | NA | | | 2 | × | × |
| Sudoku | 1 | 5 | ✓ | ✓ | 4 | ✓ | × |

For conducting the experiments with potentially time-consuming and multi-threaded tasks, we use an Ubuntu server with AMD EPYC 7763. The results other than massively parallel runs are also replicable with commodity PCs. For ASP solving, we used Clingo 5.8.0 which is the standard choice for ASP solving with multi-threading support. While for SAT solving, since numerous SAT solvers are available, we picked several well-known solvers from either the MiniSat [52] family (the most influential on later systems, with Python API and multi-threading support) or the CaDiCaL/Kissat [17] family (strong modern CDCL baselines). The specific solver choices are described in the corresponding sections.

### 5.5.1 RQ1: Modelling Common Combinatorial Tasks

To assess the expressivity of SetLah!, we implemented a suite of standard combinatorial problems from the Maple benchmark [25]. Maple is a computer algebra system based on MapleSAT [105]; in [25], problems such as N-queens and Sudoku are demonstrated as examples of its modelling capabilities. We use the same case studies to showcase SetLah!'s expressivity, investigating two questions: (1) which translation techniques—splitting and quantifier elimination (§ 5.4)—are needed for each problem, and (2) where set-level surface constructs make the model closer to the natural problem statement than pure FOL alone. Table 5.2 summarises the results. Each

problem is modelled with one constraint per logical check, reflecting how the problem is naturally stated, rather than conjoining all constraints into a single formula. Problems that decompose into sub-problems (*e.g.*, Einstein riddle with 5 blocks, Graeco-Latin Square with 3) directly benefit from SetLah!'s block structure.

As the table shows, set operations are primarily a modelling convenience rather than a separate solving technique. They generally do not reduce the number of constraints; instead, they replace quantified or relational boilerplate with direct set-level notation. For problems like Graeco-Latin Square, N-Queens, and Sudoku, cardinality constraints avoid spelling uniqueness through quantified FOL formulas, eliminating the need for quantifier elimination while still requiring splitting. Conversely, for 15-puzzle and Einstein riddle, set operations express relational bookkeeping more compactly, removing the need for splitting while quantifier elimination remains necessary. The performance contribution evaluated below is still the FOL-to-ASP pipeline—especially splitting and QE—whereas set operations explain why the source models can stay concise.

The concision benefit is not only the number of constraints, but their alignment with the mathematical specification. Taking the Einstein riddle as an example (a puzzle of 5 people, each with 5 attributes), the 25 FOL constraints without set operations correspond to the 15 given clues and 10 bijectivity constraints, naturally matching the problem statement and distributing the model across 5 blocks for the attributes. By contrast, the translated ASP program is a single monolithic encoding with 9 appearances of different forms of aggregates, which is less intuitive to implement even when the program size is comparable to the SetLah! version.

A notable case is N-clique, which *cannot* be expressed in pure FOL within SetLah! (marked NA) since the formula would require a conjunction of $N$ atoms, which is not achievable when the value of $N$ is not fixed at modelling time. With set operations, the problem is directly expressed by cardinality constraints. By contrast, N-queens *can* be expressed in pure FOL because its constraints are pairwise (no two queens attack each other), so they do not require an $N$-way conjunction.

The compilation time from SetLah! to ASP is short—around 1 second for most cases and up to 5 seconds across our full evaluation. Since compilation is performed once and the resulting ASP program is reused across instances, all runtime comparisons below exclude compilation time. Note that, unlike RQ1 where we use all valid set operations, later benchmarks only utilise set partition as the extension of constraints, which desugars into standard ASP aggregates (see Appendix A.2). The performance results are therefore attributable to the FOL constraint pipeline.

### 5.5.2 RQ2: Performance and Scalability

**GL Square Evaluation.** Among the cases in Table 5.2, we select the Graeco-Latin Square for a detailed runtime investigation: the fixed-size puzzles (15-puzzle, Einstein riddle, and standard $9 \times 9$ Sudoku) are solved within seconds, whereas for the other parameterised problems, we refer the reader to existing studies (*e.g.*, N-queens [20], N-clique [94]). The GL square is also a historically significant benchmark: in [25], the author notes that "Euler could have disproved the conjecture (with 23 hours) if given Maple." We also tried first-order modelling with SMT solvers, including finite model finding in CVC5 [135], but none scaled beyond small orders.

All encodings—including the Maple and SMT attempts discussed above—apply standard symmetry-breaking by fixing the first row and column (*i.e.*, the *reduced* GL square). We compare SetLah!-generated ASP encodings against SAT solvers: CaDiCaL 1.9.5 and Glucose 4.2 for sequential runs (via PySAT [83]), and ParKissat-RS [127] and CryptoMiniSat 5.1.3 [151] for parallel runs. Each configuration runs with 10 random seeds to account for heuristic variance, under timeouts of 86,400s (sequential) and 7,200s (parallel). We report *PAR-2* (mean runtime where timeouts count as $2\times$ the limit) and the *Coefficient of Variation* (CV% under PAR-2), which captures how sensitive each solver is to random-seed choices. Table 5.3 compares three execution modes: Sequential (1T), Parallel (8T), and Massively Parallel (64T).

Table 5.3: PAR-2 Performance of GL Square in seconds ($\text{Mean}_{\text{CV\%}}$). Comparison of Sequential (1T), Parallel (8T), and Massively Parallel (64T) execution. *Timeout = 86,400s (sequential), 7,200s (parallel)*

| | Sequential (1 Thread) | | | Parallel (8 Threads) | | | Parallel (64 Threads) | | |
|---|---|---|---|---|---|---|---|---|---|
| **Ord** | **SetLah!** | **CaDiCaL** | **Glucose** | **SetLah!** | **Parkissat** | **CMS** | **SetLah!** | **Parkissat** | **CMS** |
| 6 | $10.7_{13\%}$ | $\mathbf{9.9}_{8\%}$ | $12.0_{0\%}$ | $\mathbf{2.6}_{12\%}$ | $9.9_{5\%}$ | $39.8_{7\%}$ | $\mathbf{1.7}_{12\%}$ | $3.9_{13\%}$ | $14.5_{10\%}$ |
| 7 | $\mathbf{0.6}_{69\%}$ | $0.8_{53\%}$ | $2.7_{1\%}$ | $\mathbf{0.2}_{35\%}$ | $1.3_{7\%}$ | $1.7_{95\%}$ | $\mathbf{0.2}_{17\%}$ | $1.3_{4\%}$ | $0.4_{4\%}$ |
| 8 | $\mathbf{50}_{69\%}$ | $72_{110\%}$ | $73_{0\%}$ | $10.6_{71\%}$ | $23.1_{36\%}$ | $\mathbf{2.7}_{3\%}$ | $\mathbf{0.9}_{109\%}$ | $4.6_{39\%}$ | $3.0_{2\%}$ |
| 9 | $2941_{110\%}$ | $1073_{83\%}$ | $\mathbf{259}_{1\%}$ | $618_{126\%}$ | $1053_{79\%}$ | $\mathbf{426}_{1\%}$ | $\mathbf{34}_{78\%}$ | $239_{83\%}$ | $76_{13\%}$ |
| 10 | $82780_{80\%}$ | $\mathbf{7157}_{85\%}$ | $172800_{0\%}$ | $\mathbf{4425}_{120\%}$ | $8396_{76\%}$ | $13621_{18\%}$ | $\mathbf{504}_{135\%}$ | $1915_{92\%}$ | $1602_{13\%}$ |

**Analysis.** The key message from Table 5.3 is not that any single solver dominates—the high CV% values confirm that runtime is heavily influenced by solver heuristics.

In the sequential baseline, CaDiCaL leads on larger orders (Order 9–10), which is unsurprising given decades of SAT-specific engineering. However, CaDiCaL does not natively support parallel execution. Clingo, by contrast, scales seamlessly: at Order 10, moving from 1 to 8 threads yields a super-linear speedup (from 82,780s to 4,425s), surpassing both parallel SAT solvers.

With 64 threads, Clingo achieves the best PAR-2 across *all* orders, confirming that ASP solving is a strong competitor for hard combinatorial problems even against well-known SAT solvers.

**Grounding Scalability: Alloy's benchmarks.** The GL Square experiment tests solving competitiveness; we now examine whether SetLah!'s compilation also makes the grounding scalable. Unlike the GL Square, where solving dominates, many practical instances are bottlenecked by the translation from high-level specification to solver-specific encodings. We evaluate on benchmarks adapted from Alloy [84], a mature modelling tool (over 20 years of engineering) based on FOL with relational algebra and SAT as backend. Its constraint language is similar to the FOL constraints in SetLah!, making these benchmarks a natural fit. Two differences between Alloy and SetLah! should be noted. First, Alloy targets properties of software systems rather than general combinatorial problems. Second, its backend Kodkod [155] applies *equisatisfiable* transformations—rewrites that preserve satisfiability but not necessarily the exact set of models (in contrast

to SetLah! which preserves the quasi-equivalent models as mentioned in § 5.4.4). Thus the comparison is not a claim that the two tools implement identical encodings; it isolates whether SetLah!'s FOL-to-ASP translation can instantiate comparable relational constraints more efficiently.

In both tools, the expensive preprocessing step is a form of grounding: Kodkod instantiates bounded relational formulas into propositional SAT variables and clauses, while SetLah! compiles first-order constraints into ASP rules and lets Clingo ground those rules. The semi-declarative block structure also matters for modelling these examples. For instance, `academia_3` first chooses whether each person is faculty or student, and only then chooses whether chosen students are graduate or undergraduate. This staged dependency is written directly as two SetLah! blocks; flattening it into one ASP stratum would force the modeller to manually recover the same dependency structure.

We selected four representative instances from [112] that span two axes: *recursive vs. non-recursive* definitions, and *SAT vs. UNSAT* outcomes. For each, we scale the input size and compare grounding and solving times between Alloy (version 6.2, compiled to Glucose) and SetLah! (compiled to Clingo, single-threaded). As discussed in § 5.4.4, the auxiliary-based QE option can improve grounding scalability by replacing large aggregate bodies with shareable ordinary rules. We enable this option for the Alloy benchmarks because their runtime is dominated by grounding rather than by search. Table 5.4 presents the results.

**Analysis.** The first three benchmarks show consistent advantages for SetLah!, for both grounding and solving phases. The speedup is predominantly grounding-driven: SetLah!'s translation to Clingo produces ground programs that are substantially smaller than those generated by Alloy's Kodkod backend, so that the grounding time is reduced. In other words, the splitting in SetLah!'s translation (§ 5.4) effectively mitigates the grounding bottleneck for large instances, resulting in better scalability. The grounding time dominates the total runtime across those three benchmarks for SetLah!; while in academia_2 at $n$=500, Alloy's solving time spikes to 96s (compared to under

Table 5.4: Scalability comparison of Alloy vs. SetLah!. Times in seconds (Grd = grounding, Slv = solving). Speedup columns show the ratio (positive means SetLah! is faster, negative means Alloy is faster).

**genealogy** (recursive, SAT)
7 definitions, 11 constraints

| | Alloy | | SetLah! | | Speedup | |
|---|---|---|---|---|---|---|
| $n$ | Grd | Slv | Grd | Slv | Grd | Total |
| 15 | 0.14 | 0.05 | 0.04 | 0.00 | 3.5× | 4.8× |
| 30 | 1.24 | 0.80 | 0.41 | 0.09 | 3.0× | 4.1× |
| 45 | 6.42 | 1.54 | 2.06 | 0.87 | 3.1× | 2.7× |
| 60 | 22.21 | 12.84 | 6.65 | 0.95 | 3.3× | 4.6× |
| 75 | 58.76 | 23.08 | 16.85 | 1.32 | 3.5× | 4.5× |

**academia_3** (recursive, UNSAT)
17 definitions, 22 constraints

| | Alloy | | SetLah! | | Speedup | |
|---|---|---|---|---|---|---|
| $n$ | Grd | Slv | Grd | Slv | Grd | Total |
| 25 | 0.38 | 0.08 | 0.05 | 0.00 | 7.6× | 9.2× |
| 50 | 2.67 | 0.49 | 0.38 | 0.00 | 7.0× | 8.3× |
| 75 | 10.14 | 1.52 | 1.43 | 0.00 | 7.1× | 8.2× |
| 100 | 25.08 | 1.52 | 3.45 | 0.00 | 7.3× | 7.7× |
| 125 | 49.17 | 7.46 | 8.05 | 0.00 | 6.1× | 7.0× |

**academia_2** (non-recursive, SAT)
8 definitions, 9 constraints

| | Alloy | | SetLah! | | Speedup | |
|---|---|---|---|---|---|---|
| $n$ | Grd | Slv | Grd | Slv | Grd | Total |
| 100 | 0.72 | 0.22 | 0.12 | 0.00 | 6.0× | 7.8× |
| 200 | 4.09 | 0.97 | 0.53 | 0.02 | 7.7× | 9.2× |
| 300 | 11.40 | 2.29 | 1.13 | 0.03 | 10.1× | 11.8× |
| 400 | 24.79 | 12.62 | 2.24 | 0.05 | 11.1× | 16.3× |
| 500 | 44.85 | 96.45 | 3.59 | 0.07 | 12.5× | 38.6× |

**birthday** (non-recursive, UNSAT)
4 definitions, 4 constraints

| | Alloy | | SetLah! | | Speedup | |
|---|---|---|---|---|---|---|
| $n$ | Grd | Slv | Grd | Slv | Grd | Total |
| 5 | 0.01 | 0.00 | 0.01 | 0.00 | ±1.0× | ±1.0× |
| 10 | 0.06 | 0.03 | 0.03 | 0.10 | 2.0× | −1.4× |
| 15 | 0.19 | 0.06 | 0.12 | 1.94 | 1.6× | −8.2× |
| 20 | 0.29 | 0.08 | 0.44 | 15.28 | −1.5× | −42.5× |
| 25 | 0.61 | 0.14 | 1.31 | 81.01 | −2.1× | −109.8× |

0.1s in SetLah!), indicating that the redundancy of Kodkod's propositional encoding also contributes to solving overhead. Note that the "recursive" benchmarks use recursion only for transitive closure, which both Clingo and Kodkod handle natively; the grounding advantage of SetLah! is therefore not specific to transitive closures but more generally to recursive definitions.

The **birthday** benchmark reveals the differences. Unlike the others, this instance models a transition system rather than a classical combinatorial search problem, and the search space grows exponentially with depth. Kodkod's automatic symmetry-breaking is particularly effective here because model checking of transition systems is inherently highly symmetric. SetLah! struggles from $n$=15 on solving time without such automated techniques; grounding also grows faster, likely due to the frame problem [80]: FOL must explicitly state non-change of unaffected variables, whereas Alloy's "override" operator handles this naturally. Bringing manual symmetry-breaking in SetLah! could close this gap, but we use only the plain FOL encoding matching Alloy's specification. This is a limitation of the current surface language rather than a fundamental limit of ASP as a backend: a frame or override construct could desugar to ASP programs (at least capturing

the propositional encoding in Alloy) in the same way that set operations desugar to aggregates. We leave such transition-system-specific modelling support to future work.

### 5.5.3 RQ3: Qubit Layout Synthesis as a Case Study

A key advantage of SetLah!'s translation is *solver transparency*: the output is a reusable ASP program, not only a one-shot ground instance (unlike SAT-based modelling tools, *cf.* § 5.6.2), so it can be embedded in Clingo's multishot API and solved incrementally or in parallel (§ 5.6.2). We demonstrate this on *quantum circuit layout synthesis* [153] from Q-Synth v2 [144], a real-world problem where multishot programming [66], analogous to incremental SAT [52], is essential.

The task is to map logical qubits of a quantum circuit onto the physical topology of a quantum processor while minimising the number of SWAP gates, which is NP-complete. Q-Synth formulates each mapping step as a satisfiability problem solved incrementally: the SWAP count is increased step by step until a valid layout is found (SAT) or the bound is exhausted (UNSAT); see [144] for details of the original problem and method, which are orthogonal to our experimental focus.

**Modelling comparison.** Similar to how we demonstrate the GL Square in § 5.2, Q-Synth's original encoding [144] uses 8 parameterised variables with 5 groups of constraints—already concise on paper, but implemented through a CNF-generation layer, totalling roughly 600 lines of Python code. The SetLah! encoding is substantially more compact: 9 declarations, 9 blocks with 11 rules and 22 constraints expressing the same logic. The compiled ASP program is about 40 lines, with about 60 lines of Python glue code needed to integrate it into Q-Synth. The SetLah! model is shown in Appendix A.3.

The blocks follow the structure of the synthesis algorithm thanks to the *semi-declarativeness* of SetLah!: the case when the choice on one predicate is attempted only after the previous choice is fixed appears several times in the

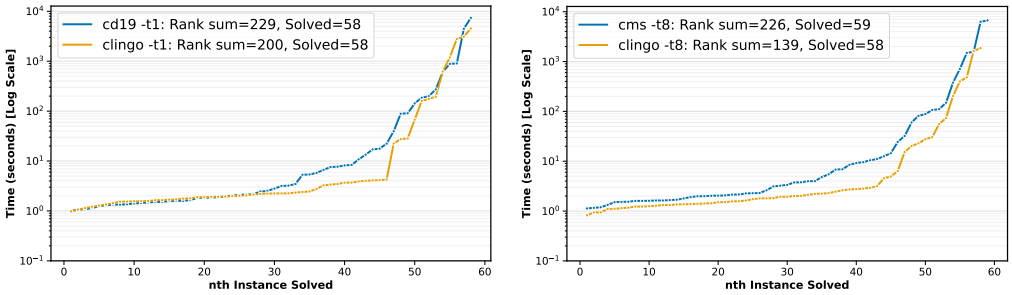



Figure 5.4: Cactus plots for Q-Synth qubit layout synthesis (69 instances, timeout = 10,800s). Left: single-threaded; Right: 8 threads. The $x$-axis shows the $n$-th fastest-solved instance per configuration (not necessarily the same instance); the $y$-axis shows cumulative runtime on a log scale. The legend reports *rank sum* (lower = better: sum of per-instance runtime ranks across all instances) and total instances solved.

PCLP rules, in contrast to the original encoding where the dependencies are implicit in the boolean formulas. The complex FOL constraints involve at most 12 atoms or 10 connectives and quantifiers in a single formula; writing the equivalent ASP by hand is possible, and would likely match the backend performance, but it requires the same auxiliary-predicate decomposition that SetLah! generates automatically.

**Setup.** We use Q-Synth's benchmark of 69 instances of varying difficulty drawn from standard quantum circuit families, and compare Q-Synth's original SAT-based backend against a replacement using SetLah!'s compiled ASP program with Clingo, in two matched settings:

- **Single-threaded:** `clingo -t1` vs. `cd19` (CaDiCaL 1.9.5, the fastest on this task in PySAT [83]).
- **Multi-threaded (8T):** `clingo -t8` vs. `cms -t8` (CryptoMiniSat 5.1.3 [151], a well-maintained parallel SAT solver with Python API support).

All configurations use the same incremental solving loop from Q-Synth; only the backend solver and encoding differ. The timeout is 10,800s per instance.

**Results and Analysis.** Figure 5.4 presents the cactus plots. We use *rank sum* among 4 solvers—the sum of each solver's per-instance runtime rank—as an aggregate measure of runtime consistency, complementing the solved-instance count. Note that the $n$-th point on each curve corresponds to that solver's $n$-th fastest instance, which need not be the same instance across solvers; thus a solver whose curve is uniformly lower is *overall* faster, not necessarily on every individual instance.

In the single-threaded setting, both `clingo -t1` and `cd19` solve 58 of 69 instances. `clingo -t1` achieves a rank sum of **200** vs. 229 for `cd19`, indicating overall faster runtimes. Given that CaDiCaL is a strong modern CDCL SAT solver (and the fastest among the solvers integrated in PySAT for incremental solving), matching it on solved instances while achieving a better rank sum is strong evidence of SetLah!'s competitiveness. On the first ∼25 easy instances both solvers stay in the 1–2s range. From instance 25 onward the curves separate: `clingo -t1` remains at ∼2–4s through instance 45, while `cd19` climbs to ∼8–20s. On the hardest instances (50+), both spike into the $10^3$–$10^4$s range and the gap narrows.

The multi-threaded comparison shows a stronger separation. `clingo -t8` achieves a rank sum of **139**—far below `cms -t8`'s 226—with the gap visible from the earliest instances onward. In the medium-difficulty range (instances 40–55), the gap widens to roughly an order of magnitude. `cms -t8` does solve one additional instance (59 vs. 58), but Clingo dominates on aggregate runtime. This demonstrates that Clingo's native incremental solving, made accessible through SetLah!'s solver-transparent translation, is highly effective on real-world iterative search problems.

Unlike the GL Square, Q-Synth is not one monolithic satisfiability problem. It solves a sequence of increasingly deep subproblems, most of which are UNSAT before the first valid layout is found. This structure favours Clingo's incremental grounding and solving, which can retain program structure across steps. Multi-threading yields less dramatic speedups than in GL Square because each incremental subproblem is smaller; the bottleneck is the number of depth steps, not only the size of one search instance.

### 5.5.4 Discussion

State-of-the-art SAT and ASP solvers are mature systems developed over decades, with CDCL-style search providing the core performance engine [71]. Performance among different solvers is highly instance-dependent, especially on satisfiable cases where heuristics play a decisive role [59, 82]. No single solver dominates all instances, so our evaluation should be read as evidence that SetLah! with Clingo is *competitive*, not universally superior. Beyond the performance results above, we highlight several broader insights emerging from our experiments.

**ASP as a viable backend.** Although less well-known than SAT in the Programming Languages community, ASP is a competitive alternative for combinatorial problems. Clingo's portfolio-based parallel solving [48, 70] and its native support for incremental solving make it one of the most capable maintained systems in this space—yet it has received limited attention outside the LP/KR communities (*e.g.*, [15, 166, 111] start to address the use of ASP in PL only from 2023), partly due to the difficulty of its input language. SetLah! lowers this barrier: its block structure decomposes problems into sub-problems, each modelled declaratively with guess-and-check via FOL constraints, letting non-experts automatically obtain efficient ASP encodings.

**Modularity and extensibility.** SetLah!'s rewriting pipeline is modular and orthogonal to existing optimisation techniques for logic programs: rule rewriting for better grounding [14, 46] or solving [162, 79] can be integrated into the translation or post-processing without changing the source language. Supporting alternative backends such as SAT or Datalog—when ASP's full expressiveness is not needed—is a promising direction for combining the strengths of different solvers.

**LLMs and SetLah!** One may wonder whether large language models (LLMs) make SetLah! unnecessary. In our experience, LLMs can generate

PySAT code for GL Squares but struggle with less common APIs (*e.g.*, CryptoMiniSat), and—more fundamentally—the resulting CNF encodings are hard to trust: like manual modelling (§ 5.2.1), the encoding may diverge from the problem statement, making verification difficult. Even as models improve, SetLah! can serve as a preferable generation target: its FOL constraints are closer to natural language, and its encodings are far smaller than equivalent SAT code, making them both cheaper to generate and easier to verify.

## 5.6 Related Work

### 5.6.1 Transformation of Formulas and Programs

The CNF transformations in SAT solving (*e.g.*, the Tseitin transformation [156]) convert arbitrary propositional formulas into the CNF format required by SAT solvers. Although the Tseitin transformation is linear in the size of its propositional input, obtaining that input by instantiating a quantified finite-domain specification can already be exponential in the size of the specification. For example, in preliminary experiments on Graeco–Latin squares, generating CNF by first grounding the complete FOL formula and then applying the Tseitin transformation took minutes already at $n = 6$. In contrast, the translation from SetLah! to ASP (§ 5.4) operates at a higher level of abstraction: each FOL formula is first rewritten into split ASP constraints to mitigate the grounding cost. The equality-saturation pass used to explore equivalent constraint shapes is implemented with Egglog [170]. Its use in automated reasoning is related to Lean-Egg [140], which integrates equality saturation with a theorem prover. In SetLah!, however, equality saturation is not itself the reasoning procedure: it is a compiler mechanism for selecting rewrites that reduce the cost of downstream grounding.

F2LP [101] also translates first-order formulas into logic programs, so quantifier elimination through auxiliary predicates is not by itself new. Its aim is to compute stable models of first-order formulas under the first-order

stable-model semantics. By contrast, SetLah! embeds classical FOL constraints in a modelling language with an explicit guess-and-check semantics and treats their translation as an optimisation pipeline: conjunction splitting is performed before QE so that the ASP grounder sees constraints with fewer variables. This ordering is the source of the grounding improvements discussed in § 5.5.2.

Closely related to our work, FOLASP [45] accepts the language of IDP [30] as input for ASP solvers, also sharing the motivation of bridging FOL and ASP. However, the language is designed upon inductive definitions [44], making the semantics less intuitive than SetLah!'s branching fixpoint semantics based on guess-and-check. Moreover, FOLASP's translation recursively introduces auxiliary predicates from the bottom up, without a preceding decomposition pass aimed at reducing grounding width. The resulting ASP programs can therefore ground poorly on larger domains, as also observed in its evaluation [45].

Alloy [84] and its Kodkod backend [155] form another close point of comparison. Like SetLah!, Alloy offers a relational, FOL-like notation over finite scopes. Kodkod then constructs a propositional encoding for a SAT solver and applies aggressive optimisations, including symmetry breaking. This mature tooling is especially effective on highly symmetric transition systems, as the birthday benchmark in § 5.5.2 shows. The trade-off is that its compilation is a one-shot, equisatisfiable reduction to a ground SAT instance. SetLah! instead produces a readable ASP program whose predicates and blocks remain available to multi-shot solving, model enumeration, and other ASP-level operations. Thus the comparison is not that one backend uniformly dominates the other, but that they preserve different structure and expose different solver capabilities.

### 5.6.2 Relevant Paradigms for Constraint Solving

As illustrated in § 5.2, the principal differences between ASP and SAT are: 1) the head-body structure of logic programs vs. the propositional implication, and 2) the first-order variables in ASP vs. the propositional

syntax of SAT. The former was first characterised by the loop formulas [108], which connect the stable model semantics to SAT solving, enabling DPLL-based SAT solvers [40] to be adapted for ASP [103, 148] and leading to the development of CDCL-based ASP solvers [69, 7].

EZASP [57] is closest to SetLah! in spirit: it also argues that many users do not need the full generality of stable-model semantics and therefore presents a stratified, easier-to-read fragment of ASP. SetLah! adopts the same broad lesson but makes two different choices. First, it removes negation from rule bodies and makes choice explicit; this rules out unstratified negation, and even negation-heavy encodings that are legal in EZASP, in exchange for a direct guess-and-check reading. Second, SetLah! sequences PCLP blocks, so choices in a later block may depend on atoms chosen in an earlier block. This dependency is needed in examples from Alloy and Q-SYNTH, where later search spaces are parameterised by earlier choices; a single EZASP stratum cannot express such choice-on-choice dependencies directly.

As discussed, SMT's quantifier handling [39] may not scale on harder combinatorial search problems in finite domains: quantifiers are processed lazily compared to ASP's eager grounding, since SMT is designed for expressive (possibly infinite) theories rather than finite-domain enumeration. Performance advantages of ASP over SMT have also been observed in Datalog synthesis [15, 134], where fixpoint-based reasoning is more natural. A common misconception is that SMT, being "more expressive" than SAT, should subsume it on finite problems—but *SMT is not first-order SAT*: a solver for problem class $A$ is not necessarily efficient on every subset of $A$. This motivates future work on detecting finite-domain SMT problems and translating them into SetLah! for efficient solving.

### 5.6.3 Higher-Order Satisfiability and Constraint Programming

Higher-order solving extends SAT and QBF by quantifying over Boolean functions, including functions that take functions as arguments. HOSAT is

TOWER-complete and can succinctly encode problems far beyond NP, with proposed applications including Presburger arithmetic, memory-consistency verification, multi-agent planning, and secure synthesis [33, 157]; HOMC-SAT provides the first solver by reducing such formulas to higher-order model checking [157]. This is outside the expressiveness of both PCLP and the normal ASP fragment targeted by SetLah!, which capture NP: choice rules select relations, but predicates and functions are not themselves values that can be quantified or passed as arguments. Quantified ASP provides a nearer intermediate step. Quantified Answer Set Programming (QASP) adds existential and universal quantifiers over atoms and has been applied to conformant and conditional planning [56]; ASP with Quantifiers (ASP(Q)) instead quantifies over the answer sets of program blocks, extending normal ASP across the polynomial hierarchy [9]. The related stable–unstable semantics makes one ASP search an explicit oracle for another, and has been implemented using ASPTOOLS with Clingo as the backend [86]. These approaches quantify over answer sets rather than higher-order functions, but their block structure is close to SetLah!'s. Extending SetLah! with answer-set quantifiers could therefore provide a modelling layer for QASP, and offers a promising first step towards investigating whether ASP-based modelling can also bridge to genuinely higher-order solving.

The idea of branching fixpoint semantics is closely related to the semantics of Dusa [111], which is also a choice-based language that can mimic stable model semantics. However, Dusa uses its own backtracking-based evaluation rather than leveraging existing ASP systems, and supports restricted constraints, limiting its capability on modelling and solving general combinatorial problems. In this work, we use SetLah! to expose the power of ASP solvers for combinatorial search, but we have neither illustrated every practical advantage of ASP over SAT nor supported every feature of ASP in SetLah!. For example, ASP solvers natively support enumerating all stable models (*e.g.*, 92 solutions for the 8-queens problem), which is rarely available in SAT/SMT solvers [90]—manually banning previously found models introduces significant overhead in both modelling and solving [20]. Features like brave/cautious reasoning [8] and projection [68] are also natu-

rally supported by the compiled ASP programs. Other useful features such as weak constraints [27] and heuristics [67] require language-level extensions to SetLah!, to be investigated in future work.

The other line of related work concerns constraint programming (CP) languages. Traditional CP systems are rooted in Prolog-style top-down unification and backtracking, whose scalability limitations compared with SAT-based methods are well discussed [22]. More recent systems combine high-level models with modern backends. ENFRAGMO [2] grounds extended FOL to SAT, while PICAT [171] combines logic programming with dedicated constraint solvers. These systems usually separate the source model from a solver-specific, instance-level representation. In contrast, SetLah! compiles to a reusable ASP program, preserving enough predicate structure for users to inspect the output and employ ASP facilities such as multi-shot solving and projection. For arithmetic-heavy finite-domain problems, however, classical CLP(FD) [47] and modern CP systems [129] offer native propagators and may be a better fit than SAT- or ASP-based methods.

Among CP tools, MINIZINC [120] and CONJURE [3] share SetLah!'s motivation of providing a high-level declarative modelling language. MINIZINC standardises a solver-independent model that is flattened for a selected backend, whereas CONJURE refines abstract ESSENCE specifications [61] into one or more concrete constraint models. These languages, as well as set-oriented CLP systems such as {LOG} [49], support sets as first-class modelling objects. They therefore provide substantially broader support for arithmetic, sets, global constraints, and representation selection than SetLah!. However, their language designs differ fundamentally. CP languages are primarily *algebraic*, whereas SetLah! is intentionally *relational*: its kernel consists of positive rules, explicit choices, and FOL constraints. This narrower design preserves the head–body and predicate structure of the generated ASP and makes staged choices natural, but it does not replace CP's specialised numerical and global constraints. This difference is why § 5.5 compares SetLah! primarily with SAT-based relational tools rather than claiming general superiority over CP. Similar to SMT vs. SAT, for combinatorial problems naturally modelled in pure ASP, native ASP

solvers outperform MiniZinc approaches in the benchmark of [139, 53]. Conversely, some problems admit very compact CP models because of global constraints: for instance, Graeco–Latin squares can be modelled with `all-different` constraints and specialised symmetry breaking in CP/IP formulations [142]. Such comparisons are therefore modelling-paradigm dependent; a compact CP model for N-queens or Latin squares does not imply the same advantage for relational encodings such as logic-gate circuits or the staged Q-Synth layout constraints.

# Chapter 6

# Discussion and Concluding Remarks

> *Historians of science have often noted that at any given time scholars in a particular field tend to share basic assumptions about their subject.*
>
> —Daniel Kahneman, *Thinking, Fast and Slow* [88]

## 6.1 The Positions of the Thesis

### 6.1.1 The Impact of This Thesis for PL Research

Program synthesis has long been regarded as a central challenge in computer science [75], motivated by the ambition of "automatic programming". Closely related areas—including invariant inference, specification mining, and inductive logic programming—share a common concern: how to search large spaces of candidate programs or specifications while exploiting domain-specific structure. For example, the partial-order-based pruning used in DistAI [169] is closely related in spirit to pruning techniques used in Popper [37]. From this perspective, the techniques developed in Chapter 3 and Chapter 4 are not limited to their immediate domains, but point to reusable ways of organising and pruning inductive search spaces.

The growing interest in specification synthesis is both a response to need and a shift in emphasis. Formal methods are increasingly relevant to software development, which increases the demand for formal specifications; at the

same time, large language models (LLMs) have changed expectations about what can be generated from natural-language intent. This does not remove the need for inductive generalisation. LLMs still struggle with general inductive reasoning [107] and with program-by-example tasks that require systematic generalisation [104, 161, 89]. A plausible long-term direction is therefore not to replace inductive synthesis, but to make synthesis engines easier to construct, configure, and combine. Existing pruning techniques for synthesis can then become part of a library for building domain-specific inductive synthesisers.

Another position of this thesis is that solver choice should be treated as a modelling and engineering decision, not merely as a question of expressiveness. SMT solvers such as Z3 [41] and CVC5 [13] are mature and highly effective for many applications. However, the fact that SMT is more expressive than SAT does not imply that SMT solvers dominate SAT- or ASP-based encodings on every finite-domain problem: a solver for a broader class need not be the most efficient engine for every subclass. This thesis therefore argues for a more deliberate use of solver backends, especially when a problem is finite, highly combinatorial, and naturally expressible as structured search. A natural future experiment is to revisit existing SMT-based encodings with SetLah! as an ASP-oriented modelling layer and measure when this change of backend is beneficial.

### 6.1.2 The Impact of This Thesis in the LLM Era

The broader research landscape is changing rapidly. Even communities that have traditionally been cautious about machine-learning-based tools, including parts of the PL community, now increasingly consider how LLMs change programming, verification, and tool use. It is therefore useful to ask what role PL research should play in this setting.

SetLah! can be viewed as one such contribution. Beyond improving the modelling experience for constraint solving, it suggests a useful middle ground between LLMs and PL: specialised languages can act as precise interfaces between natural-language intent and reliable solver-backed

computation. As LLMs become increasingly capable at general-purpose programming, specialised languages remain valuable because they expose domain structure that a general-purpose language hides. In this sense, LLM-assisted programming may become increasingly *higher-order*: when a model cannot solve a problem directly, it can select or generate a program in a specialised language whose semantics and backend solver provide the required search or reasoning capability.

Another important agenda is to understand why LLMs are effective on some programming tasks but unreliable on others. In particular, LLMs are strong at generating code from natural-language descriptions, a setting that differs from classical program-by-example synthesis. One possible explanation is that many such tasks are closer to guided deduction from an intended behaviour than to open-ended inductive generalisation. LLMs can make local implementation progress quickly and iterate, but they still benefit from external tools that enforce semantic constraints. This reinforces the role of formal languages and solvers as reliable interfaces around LLM-generated artefacts.

## 6.2 Future Directions

### 6.2.1 Combining ASP with E-graphs

Similar to how Datalog is combined with e-graphs in Egglog [170], the constraint-solving power of ASP can be combined with the efficient equality reasoning of e-graphs. For example, our initial experiments on e-graph extraction [74, 28] show promising performance from ASP-based extraction. Further directions include: (1) pruning equality saturation in e-graphs with ASP-based constraints; and (2) using e-graphs as a compact internal representation for the search space of ASP-based synthesis.

### 6.2.2 Closing the Loop: From Synthesis to Verification

Currently, the outputs of Sippy and FORCE rely on external checking and verification—Sippy tests candidate SL predicates on concrete memory graphs via an external Prolog-based evaluator before passing the synthesised predicates to downstream SL verifiers [85, 132] or deductive synthesisers such as SuSLik [133], and FORCE hands its synthesised FO invariants off to Ivy [125] or Veil [130]. An important engineering direction is to tighten this integration: incorporating the verifier's feedback directly into the synthesis loop, so that a failed verification attempt guides the next round of candidate generation. This would transform the current *generate-then-verify* pipeline into a fully automated *guess-check-revise* loop, similar in spirit to CEGIS [150].

### 6.2.3 Synthesising Inductive Synthesisers

As argued in § 6.1, I believe the long-term future of inductive generalisation is *meta*: rather than building one carefully hand-crafted synthesiser per domain, the goal should be to synthesise the synthesiser itself from high-level specifications of the target hypothesis class and the pruning constraints. The ASP-based architecture of Sippy and FORCE is well-positioned for this: the ASP encoding *is* the synthesiser, and SetLah! provides a higher-level language in which to write it. A fully realised vision would allow a practitioner to specify a new synthesis domain in SetLah!—declaring the hypothesis grammar, the consistency constraints, and the pruning heuristics—and have SetLah! compile this specification into a ready-to-run synthesis engine backed by ASP solvers.

### 6.2.4 SetLah! as a Platform and an Interface for LLMs

The current SetLah! prototype establishes the semantic foundation—constructive fixpoint semantics and compilation to ASP—but the language

itself has significant room to grow. On the language side, a *static type system* would catch the most common modelling errors (e.g., sort mismatches in quantified formulas) before compilation; extending the backend to emit SMT-LIB for Z3 [41] or QBF for quantified problems would further broaden the class of problems SetLah! can address without changing its surface syntax. Most directly motivated by this thesis, re-encoding the synthesis problems from Chapter 3 and Chapter 4 *in* SetLah! would close the recursive loop explicitly. On the LLM side, SetLah!'s structured, block-based syntax and its explicit separation of the *what* (FOL constraints) from the *how* (ASP compilation) make it a natural target for LLM-assisted specification [110, 164]: LLMs already exhibit reasonable competence at generating first-order logical constraints, but struggle to produce correct raw ASP directly. SetLah! can therefore serve not just as a tool for human modellers, but as a well-typed interface through which LLMs safely access the power of combinatorial solvers—exactly the mid-ground between LLM and PL argued for in § 6.1.

## 6.3 Epilogue

This thesis has argued that Answer Set Programming is both a practical response to the Specification Bottleneck and, self-referentially, a compelling instance of it. Through Sippy and FORCE, the thesis shows that inductive search for Separation Logic predicates and First-Order Logic invariants can be expressed declaratively and delegated to ASP solvers, yielding complete and efficient synthesis from positive examples alone. Through SetLah!, it then turns the same argument inward: the ASP encodings that make such tools possible are themselves specifications, and therefore deserve language support, semantic foundations, and correct compilation. Taken together, these systems suggest a unified view of logic-based tool building: expose the structure of a search problem at the right level of abstraction, and let a solver carry out the combinatorial work. The future directions above point to several ways in which this view can be extended, from tighter synthesis–

verification loops to solver-backed interfaces for LLMs and higher-level languages for synthesising synthesisers themselves.

Considering the fact that logic programming is rooted in AI, it is undoubtedly ironic that the most recent wave of AI research—the LLM revolution—has largely ignored logic programming. This thesis instead uses ASP, a relatively popular logic programming paradigm, as a formal method. Can formal methods complement AI, compete with AI, or be replaced by AI? The future is not to be predicted, but to be shaped. The way this thesis potentially shapes the future is by showing the way formal methods can be used in the tasks where LLMs are currently not effective. In maybe at most 5 years, we will see whether the ongoing growth of LLMs will make formal methods obsolete, or there is a bottleneck in LLMs that formal methods can address. What is the next evolution in computer science?

# Bibliography

[1] Amir Aavani. *Enfragmo: A system for grounding extended first-order logic to SAT*. PhD thesis, Simon Fraser University, 2014.

[2] Amir Aavani, Xiongnan (Newman) Wu, Shahab Tasharrofi, Eugenia Ternovska, and David G. Mitchell. Enfragmo: A system for modelling and solving search problems with logic. In *LPAR*, volume 7180 of *LNCS*, pages 15–22. Springer, 2012. doi:10.1007/978-3-642-28717-6_4.

[3] Özgür Akgün, Alan M. Frisch, Ian P. Gent, Christopher Jefferson, Ian Miguel, and Peter Nightingale. Conjure: Automatic generation of constraint models from problem specifications. *Artif. Intell.*, 310:103751, 2022. URL: https://doi.org/10.1016/j.artint.2022.103751, doi:10.1016/J.ARTINT.2022.103751.

[4] Aws Albarghouthi, Isil Dillig, and Arie Gurfinkel. Maximal Specification Synthesis. In *POPL*, pages 789–801. ACM, 2016. doi:10.1145/2837614.2837628.

[5] Aws Albarghouthi, Sumit Gulwani, and Zachary Kincaid. Recursive Program Synthesis. In *CAV*, volume 8044 of *LNCS*, pages 934–950. Springer, 2013. doi:10.1007/978-3-642-39799-8_67.

[6] Rajeev Alur, Rastislav Bodík, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. Syntax-guided synthesis. In *FMCAD*, pages 1–8. IEEE, 2013. doi:10.1109/FMCAD.2013.6679385.

[7] Mario Alviano, Carmine Dodaro, Nicola Leone, and Francesco Ricca. Advances in WASP. In *Logic Programming and Nonmonotonic Rea-*

*soning - 13th International Conference, LPNMR 2015, Lexington, KY, USA, September 27-30, 2015. Proceedings*, volume 9345 of *LNCS*, pages 40–54. Springer, 2015. doi:10.1007/978-3-319-23264-5_5.

[8] Mario Alviano, Carmine Dodaro, and Francesco Ricca. Anytime computation of cautious consequences in answer set programming. *Theory Pract. Log. Program.*, 14(4-5):755–770, 2014. doi:10.1017/S1471068414000325.

[9] Giovanni Amendola, Francesco Ricca, and Miroslaw Truszczynski. Beyond NP: Quantifying over answer sets. *Theory Pract. Log. Program.*, 19(5-6):705–721, 2019. doi:10.1017/S1471068419000140.

[10] Dana Angluin. Inductive inference of formal languages from positive data. *Information and Control*, 45(2):117–135, 1980. doi:10.1016/S0019-9958(80)90285-5.

[11] Andrew W. Appel, Robert Dockins, Aquinas Hobor, Lennart Beringer, Josiah Dodds, Gordon Stewart, Sandrine Blazy, and Xavier Leroy. *Program Logics for Certified Compilers*. Cambridge University Press, 2014. doi:10.1017/CBO9781107256552.

[12] Angello Astorga, Shambwaditya Saha, Ahmad Dinkins, Felicia Wang, P. Madhusudan, and Tao Xie. Synthesizing contracts correct modulo a test generator. *Proc. ACM Program. Lang.*, 5(OOPSLA):1–27, 2021. doi:10.1145/3485481.

[13] Haniel Barbosa, Clark W. Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. cvc5: A versatile and industrial-strength SMT solver. In *TACAS*, volume 13243 of *LNCS*, pages 415–442. Springer, 2022. doi:10.1007/978-3-030-99524-9_24.

[14] Alexander Beiser, Markus Hecher, Kaan Unalan, and Stefan Woltran. Bypassing the ASP bottleneck: Hybrid grounding by splitting and

rewriting. In *IJCAI*, pages 3250–3258. IJCAI, 2024. doi:10.24963/ijcai.2024/360.

[15] Aaron Bembenek, Michael Greenberg, and Stephen Chong. From SMT to ASP: solver-based approaches to solving datalog synthesis-as-rule-selection problems. *Proc. ACM Program. Lang.*, 7(POPL):185–217, 2023. doi:10.1145/3571200.

[16] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. Symbolic execution with separation logic. In *APLAS*, volume 3780 of *LNCS*, pages 52–68. Springer, 2005. doi:10.1007/11575467_5.

[17] Armin Biere, Tobias Faller, Katalin Fazekas, Mathias Fleury, Nils Froleyks, and Florian Pollitt. CaDiCaL 2.0. In *CAV*, volume 14681 of *LNCS*, pages 133–152. Springer, 2024. doi:10.1007/978-3-031-65627-9_7.

[18] Armin Biere, Mathias Fleury, Nils Froleyks, and Marijn J. H. Heule. The SAT museum. In *Pragmatics of SAT*, volume 3545 of *CEUR Workshop Proceedings*, pages 72–87. CEUR-WS.org, 2023. URL: https://ceur-ws.org/Vol-3545/paper6.pdf.

[19] Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors. *Handbook of Satisfiability - Second Edition*, volume 336 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2021. doi:10.3233/FAIA336.

[20] Nikolaj S. Bjørner, Clemens Eisenhofer, and Laura Kovács. User-propagation for custom theories in smt solving. In *SMT*, volume 3185 of *CEUR Workshop Proceedings*, pages 71–79. CEUR-WS.org, 2022. URL: https://ceur-ws.org/Vol-3185/extended6630.pdf.

[21] Jan H. Boockmann and Gerald Luettgen. Learning data structure shapes from memory graphs. In *LPAR*, volume 73 of *EPiC Series in Computing*, pages 151–168. EasyChair, 2020. doi:10.29007/dhpw.

[22] Lucas Bordeaux, Youssef Hamadi, and Lintao Zhang. Propositional satisfiability and constraint programming: A comparative survey. *ACM Comput. Surv.*, 38(4):12, 2006. doi:10.1145/1177352.1177354.

[23] Aaron R. Bradley. SAT-Based Model Checking without Unrolling. In *VMCAI*, volume 6538 of *LNCS*, pages 70–87. Springer, 2011. doi:10.1007/978-3-642-18275-4_7.

[24] Gerhard Brewka, Thomas Eiter, and Miroslaw Truszczynski. Answer set programming at a glance. *Commun. ACM*, 54(12):92–103, 2011. doi:10.1145/2043174.2043195.

[25] Curtis Bright, Jürgen Gerhard, Ilias S. Kotsireas, and Vijay Ganesh. Effective problem solving using SAT solvers. In *Maple Conference*, volume 1125 of *Communications in Computer and Information Science*, pages 205–219. Springer, 2019. doi:10.1007/978-3-030-41258-6_15.

[26] Marc Brockschmidt, Yuxin Chen, Pushmeet Kohli, Siddharth Krishna, and Daniel Tarlow. Learning shape analysis. In *SAS*, volume 10422 of *LNCS*, pages 66–87. Springer, 2017. doi:10.1007/978-3-319-66706-5_4.

[27] Francesco Buccafurri, Nicola Leone, and Pasquale Rullo. Enhancing disjunctive datalog by constraints. *IEEE Trans. Knowl. Data Eng.*, 12(5):845–860, 2000. doi:10.1109/69.877512.

[28] Yaohui Cai, Kaixin Yang, Chenhui Deng, Cunxi Yu, and Zhiru Zhang. SmoothE: Differentiable E-Graph extraction. In Lieven Eeckhout, Georgios Smaragdakis, Kaitai Liang, Adrian Sampson, Martha A. Kim, and Christopher J. Rossbach, editors, *ASPLOS*, pages 1020–1034. ACM, 2025. doi:10.1145/3669940.3707262.

[29] Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. Compositional shape analysis by means of bi-abduction. *J. ACM*, 58(6):26:1–26:66, 2011. doi:10.1145/2049697.2049700.

[30] Broes De Cat, Bart Bogaerts, Maurice Bruynooghe, Gerda Janssens, and Marc Denecker. Predicate logic as a modeling language: the IDP system. In *Declarative Logic Programming: Theory, Systems, and Applications*, volume 20 of *ACM Books*, pages 279–323. ACM / Morgan & Claypool, 2018. doi:10.1145/3191315.3191321.

[31] Hubie Chen and Stefan Mengel. Optimally rewriting formulas and database queries: A confluence of term rewriting, structural decomposition, and complexity. In *ICDT*, volume 290 of *LIPIcs*, pages 16:1–16:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2024. URL: https://doi.org/10.4230/LIPIcs.ICDT.2024.16, doi:10.4230/LIPICS.ICDT.2024.16.

[32] Ahlem Ben Cherifa and Pierre Lescanne. Termination of rewriting systems by polynomial interpretations and its implementation. *Science of Computer Programming*, 9(2):137–159, 1987. doi:10.1016/0167-6423(87)90030-X.

[33] Dmitry Chistikov, Christoph Haase, Zahra Hadizadeh, and Alessio Mansutti. Higher-order quantified boolean satisfiability. In *MFCS*, volume 241 of *LIPIcs*, pages 33:1–33:15. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPIcs.MFCS.2022.33.

[34] Coq Development Team. *The Coq Proof Assistant Reference Manual - Version 8.16*, 2023. Available at http://coq.inria.fr/.

[35] Andrew Cropper. Learning logic programs through divide, constrain, and conquer. In *AAAI*, pages 6446–6453. AAAI Press, 2022. doi:10.1609/AAAI.V36I6.20596.

[36] Andrew Cropper, Sebastijan Dumancic, Richard Evans, and Stephen H. Muggleton. Inductive logic programming at 30. *Mach. Learn.*, 111(1):147–172, 2022. doi:10.1007/s10994-021-06089-1.

[37] Andrew Cropper and Rolf Morel. Learning programs by learning from failures. *Machine Learning*, 110(4):801–856, 2021. doi:10.1007/S10994-020-05934-Z.

[38] Andrew Cropper and Stephen H. Muggleton. Logical Minimisation of Meta-Rules Within Meta-Interpretive Learning. In *ILP*, volume 9046 of *LNCS*, pages 62–75. Springer, 2014. doi:10.1007/978-3-319-23708-4_5.

[39] Marek Danco, Petra Hozzová, and Mikolás Janota. From MBQI to enumerative instantiation and back. *CoRR*, abs/2506.22584, 2025. URL: https://doi.org/10.48550/arXiv.2506.22584, doi:10.48550/ARXIV.2506.22584.

[40] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *J. ACM*, 7(3):201–215, 1960. doi:10.1145/321033.321034.

[41] Leonardo Mendonça de Moura and Nikolaj S. Bjørner. Z3: an efficient SMT solver. In *TACAS*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008. doi:10.1007/978-3-540-78800-3_24.

[42] Leonardo Mendonça de Moura and Nikolaj S. Bjørner. Satisfiability modulo theories: introduction and applications. *Commun. ACM*, 54(9):69–77, 2011. doi:10.1145/1995376.1995394.

[43] Leonardo Mendonça de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. The lean theorem prover (system description). In *CADE*, volume 9195 of *LNCS*, pages 378–388. Springer, 2015. doi:10.1007/978-3-319-21401-6_26.

[44] Marc Denecker and Eugenia Ternovska. A logic of nonmonotone inductive definitions. *ACM Trans. Comput. Log.*, 9(2):14:1–14:52, 2008. doi:10.1145/1342991.1342998.

[45] Kylian Van Dessel, Jo Devriendt, and Joost Vennekens. FOLASP: FO(.) as input language for Answer Set solvers. *CoRR*, abs/2108.04020, 2021. doi:10.48550/arXiv.2108.04020.

[46] Carmine Dodaro, Giuseppe Mazzotta, and Francesco Ricca. Blending grounding and compilation for efficient ASP solving. In *KR*, 2024.

URL: https://doi.org/10.24963/kr.2024/30, doi:10.24963/KR.2024/30.

[47] Agostino Dovier, Andrea Formisano, and Enrico Pontelli. A comparison of CLP(FD) and ASP solutions to np-complete problems. In Maurizio Gabbrielli and Gopal Gupta, editors, *Logic Programming, 21st International Conference, ICLP 2005, Sitges, Spain, October 2-5, 2005, Proceedings*, volume 3668 of *Lecture Notes in Computer Science*, pages 67–82. Springer, 2005. doi:10.1007/11562931_8.

[48] Agostino Dovier, Andrea Formisano, and Enrico Pontelli. Parallel answer set programming. In *Handbook of Parallel Constraint Reasoning*, pages 237–282. Springer, 2018. doi:10.1007/978-3-319-63516-3_7.

[49] Agostino Dovier, Carla Piazza, Enrico Pontelli, and Gianfranco Rossi. Sets and constraint logic programming. *ACM Trans. Program. Lang. Syst.*, 22(5):861–931, 2000. doi:10.1145/365151.365169.

[50] Gregory J. Duck, Joxan Jaffar, and Roland H. C. Yap. Shape neutral analysis of graph-based data-structures. *Theory Pract. Log. Program.*, 18(3-4):470–483, 2018. doi:10.1017/S147106841800025X.

[51] Niklas Eén, Alan Mishchenko, and Robert K. Brayton. Efficient implementation of property directed reachability. In *FMCAD*, pages 125–134. FMCAD Inc., 2011. URL: http://dl.acm.org/citation.cfm?id=2157675.

[52] Niklas Eén and Niklas Sörensson. An extensible sat-solver. In *SAT*, volume 2919 of *LNCS*, pages 502–518. Springer, 2003. doi:10.1007/978-3-540-24605-3_37.

[53] Thomas Eiter, Tobias Geibinger, Nysret Musliu, Johannes Oetsch, and Tobias Kaminski. ASP-FZN: A translation-based constraint answer set solver. *Theory Pract. Log. Program.*, 25(4):649–667, 2025. URL: https://doi.org/10.1017/s1471068425100264, doi:10.1017/S1471068425100264.

[54] Thomas Eiter, Georg Gottlob, and Heikki Mannila. Disjunctive datalog. *ACM Trans. Database Syst.*, 22(3):364–418, 1997. doi:10.1145/261124.261126.

[55] Moshe Emmer, Zurab Khasidashvili, Konstantin Korovin, and Andrei Voronkov. Encoding industrial hardware verification problems into effectively propositional logic. In *FMCAD*, pages 137–144. IEEE, 2010. URL: https://ieeexplore.ieee.org/document/5770942/.

[56] Jorge Fandinno, François Laferrière, Javier Romero, Torsten Schaub, and Tran Cao Son. Planning with incomplete information in quantified answer set programming. *Theory Pract. Log. Program.*, 21(5):663–679, 2021. doi:10.1017/S1471068421000259.

[57] Jorge Fandinno, Seemran Mishra, Javier Romero, and Torsten Schaub. Answer set programming made easy. In *Essays Dedicated to Manuel Hermenegildo*, volume 13160 of *LNCS*, pages 133–150. Springer, 2023. doi:10.1007/978-3-031-31476-6_7.

[58] Cormac Flanagan and K. Rustan M. Leino. Houdini, an Annotation Assistant for ESC/Java. In *FME*, volume 2021 of *LNCS*, pages 500–517. Springer, 2001. doi:10.1007/3-540-45251-6_29.

[59] Mathias Fleury and Daniela Kaufmann. Life span of SAT techniques. *CoRR*, abs/2402.01202, 2024. doi:10.48550/ARXIV.2402.01202.

[60] Eden Frenkel, Tej Chajed, Oded Padon, and Sharon Shoham. Efficient implementation of an abstract domain of quantified first-order formulas. *CoRR*, abs/2405.10308, 2024. doi:10.48550/ARXIV.2405.10308.

[61] Alan M. Frisch, Warwick Harvey, Christopher Jefferson, Bernadette Martínez Hernández, and Ian Miguel. ESSENCE: A constraint language for specifying combinatorial problems. *Constraints*, 13(3):268–306, 2008. doi:10.1007/s10601-008-9047-y.

[62] Yeting Ge and Leonardo Mendonça de Moura. Complete instantiation for quantified formulas in satisfiability modulo theories. In *CAV*,

volume 5643 of *LNCS*, pages 306–320. Springer, 2009. doi:10.1007/978-3-642-02658-4_25.

[63] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. *Answer Set Solving in Practice*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, 2012. doi:10.2200/S00457ED1V01Y201211AIM019.

[64] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. *Answer Set Solving in Practice*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, 2012. doi:10.2200/S00457ED1V01Y201211AIM019.

[65] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. Clingo = ASP + Control: Preliminary Report. *CoRR*, abs/1405.3694, 2014. URL: http://arxiv.org/abs/1405.3694.

[66] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. Multi-shot ASP solving with clingo. *Theory Pract. Log. Program.*, 19(1):27–82, 2019. doi:10.1017/S1471068418000054.

[67] Martin Gebser, Benjamin Kaufmann, Javier Romero, Ramón Otero, Torsten Schaub, and Philipp Wanko. Domain-specific heuristics in answer set programming. In *AAAI*, pages 350–356. AAAI Press, 2013. URL: https://doi.org/10.1609/aaai.v27i1.8585, doi:10.1609/AAAI.V27I1.8585.

[68] Martin Gebser, Benjamin Kaufmann, and Torsten Schaub. Solution enumeration for projected boolean search problems. In *CPAIOR*, volume 5547 of *LNCS*, pages 71–86. Springer, 2009. doi:10.1007/978-3-642-01929-6_7.

[69] Martin Gebser, Benjamin Kaufmann, and Torsten Schaub. Conflict-driven answer set solving: From theory to practice. *Artif. Intell.*, 187:52–89, 2012. URL: https://doi.org/10.1016/j.artint.2012.04.001, doi:10.1016/J.ARTINT.2012.04.001.

[70] Martin Gebser, Benjamin Kaufmann, and Torsten Schaub. Multi-threaded ASP solving with clasp. *Theory Pract. Log. Program.*, 12(4-5):525–545, 2012. doi:10.1017/S1471068412000166.

[71] Martin Gebser, Nicola Leone, Marco Maratea, Simona Perri, Francesco Ricca, and Torsten Schaub. Evaluation techniques and systems for answer set programming: a survey. In *IJCAI*, pages 5450–5456. ijcai.org, 2018. URL: https://doi.org/10.24963/ijcai.2018/769, doi:10.24963/IJCAI.2018/769.

[72] Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In *ICLP*, pages 1070–1080. MIT Press, 1988.

[73] Aman Goel and Karem A. Sakallah. On symmetry and quantification: A new approach to verify distributed protocols. In *NASA Formal Methods*, volume 12673 of *LNCS*. Springer, 2021. doi:10.1007/978-3-030-76384-8_9.

[74] Amir Kafshdar Goharshady, Chun Kit Lam, and Lionel Parreaux. Fast and optimal extraction for sparse equality graphs. *Proc. ACM Program. Lang.*, 8(OOPSLA2):2551–2577, 2024. doi:10.1145/3689801.

[75] Sumit Gulwani, Oleksandr Polozov, and Rishabh Singh. Program synthesis. *Found. Trends Program. Lang.*, 4(1-2):1–119, 2017. doi:10.1561/2500000010.

[76] Bolei Guo, Neil Vachharajani, and David I. August. Shape analysis with inductive recursion synthesis. In *PLDI*, pages 256–265. ACM, 2007. doi:10.1145/1250734.1250764.

[77] Peter L. Hammer and Alexander Kogan. Optimal compression of propositional horn knowledge bases: Complexity and approximation. *Artif. Intell.*, 64(1):131–145, 1993. doi:10.1016/0004-3702(93)90062-G.

[78] Travis Hance, Marijn Heule, Ruben Martins, and Bryan Parno. Finding Invariants of Distributed Systems: It's a Small (Enough)

World After All. In *NSDI*, pages 115–131. USENIX Association, 2021. URL: https://www.usenix.org/conference/nsdi21/presentation/hance.

[79] Philipp Hanisch and Markus Krötzsch. Rule rewriting revisited: A fresh look at static filtering for datalog and ASP. *CoRR*, abs/2601.05108, 2026. doi:10.48550/ARXIV.2601.05108.

[80] Patrick J Hayes. The frame problem and related problems in artificial intelligence. In *Readings in artificial intelligence*, pages 223–230. Elsevier, 1981.

[81] Werner Heisenberg. *Physics and Philosophy: The Revolution in Modern Science*. Harper, New York, 1958.

[82] Marijn Heule and Torsten Schaub. What's hot in the SAT and ASP competitions. In *AAAI*, pages 4322–4323. AAAI Press, 2015. doi:10.1609/AAAI.V29I1.9348.

[83] Alexey Ignatiev, Antonio Morgado, and Joao Marques-Silva. PySAT: A Python toolkit for prototyping with SAT oracles. In *SAT*, pages 428–437, 2018. doi:10.1007/978-3-319-94144-8_26.

[84] Daniel Jackson. Alloy: a lightweight object modelling notation. *ACM Trans. Softw. Eng. Methodol.*, 11(2):256–290, 2002. doi:10.1145/505145.505149.

[85] Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java. In *NASA Formal Methods*, volume 6617 of *LNCS*, pages 41–55. Springer, 2011. doi:10.1007/978-3-642-20398-5_4.

[86] Tomi Janhunen. Implementing stable-unstable semantics with ASP-TOOLS and Clingo. In *PADL*, volume 13165 of *LNCS*, pages 135–153. Springer, 2022. doi:10.1007/978-3-030-94479-7_9.

[87] Ruyi Ji, Jingtao Xia, Yingfei Xiong, and Zhenjiang Hu. Generalizable synthesis through unification. *Proc. ACM Program. Lang.*, 5(OOPSLA):1–28, 2021. doi:10.1145/3485544.

[88] Daniel Kahneman. *Thinking, Fast and Slow.* Farrar, Straus and Giroux, New York, 2011.

[89] Ruhma Khan, Sumit Gulwani, Vu Le, Arjun Radhakrishna, Ashish Tiwari, and Gust Verbruggen. Llm-guided compositional program synthesis. *CoRR*, abs/2503.15540, 2025. doi:10.48550/ARXIV.2503.15540.

[90] Sarfraz Khurshid, Darko Marinov, Ilya Shlyakhter, and Daniel Jackson. A case for efficient solution enumeration. In *SAT*, volume 2919 of *LNCS*, pages 272–286. Springer, 2003. doi:10.1007/978-3-540-24605-3_21.

[91] Dominic Klyve and Lee Stemkoski. Graeco-Latin squares and a mistaken conjecture of Euler. *The College Mathematics Journal*, 37(1):2–15, 2006.

[92] Jason R. Koenig, Oded Padon, Neil Immerman, and Alex Aiken. First-order quantified separators. In *PLDI*, pages 703–717. ACM, 2020. doi:10.1145/3385412.3386018.

[93] Jason R. Koenig, Oded Padon, Sharon Shoham, and Alex Aiken. Inferring invariants with quantifier alternations: Taming the search space explosion. In *TACAS*, volume 13243 of *LNCS*, pages 338–356. Springer, 2022. doi:10.1007/978-3-030-99524-9_18.

[94] Tuukka Korhonen, Jeremias Berg, and Matti Järvisalo. Enumerating potential maximal cliques via SAT and ASP. In *IJCAI*, pages 1116–1122. ijcai.org, 2019. URL: https://doi.org/10.24963/ijcai.2019/156, doi:10.24963/IJCAI.2019/156.

[95] Robert A. Kowalski. Predicate logic as programming language. In *Information Processing, Proceedings of the 6th IFIP Congress 1974,*

*Stockholm, Sweden, August 5-10, 1974*, pages 569–574. North-Holland, 1974.

[96] Elias Kuiter, Sebastian Krieter, Chico Sundermann, Thomas Thüm, and Gunter Saake. Tseitin or not Tseitin? The impact of CNF transformations on feature-model analyses. In *37th IEEE/ACM International Conference on Automated Software Engineering, ASE 2022, Rochester, MI, USA, October 10-14, 2022*, pages 110:1–110:13. ACM, 2022. doi:10.1145/3551349.3556938.

[97] Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, 1998. doi:10.1145/279227.279229.

[98] Steven E. Landsburg. *The Armchair Economist: Economics and Everyday Life*. Free Press, New York, 1993.

[99] Quang Loc Le, Azalea Raad, Jules Villard, Josh Berdine, Derek Dreyer, and Peter W. O'Hearn. Finding real bugs in big programs with incorrectness logic. *Proc. ACM Program. Lang.*, 6(OOPSLA1):1–27, 2022. doi:10.1145/3527325.

[100] Ton Chanh Le, Guolong Zheng, and ThanhVu Nguyen. SLING: using dynamic analysis to infer program invariants in separation logic. In *PLDI*, pages 788–801. ACM, 2019. doi:10.1145/3314221.3314634.

[101] Joohyung Lee and Ravi Palla. System f2lp - computing answer sets of first-order formulas. In *LPNMR*, volume 5753 of *LNCS*, pages 515–521. Springer, 2009. doi:10.1007/978-3-642-04238-6_51.

[102] Woosuk Lee. Combining the top-down propagation and bottom-up enumeration for inductive program synthesis. *Proc. ACM Program. Lang.*, 5(POPL):1–28, 2021. doi:10.1145/3434335.

[103] Nicola Leone, Gerald Pfeifer, Wolfgang Faber, Thomas Eiter, Georg Gottlob, Simona Perri, and Francesco Scarcello. The DLV system for knowledge representation and reasoning. *ACM Trans. Comput. Log.*, 7(3):499–562, 2006. doi:10.1145/1149114.1149117.

[104] Wen-Ding Li and Kevin Ellis. Is programming by example solved by llms? In *NeurIPS*, 2024. URL: http://papers.nips.cc/paper_files/paper/2024/hash/4eff61b79274124bc71efe2ee9772f95-Abstract-Conference.html.

[105] Jia Hui Liang, Vijay Ganesh, Pascal Poupart, and Krzysztof Czarnecki. Learning rate based branching heuristic for SAT solvers. In *SAT*, volume 9710 of *LNCS*, pages 123–140. Springer, 2016. doi:10.1007/978-3-319-40970-2_9.

[106] Vladimir Lifschitz. Answer set programming and plan generation. *Artificial Intelligence*, 138(1-2):39–54, 2002. doi:10.1016/S0004-3702(02)00186-8.

[107] Brian S. Lin, Jiaxin Yuan, Zihan Zhou, Shouli Wang, Shuo Wang, Cunliang Kong, Qi Shi, Yuxuan Li, Liner Yang, Zhiyuan Liu, and Maosong Sun. On llm-based scientific inductive reasoning beyond equations. In *EMNLP*, pages 9371–9394. Association for Computational Linguistics, 2025. doi:10.18653/V1/2025.EMNLP-MAIN.476.

[108] Fangzhen Lin and Yuting Zhao. ASSAT: computing answer sets of a logic program by SAT solvers. In *AAAI*, pages 112–118. AAAI Press / The MIT Press, 2002. URL: http://www.aaai.org/Library/AAAI/2002/aaai02-018.php.

[109] Haojun Ma, Aman Goel, Jean-Baptiste Jeannin, Manos Kapritsos, Baris Kasikci, and Karem A. Sakallah. I4: incremental inference of inductive invariants for verification of distributed protocols. In *SOSP*, pages 370–384. ACM, 2019. doi:10.1145/3341301.3359651.

[110] Lezhi Ma, Shangqing Liu, Yi Li, Xiaofei Xie, and Lei Bu. SpecGen: Automated generation of formal program specifications via large language models. *CoRR*, abs/2401.08807, 2024. doi:10.48550/ARXIV.2401.08807.

[111] Chris Martens, Robert J. Simmons, and Michael Arntzenius. Finite-choice logic programming. *Proc. ACM Program. Lang.*, 9(POPL):362–390, 2025. doi:10.1145/3704849.

[112] Baoluo Meng, Andrew Reynolds, Cesare Tinelli, and Clark W. Barrett. Relational constraint solving in SMT. In *CADE*, volume 10395 of *LNCS*, pages 148–165. Springer, 2017. doi:10.1007/978-3-319-63046-5_10.

[113] Facundo Molina, Renzo Degiovanni, Pablo Ponzio, Germán Regis, Nazareno Aguirre, and Marcelo F. Frias. Training binary classifiers as data structure invariants. In *ICSE*, pages 759–770. IEEE / ACM, 2019. doi:10.1109/ICSE.2019.00084.

[114] Facundo Molina, Pablo Ponzio, Nazareno Aguirre, and Marcelo F. Frias. EvoSpex: An evolutionary algorithm for learning postconditions. In *ICSE*, pages 1223–1235. IEEE, 2021. doi:10.1109/ICSE43902.2021.00112.

[115] Federico Mora, Ankush Desai, Elizabeth Polgreen, and Sanjit A. Seshia. Message chains for distributed system verification. *Proc. ACM Program. Lang.*, 7(OOPSLA2):2224–2250, 2023. doi:10.1145/3622876.

[116] Stephen H. Muggleton. Inductive Logic Programming. *New Gener. Comput.*, 8(4):295–318, 1991. doi:10.1007/BF03037089.

[117] Stephen H. Muggleton. Inverse entailment and Progol. *New generation computing*, 13(3):245–286, 1995. doi:10.1007/BF03037227.

[118] Stephen H. Muggleton, Dianhuan Lin, Niels Pahlavi, and Alireza Tamaddoni-Nezhad. Meta-interpretive learning: application to grammatical inference. *Mach. Learn.*, 94(1):25–49, 2014. doi:10.1007/s10994-013-5358-3.

[119] Stephen H. Muggleton and Luc De Raedt. Inductive logic programming: Theory and methods. *J. Log. Program.*, 19/20:629–679, 1994. doi:10.1016/0743-1066(94)90035-3.

[120] Nicholas Nethercote, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido Tack. MiniZinc: Towards a standard CP modelling language. In *CP*, volume 4741 of *LNCS*, pages 529–543. Springer, 2007. doi:10.1007/978-3-540-74970-7_38.

[121] Ilkka Niemelä. Logic programs with stable model semantics as a constraint programming paradigm. *Ann. Math. Artif. Intell.*, 25(3-4):241–273, 1999. doi:10.1023/A:1018930122475.

[122] Ilkka Niemelä. Answer set programming without unstratified negation. In *ICLP*, volume 5366 of *LNCS*, pages 88–92. Springer, 2008. doi:10.1007/978-3-540-89982-2_15.

[123] Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. In *CSL*, volume 2142 of *LNCS*, pages 1–19. Springer, 2001. doi:10.1007/3-540-44802-0_1.

[124] Saswat Padhi, Todd D. Millstein, Aditya V. Nori, and Rahul Sharma. Overfitting in synthesis: Theory and practice. In *CAV*, volume 11561 of *LNCS*, pages 315–334. Springer, 2019. doi:10.1007/978-3-030-25540-4_17.

[125] Oded Padon, Kenneth L. McMillan, Aurojit Panda, Mooly Sagiv, and Sharon Shoham. Ivy: Safety verification by interactive generalization. In *PLDI*, pages 614–630. ACM, 2016. doi:10.1145/2908080.2908118.

[126] Kanghee Park, Loris D'Antoni, and Thomas Reps. Synthesizing specifications. *Proc. ACM Program. Lang.*, 7(OOPSLA2):285:1–285:30, 2023. doi:10.1145/3622861.

[127] ParKissat-RS, https://github.com/shaowei-cai-group/ParKissat-RS. Commit 537bbea; accessed 2026-01-15.

[128] Edgar Pek, Xiaokang Qiu, and P. Madhusudan. Natural proofs for data structure manipulation in C using separation logic. In *PLDI*, pages 440–451. ACM, 2014. doi:10.1145/2594291.2594325.

[129] Laurent Perron and Frédéric Didier. CP-SAT. URL: https://developers.google.com/optimization/cp/cp_solver/.

[130] George Pîrlea, Vladimir Gladshtein, Elad Kinsbruner, Qiyuan Zhao, and Ilya Sergey. Veil: A framework for automated and interactive verification of transition systems. In *CAV*, volume 15933 of *LNCS*, pages 26–41. Springer, 2025. doi:10.1007/978-3-031-98682-6_2.

[131] Ruzica Piskac, Leonardo Mendonça de Moura, and Nikolaj S. Bjørner. Deciding effectively propositional logic using DPLL and substitution sets. *J. Autom. Reason.*, 44(4):401–424, 2010. doi:10.1007/S10817-009-9161-6.

[132] Ruzica Piskac, Thomas Wies, and Damien Zufferey. GRASShopper - complete heap verification with mixed specifications. In *TACAS*, volume 8413 of *LNCS*, pages 124–139. Springer, 2014. doi:10.1007/978-3-642-54862-8_9.

[133] Nadia Polikarpova and Ilya Sergey. Structuring the synthesis of heap-manipulating programs. *Proc. ACM Program. Lang.*, 3(POPL):72:1–72:30, 2019. doi:10.1145/3290385.

[134] Mukund Raghothaman, Jonathan Mendelson, David Zhao, Mayur Naik, and Bernhard Scholz. Provenance-guided synthesis of datalog programs. *Proc. ACM Program. Lang.*, 4(POPL):62:1–62:27, 2020. doi:10.1145/3371130.

[135] Andrew Reynolds, Cesare Tinelli, Amit Goel, and Sava Krstic. Finite model finding in SMT. In *CAV*, volume 8044 of *LNCS*, pages 640–655. Springer, 2013. doi:10.1007/978-3-642-39799-8_42.

[136] Andrew Reynolds, Cesare Tinelli, Amit Goel, Sava Krstic, Morgan Deters, and Clark W. Barrett. Quantifier instantiation techniques for finite model finding in SMT. In *CADE*, volume 7898 of *LNCS*, pages 377–391. Springer, 2013. doi:10.1007/978-3-642-38574-2_26.

[137] John C Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *LICS*, pages 55–74. IEEE, IEEE Computer Society, 2002. doi:10.1109/LICS.2002.1029817.

[138] John C. Reynolds. An introduction to separation logic (preliminary draft). Technical report, Carnegie Mellon University, 2008.

[139] Nicola Rizzo and Agostino Dovier. 3coSoKu and its declarative modeling. *J. Log. Comput.*, 32(2):307–330, 2022. URL: https://doi.org/10.1093/logcom/exab086, doi:10.1093/LOGCOM/EXAB086.

[140] Marcus Rossel, Rudi Schneider, Thomas Koehler, Michel Steuwer, and Andrés Goens. Towards pen-and-paper-style equational reasoning in interactive theorem provers by equality saturation. *Proc. ACM Program. Lang.*, 10(POPL):718–747, 2026. doi:10.1145/3776667.

[141] Subhajit Roy. From concrete examples to heap manipulating programs. In *SAS*, volume 7935 of *LNCS*, pages 126–149. Springer, 2013. doi:10.1007/978-3-642-38856-9_9.

[142] Noah Rubin, Curtis Bright, Kevin K. H. Cheung, and Brett Stevens. Improving integer and constraint programming for Graeco-Latin squares. In *ICTAI*, pages 604–608. IEEE, 2021. URL: https://arxiv.org/abs/2103.11018.

[143] William Schultz, Edward Ashton, Heidi Howard, and Stavros Tripakis. Scalable, interpretable distributed protocol verification by inductive proof slicing. *CoRR*, abs/2404.18048, 2024. doi:10.48550/ARXIV.2404.18048.

[144] Irfansha Shaik and Jaco van de Pol. Optimal layout synthesis for deep quantum circuits on NISQ processors with 100+ qubits. In *SAT*,

volume 305 of *LIPIcs*, pages 26:1–26:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2024. doi:10.4230/LIPICS.SAT.2024.26.

[145] Xujie Si, Woosuk Lee, Richard Zhang, Aws Albarghouthi, Paraschos Koutris, and Mayur Naik. Syntax-guided synthesis of datalog programs. In *ESEC/SIGSOFT FSE*, pages 515–527. ACM, 2018. doi:10.1145/3236024.3236034.

[146] Mihaela Sighireanu, Juan Antonio Navarro Pérez, Andrey Rybalchenko, Nikos Gorogiannis, Radu Iosif, Andrew Reynolds, Cristina Serban, Jens Katelaan, Christoph Matheja, Thomas Noll, Florian Zuleger, Wei-Ngan Chin, Quang Loc Le, Quang-Trung Ta, Ton-Chanh Le, Thanh-Toan Nguyen, Siau-Cheng Khoo, Michal Cyprian, Adam Rogalewicz, Tomás Vojnar, Constantin Enea, Ondrej Lengál, Chong Gao, and Zhilin Wu. SL-COMP: competition of solvers for separation logic. In *TACAS*, volume 11429 of *LNCS*, pages 116–132. Springer, 2019. doi:10.1007/978-3-030-17502-3_8.

[147] João P. Marques Silva and Karem A. Sakallah. GRASP: A search algorithm for propositional satisfiability. *IEEE Trans. Computers*, 48(5):506–521, 1999. doi:10.1109/12.769433.

[148] Patrik Simons, Ilkka Niemelä, and Timo Soininen. Extending and implementing the stable model semantics. *Artif. Intell.*, 138(1-2):181–234, 2002. doi:10.1016/S0004-3702(02)00187-X.

[149] Rishabh Singh and Armando Solar-Lezama. SPT: Storyboard Programming Tool. In *CAV*, volume 7358 of *LNCS*, pages 738–743. Springer, 2012. doi:10.1007/978-3-642-31424-7_58.

[150] Armando Solar-Lezama. Program sketching. *Int. J. Softw. Tools Technol. Transf.*, 15(5-6):475–495, 2013. doi:10.1007/S10009-012-0249-7.

[151] Mate Soos, Karsten Nohl, and Claude Castelluccia. Extending SAT solvers to cryptographic problems. In *SAT*, volume 5584 of *LNCS*,

pages 244–257. Springer, 2009. doi:10.1007/978-3-642-02777-2_24.

[152] Chuyue Sun, Viraj Agashe, Saikat Chakraborty, Jubi Taneja, Clark W. Barrett, David L. Dill, Xiaokang Qiu, and Shuvendu K. Lahiri. ClassInvGen: Class invariant synthesis using large language models. In *SAIV*, volume 15947 of *LNCS*, pages 64–96. Springer, 2025. doi:10.1007/978-3-031-99991-8_4.

[153] Bochen Tan and Jason Cong. Optimal layout synthesis for quantum computing. In *ICCAD*, pages 137:1–137:9. IEEE, 2020. doi:10.1145/3400302.3415620.

[154] Aalok Thakkar, Aaditya Naik, Nathaniel Sands, Rajeev Alur, Mayur Naik, and Mukund Raghothaman. Example-guided synthesis of relational queries. In *PLDI*, pages 1110–1125. ACM, 2021. doi:10.1145/3453483.3454098.

[155] Emina Torlak and Daniel Jackson. Kodkod: A relational model finder. In *TACAS*, volume 4424 of *LNCS*, pages 632–647. Springer, 2007. doi:10.1007/978-3-540-71209-1_49.

[156] Grigori S Tseitin. On the complexity of derivation in propositional calculus. In *Automation of reasoning: 2: Classical papers on computational logic 1967–1970*, pages 466–483. Springer, 1983.

[157] Hiroshi Unno, Takeshi Tsukada, and Jie-Hong Roland Jiang. Solving higher-order quantified boolean satisfiability via higher-order model checking. In *AAAI*, volume 39, pages 11372–11380. AAAI Press, 2025. doi:10.1609/aaai.v39i11.33237.

[158] Robbert van Renesse and Deniz Altinbuken. Paxos made moderately complex. *ACM Comput. Surv.*, 47(3):42:1–42:36, 2015. doi:10.1145/2673577.

[159] Ondřej Čepek. *Structural properties and minimization of Horn Boolean functions*. PhD thesis, Rutgers University, 1995.

[160] Marina De Vos and Dirk Vermeir. On the role of negation in choice logic programs. In *LPNMR*, volume 1730 of *LNCS*, pages 236–246. Springer, 1999. doi:10.1007/3-540-46767-X_17.

[161] Ruocheng Wang, Eric Zelikman, Gabriel Poesia, Yewen Pu, Nick Haber, and Noah D. Goodman. Hypothesis search: Inductive reasoning with language models. In *ICLR*. OpenReview.net, 2024. URL: https://openreview.net/forum?id=G7UtIGQmjm.

[162] Yisu Remy Wang, Mahmoud Abo Khamis, Hung Q. Ngo, Reinhard Pichler, and Dan Suciu. Optimizing recursive queries with program synthesis. In *SIGMOD*, pages 79–93. ACM, 2022. doi:10.1145/3514221.3517827.

[163] Yasunari Watanabe, Kiran Gopinathan, George Pîrlea, Nadia Polikarpova, and Ilya Sergey. Certifying the synthesis of heap-manipulating programs. *Proc. ACM Program. Lang.*, 5(ICFP):1–29, 2021. doi:10.1145/3473589.

[164] Cheng Wen, Jialun Cao, Jie Su, Zhiwu Xu, Shengchao Qin, Mengda He, Haokun Li, Shing-Chi Cheung, and Cong Tian. Enchanting program specification synthesis by large language models using static analysis and program verification. In *CAV*, volume 14682 of *LNCS*, pages 302–328. Springer, 2024. doi:10.1007/978-3-031-65630-9_16.

[165] Ziyi Yang, George Pîrlea, and Ilya Sergey. Inductive first-order formula synthesis by ASP: A case study in invariant inference. In *ICLP*, EPTCS, pages 511–527, January 2025. doi:10.4204/EPTCS.439.35.

[166] Ziyi Yang and Ilya Sergey. Inductive synthesis of inductive heap predicates. *Proc. ACM Program. Lang.*, 9(OOPSLA1):169–195, 2025. doi:10.1145/3720420.

[167] Ziyi Yang and Ilya Sergey. Semi-declarative language for combinatorial search. *Proc. ACM Program. Lang.*, 10(OOPSLA2), October 2026. doi:10.1145/3839531.

[168] Jianan Yao, Runzhou Tao, Ronghui Gu, and Jason Nieh. DuoAI: Fast, automated inference of inductive invariants for verifying distributed protocols. In *OSDI*, pages 485–501. USENIX Association, 2022. URL: https://www.usenix.org/conference/osdi22/presentation/yao.

[169] Jianan Yao, Runzhou Tao, Ronghui Gu, Jason Nieh, Suman Jana, and Gabriel Ryan. DistAI: Data-Driven Automated Invariant Learning for Distributed Protocols. In *OSDI*, pages 405–421. USENIX Association, 2021. URL: https://www.usenix.org/conference/osdi21/presentation/yao.

[170] Yihong Zhang, Yisu Remy Wang, Oliver Flatt, David Cao, Philip Zucker, Eli Rosenthal, Zachary Tatlock, and Max Willsey. Better together: Unifying datalog and equality saturation. *Proc. ACM Program. Lang.*, 7(PLDI):468–492, 2023. doi:10.1145/3591239.

[171] Neng-Fa Zhou, Håkan Kjellerstrand, and Jonathan Fruhman. *Constraint Solving and Planning with Picat*. Springer Briefs in Intelligent Systems. Springer, 2015. doi:10.1007/978-3-319-25883-6.

[172] He Zhu, Gustavo Petri, and Suresh Jagannathan. Automatically learning shape specifications. In *PLDI*, pages 491–507. ACM, 2016. doi:10.1145/2908080.2908125.

# Appendix A

# Deferred Proofs and Model Listings for Chapter 5

## A.1 Deferred Proofs

### A.1.1 Translation Correctness

This subsection gives the deferred proofs of Theorem 4, Theorem 5, and Theorem 6.

**Statement of Theorem 4 (Correctness and Termination of IC conversion).** *For every closed FOL formula $\varphi$ over a finite Herbrand base $\mathcal{A}_{\mathcal{D}}$, the translation procedure terminates and produces a finite set of ICs and auxiliary rules that is equivalent to $\varphi$ over $\mathcal{A}_{\mathcal{D}}$.*

*Proof of Theorem 4. Termination.* Each QE step eliminates one quantifier alternation, strictly reducing the alternation depth. Since $\varphi$ has finitely many alternations and each recursive call (Algorithm 6) operates on a strictly simpler sub-formula, the procedure terminates.

*Correctness.* We show that for every interpretation $M$ over $\mathcal{A}_{\mathcal{D}}$:

$$M \not\models \varphi \iff M \text{ is eliminated by the translated ICs and auxiliary rules.}$$

*(i) Conjunction splitting* decomposes $\varphi$ into conjuncts $\varphi_1 \wedge \cdots \wedge \varphi_m$ using the rules of Table 5.1. By logical equivalence, $M \models \varphi$ iff $M \models \varphi_j$ for all $j$.

*(ii) Normalisation* applies standard equivalences (De Morgan, double negation, quantifier scoping). These are sound, so satisfying interpretations are unchanged.

*(iii) Aggregate-based QE.* Over a finite domain, $M \models \exists \bar{Y}.\, \psi(\bar{X}, \bar{Y})$ iff the count $|\{\bar{d} \in \mathcal{A}_{\mathcal{D}}^{|\bar{Y}|} \mid M \models \psi(\bar{X}, \bar{d})\}| \geq 1$, which is exactly what the aggregate constraint tests.

*(iv) Auxiliary-based QE.* For $\exists \bar{Y}.\, \bigvee_i (L_{i,1} \wedge \cdots \wedge L_{i,n_i})$, the auxiliary predicate $\mathsf{aux}_k$ is defined by rules $\mathsf{aux}_k(\bar{X})$ `:-` $L_{i,1}, \ldots, L_{i,n_i}$ for each disjunct $i$. Let $\hat{M}$ be the unique minimal extension of $M$ satisfying these rules. Then $M \models \exists \bar{Y}.\, \bigvee_i \cdots$ iff $\mathsf{aux}_k(\bar{a}) \in \hat{M}$. The IC `:- not aux_k`$(\bar{X}), dom(\bar{X})$. eliminates exactly those $M$ where the existential is false, and $\hat{M} \restriction_{\Sigma} = M$.

Composing (i)–(iv), the translated ICs filter out exactly those $M$ not satisfying $\varphi$, as required. □

**Statement of Theorem 5 (Block-level preservation).** *Let $P_i = (\mathcal{R}_i, \mathcal{C}_i, \Phi_i)$ be a PCLP block, $I$ an input answer set over $\Sigma$, and $\hat{P}_i$ the ASP program produced by the translation. Then: (1) when $\Phi_i = \emptyset$, $\mathrm{AS}(\hat{P}_i \cup I) = \mathcal{S}^*(P_i, I)$; (2) when $\Phi_i \neq \emptyset$, $\{M \restriction_{\Sigma} \mid M \in \mathrm{AS}(\hat{P}_i \cup I)\} = \mathcal{M}(P_i, I)$.*

*Proof of Theorem 5. Part 1: Rules.* The syntactic translation of $\mathcal{R}_i$ and $\mathcal{C}_i$ preserves the candidate set $\mathcal{S}^*(P_i, I)$.

- *Definite rules:* each $H \leftarrow B_1, \ldots, B_k$ translates to $H$ `:-` $B_1, \ldots, B_k$. All bodies are positive and $\mathcal{A}_{\mathcal{D}}$ is finite, so the $T_P$ operator and ASP's immediate-consequence operator coincide. The least fixpoint (the $\mathcal{R}$-closure) is identical.
- *Choice rules:* each $\{H\} \leftarrow B_1, \ldots, B_k$ translates to $\{H\}$ `:-` $B_1, \ldots, B_k$. By the semantics of choice rules [122, 160], each subset of fireable choice heads yields a candidate, matching the non-deterministic branching in $\mathcal{S}^*$.

Thus, when $\Phi_i = \emptyset$, $\mathrm{AS}(\hat{P}_i \cup I) = \mathcal{S}^*(P_i, I)$.

*Part 2: Constraints.* When $\Phi_i \neq \emptyset$, the PCLP semantics gives $\mathcal{M}(P_i, I) = \{M \in \mathcal{S}^*(P_i, I) \mid \forall \varphi \in \Phi_i,\ M \models \varphi\}$. By Theorem 4, for each $\varphi \in \Phi_i$ the translated ICs eliminate exactly those candidates $M$ with $M \not\models \varphi$, and projection onto $\Sigma$ recovers the original vocabulary. Part 1 establishes that $\mathrm{AS}(\hat{P}_i \cup I)$ (before ICs) equals $\mathcal{S}^*(P_i, I)$; combining with Part 2 gives $\{M \restriction_{\Sigma} \mid M \in \mathrm{AS}(\hat{P}_i \cup I)\} = \mathcal{M}(P_i, I)$. □

**Statement of Theorem 6 (SetLah! program correctness).** *Let* $\Pi = B_1, \ldots, B_n$ *be a well-formed SetLah! program, and let* $\hat{\Pi}$ *be the ASP program produced by translating each block* $B_i$*. Then* $\mathcal{M}(\Pi) = \{M \restriction_\Sigma | \ M \in \mathrm{AS}(\hat{\Pi})\}$.

*Proof of Theorem 6.* By induction on the number of blocks $n$.

*Base case.* $\mathcal{M}_0 = \{I_0\}$, the set of ground facts from external predicates and constants, which is the same initial state for both the SetLah! semantics and the ASP program $\hat{\Pi}$.

*Inductive step.* Assume $\{M \restriction_\Sigma | \ M \in \mathrm{AS}(\hat{\Pi}_{1..i-1})\} = \mathcal{M}_{i-1}$. The SetLah! semantics (§ 5.3.2.2) defines $\mathcal{M}_i = \bigcup_{I \in \mathcal{M}_{i-1}} \mathcal{M}(P_i, I)$. By Theorem 5, for every $I \in \mathcal{M}_{i-1}$, $\{M \restriction_\Sigma | \ M \in \mathrm{AS}(\hat{P}_i \cup I)\} = \mathcal{M}(P_i, I)$. Therefore $\{M \restriction_\Sigma | \ M \in \mathrm{AS}(\hat{\Pi}_{1..i})\} = \mathcal{M}_i$. At $i = n$ we obtain $\mathrm{AS}(\hat{\Pi}) \restriction_\Sigma = \mathcal{M}_n = \mathcal{M}(\Pi)$. □

### A.1.2 Finiteness of the Reachable Set

We give a full proof that the set of formulas reachable from any $\varphi_0$ of finite size in each phase, by applying the phase-oriented rules of Table 5.1, is finite. The proof uses the standard termination method with polynomial interpretations that respect associativity and commutativity (Ben Cherifa and Lescanne [32]), combined with a finite fiber bound for negation placement.

#### Structural Invariants

**Lemma 1** (Atom invariance)**.** *For every* $\psi \in \mathcal{R}(\varphi_0)$*, we have* $\mathrm{Atoms}(\psi) \subseteq \mathrm{Atoms}(\varphi_0)$.

*Proof.* We verify per rule. Identity, annihilation, and idempotence delete sub-formulas and introduce no new atoms. De Morgan, quantifier negation, and double negation redistribute or remove negations without altering atomic sub-formulas. Distribution rearranges and possibly duplicates existing sub-formulas but creates no new atoms. Quantifier rules reposition quantifiers without altering atomic sub-formulas within their scope. □

**Lemma 2** (Variable invariance)**.** *For every* $\psi \in \mathcal{R}(\varphi_0)$*, the set of variables appearing in* $\psi$ *is a subset of those in* $\varphi_0$.

*Proof.* No rule introduces a fresh variable. Quantifier rules move existing quantifiers; the side condition $x \notin \mathrm{FV}(\psi)$ prevents accidental capture but in any case creates no new variables. □

Set $k = |\mathrm{Atoms}(\varphi_0)|$ and let $V$ be the set of variables in $\varphi_0$.

**Remark 1** (Necessity of one-wayness and phase orientation)**.** *Making idempotence bidirectional ($\varphi \leftrightarrow \varphi \wedge \varphi$) breaks finiteness immediately: from any atom $p$ one generates $p$, $p \wedge p$, $(p \wedge p) \wedge (p \wedge p), \ldots$ Likewise, allowing both distribution laws in both directions simultaneously allows alternating $\wedge$-over-$\vee$ expansion and $\vee$-over-$\wedge$ expansion indefinitely. Both one-way idempotence and phase-oriented distribution are therefore necessary for finiteness.*

### Negation-Normal Form and Phase Orientation

A formula is in *negation-normal form* (NNF) if negations appear only directly above atoms. Define $\nu(\psi)$ as the result of exhaustively applying De Morgan ($\neg(\varphi \wedge \psi) \to \neg\varphi \vee \neg\psi$ and dual), quantifier negation ($\neg\forall x\, \varphi \to \exists x\, \neg\varphi$ and dual), and double-negation elimination ($\neg\neg\varphi \to \varphi$) until NNF is reached; this terminates because each step either eliminates two negations or pushes a $\neg$ past one connective or quantifier.

**Lemma 3** (De Morgan and double negation)**.** *The De Morgan, quantifier-negation, and double-negation rules, acting alone, generate only finitely many formulas from $\varphi_0$.*

*Proof.* Define the *unsigned skeleton* $\bar{\varphi}$ by stripping all negations and replacing dual connectives/quantifiers by their negation-pushed counterparts. It has $n \leq \mathrm{size}(\varphi)$ negation sites (one above each connective, quantifier, and atom). De Morgan, quantifier negation, and double negation merely transform among negation assignments to these sites (with double negation collapsing pairs). There are at most $2^n \leq 2^{\mathrm{size}(\varphi_0)}$ such assignments, which is finite. □

**Phase-Oriented Distribution and Polynomial Interpretations**

The two distribution laws are

$$A \wedge (B \vee C) \;\leftrightarrow\; (A \wedge B) \vee (A \wedge C) \tag{A.1}$$

$$A \vee (B \wedge C) \;\leftrightarrow\; (A \vee B) \wedge (A \vee C) \tag{A.2}$$

To prevent alternating expansions between Equation A.1 and Equation A.2, we orient distribution and quantifier scoping by phase (evaluated on the NNF image $\nu(\psi)$, i.e., respecting negation polarity):

- **Toward CNF (Phase 1, conjunction splitting):** allow only *Equation A.2* $\rightarrow$ $(A \vee (B \wedge C) \;\rightarrow\; (A \vee B) \wedge (A \vee C))$ and *Equation A.1* $\leftarrow$ $((A \wedge B) \vee (A \wedge C) \;\rightarrow\; A \wedge (B \vee C))$, together with the quantifier and prenex rules oriented to bubble $\wedge$ outward and push $\vee$ inward (where $x \notin \mathrm{FV}(\psi)$ in the last two rules):

$$\forall x\,(\varphi \wedge \psi) \;\rightarrow\; (\forall x\,\varphi) \wedge (\forall x\,\psi), \quad (\exists x\,\varphi) \vee (\exists x\,\psi) \;\rightarrow\; \exists x\,(\varphi \vee \psi),$$
$$(\forall x\,\varphi) \vee \psi \;\rightarrow\; \forall x\,(\varphi \vee \psi), \qquad\qquad \exists x\,(\varphi \wedge \psi) \;\rightarrow\; (\exists x\,\varphi) \wedge \psi.$$

- **Toward DNF (Phase 2, conversion to IC form):** dually allow only *Equation A.1* $\rightarrow$ $(A \wedge (B \vee C) \;\rightarrow\; (A \wedge B) \vee (A \wedge C))$ and *Equation A.2* $\leftarrow$ $((A \vee B) \wedge (A \vee C) \rightarrow A \vee (B \wedge C))$, together with the dual quantifier and prenex orientations (swapping $\wedge$ with $\vee$ and $\forall$ with $\exists$, which pulls $\exists$ and $\forall$ outward over $\wedge$ to form the IC prefix).

Following Ben Cherifa and Lescanne [32], we define an AC-compatible polynomial interpretation $[\![\cdot]\!]$ on NNF formulas for the toward-CNF phase by assigning every leaf $\ell \in L \cup \{\top, \bot\}$ (where $L = \{a, \neg a \mid a \in \mathrm{Atoms}(\varphi_0)\}$) weight $[\![\ell]\!] = 2$ (any integer $\geq 2$ suffices), and setting:

$$[\![A \vee B]\!] = [\![A]\!] \cdot [\![B]\!], \qquad [\![A \wedge B]\!] = [\![A]\!] + [\![B]\!] + 1,$$
$$[\![Qx.\,A]\!] = 3[\![A]\!] + 1 \quad (Q \in \{\forall, \exists\}).$$

For the toward-DNF phase, we use the dual interpretation $[\![\cdot]\!]_{\mathrm{DNF}}$ obtained by swapping the interpretations of $\wedge$ and $\vee$: $[\![A \wedge B]\!]_{\mathrm{DNF}} = [\![A]\!]_{\mathrm{DNF}} \cdot [\![B]\!]_{\mathrm{DNF}}$ and $[\![A \vee B]\!]_{\mathrm{DNF}} = [\![A]\!]_{\mathrm{DNF}} + [\![B]\!]_{\mathrm{DNF}} + 1$, with $[\![Qx.\,A]\!]_{\mathrm{DNF}} = 3[\![A]\!]_{\mathrm{DNF}} + 1$ and leaf weight $\geq 2$.

**Proposition 1** (AC-invariance and strict decrease)**.** *For every NNF formula* $\psi$, $[\![\psi]\!] \geq 2$ *and* $\text{size}(\psi) \leq [\![\psi]\!]$. *In the toward-CNF phase, commutativity, associativity, and quantifier commutation preserve* $[\![\cdot]\!]$, *while every other phase-oriented rule on NNF formulas strictly decreases* $[\![\cdot]\!]$. *The dual statement holds for* $[\![\cdot]\!]_{\text{DNF}}$ *in the toward-DNF phase.*

*Proof.* By structural induction, $[\![\psi]\!] \geq 2$, $\text{size}(\psi) \leq [\![\psi]\!]$, and each operator interpretation is strictly monotone in all arguments (since $[\![A]\!] \geq 2$).

- *AC and quantifier commutation:* Commutativity of $\wedge$ and $\vee$ is immediate. For associativity,

$$[\![A \vee (B \vee C)]\!] = [\![A]\!][\![B]\!][\![C]\!] = [\![(A \vee B) \vee C]\!],$$
$$[\![A \wedge (B \wedge C)]\!] = [\![A]\!] + [\![B]\!] + [\![C]\!] + 2 = [\![(A \wedge B) \wedge C]\!].$$

  For quantifier commutation, $[\![Qx.\, Qy.\, \varphi]\!] = 3(3[\![\varphi]\!]+1)+1 = 9[\![\varphi]\!]+4 = [\![Qy.\, Qx.\, \varphi]\!]$.

- *Identity and annihilation:* Since $[\![\varphi]\!] \geq 2$ and $[\![\top]\!] = [\![\bot]\!] = 2$, we have $[\![\varphi \wedge \top]\!] = [\![\varphi]\!] + 3 > [\![\varphi]\!]$, $[\![\varphi \vee \bot]\!] = 2[\![\varphi]\!] > [\![\varphi]\!]$, $[\![\varphi \wedge \bot]\!] = [\![\varphi]\!] + 3 > 2 = [\![\bot]\!]$, and $[\![\varphi \vee \top]\!] = 2[\![\varphi]\!] > 2 = [\![\top]\!]$.

- *Both idempotences:* Since $[\![\varphi]\!] \geq 2$, both idempotence rules strictly decrease $[\![\cdot]\!]$:

$$[\![\varphi \vee \varphi]\!] = [\![\varphi]\!]^2 > [\![\varphi]\!] \qquad \text{and} \qquad [\![\varphi \wedge \varphi]\!] = 2[\![\varphi]\!] + 1 > [\![\varphi]\!].$$

- *Distribution toward CNF (Equation A.2 $\rightarrow$ and Equation A.1 $\leftarrow$):* Since $[\![A]\!], [\![B]\!], [\![C]\!] \geq 2$,

$$\begin{aligned}[\![A \vee (B \wedge C)]\!] &= [\![A]\!][\![B]\!] + [\![A]\!][\![C]\!] + [\![A]\!] \\ &> [\![A]\!][\![B]\!] + [\![A]\!][\![C]\!] + 1 = [\![(A \vee B) \wedge (A \vee C)]\!], \\ [\![(A \wedge B) \vee (A \wedge C)]\!] &= ([\![A]\!] + [\![B]\!] + 1)([\![A]\!] + [\![C]\!] + 1) \\ &> [\![A]\!] + [\![B]\!][\![C]\!] + 1 = [\![A \wedge (B \vee C)]\!].\end{aligned}$$

- *Quantifier distribution and prenex conversion:* Since $[\![\varphi]\!], [\![\psi]\!] \geq 2$,

$$[\![\forall x\,(\varphi \wedge \psi)]\!] = 3[\![\varphi]\!] + 3[\![\psi]\!] + 4 > 3[\![\varphi]\!] + 3[\![\psi]\!] + 3 = [\![(\forall x\,\varphi) \wedge (\forall x\,\psi)]\!],$$
$$[\![(\exists x\,\varphi) \vee (\exists x\,\psi)]\!] = (3[\![\varphi]\!] + 1)(3[\![\psi]\!] + 1) > 3[\![\varphi]\!][\![\psi]\!] + 1 = [\![\exists x\,(\varphi \vee \psi)]\!],$$
$$[\![(\forall x\,\varphi) \vee \psi]\!] = 3[\![\varphi]\!][\![\psi]\!] + [\![\psi]\!] > 3[\![\varphi]\!][\![\psi]\!] + 1 = [\![\forall x\,(\varphi \vee \psi)]\!],$$
$$[\![\exists x\,(\varphi \wedge \psi)]\!] = 3[\![\varphi]\!] + 3[\![\psi]\!] + 4 > 3[\![\varphi]\!] + [\![\psi]\!] + 2 = [\![(\exists x\,\varphi) \wedge \psi]\!].$$

By strict monotonicity of $[\![\cdot]\!]$, applying any of these rules to a sub-formula strictly decreases the interpretation of the entire formula. The proof for $[\![\cdot]\!]_{\mathrm{DNF}}$ in the toward-DNF phase is completely symmetric by swapping $\wedge$ and $\vee$ (and $\forall$ and $\exists$). □

**Lemma 4** (Measure and size ceiling)**.** *Let $N := [\![\nu(\varphi_0)]\!]$ in the toward-CNF phase (or $N := [\![\nu(\varphi_0)]\!]_{\mathrm{DNF}}$ in the toward-DNF phase). For every NNF formula $\psi \in \mathcal{R}(\nu(\varphi_0))$, $\mathrm{size}(\psi) \leq [\![\psi]\!] \leq N$.*

*Proof.* By Proposition 1, every rewriting step on NNF formulas either preserves $[\![\cdot]\!]$ (AC and quantifier commutation) or strictly decreases $[\![\cdot]\!]$, and $\mathrm{size}(\psi) \leq [\![\psi]\!]$. Hence $\mathrm{size}(\psi) \leq [\![\psi]\!] \leq [\![\nu(\varphi_0)]\!] = N$. □

**Corollary 1** (Finite NNF reachable set)**.** *The NNF formulas in $\mathcal{R}(\nu(\varphi_0))$ form a finite set.*

*Proof.* By Lemmas 1 and 2, all literals belong to $L \cup \{\top, \bot\}$ ($|L| = 2k$) and all quantified variables belong to $V$. By Lemma 4, every reachable NNF formula has syntax-tree size $\leq N$. Since there are only finitely many syntax trees of size $\leq N$ over a finite alphabet of connectives, quantifiers over $V$, and leaves $L \cup \{\top, \bot\}$, $\mathcal{R}(\nu(\varphi_0))$ is finite. □

### Lifting to Non-NNF Formulas

It remains to lift finiteness from NNF formulas to arbitrary reachable formulas by bounding the non-NNF pre-images under $\nu$.

**Lemma 5** (NNF image is NNF-reachable)**.** *For every $\psi \in \mathcal{R}(\varphi_0)$, $\nu(\psi) \in \mathcal{R}(\nu(\varphi_0))$.*

*Proof.* By induction on the length of the rewriting sequence. *Base:* $\nu(\varphi_0) \in \mathcal{R}(\nu(\varphi_0))$ trivially. *Step:* Suppose $\nu(\psi') \in \mathcal{R}(\nu(\varphi_0))$ and $\psi$ is obtained from $\psi'$ by one rule. De Morgan, quantifier negation, and double negation either preserve $\nu(\psi) = \nu(\psi')$ or cancel a pair of negations already absent in $\nu(\psi')$. Because distribution and quantifier/prenex rules are oriented according to the polarity of the sub-formula in $\nu(\psi')$, each such step on $\psi'$ induces the corresponding phase-oriented step (or a sequence of steps when distributing across a negated sub-formula $\neg\alpha$) in $\mathcal{R}(\nu(\psi')) \subseteq \mathcal{R}(\nu(\varphi_0))$. One-way reductions and AC/quantifier commutation commute with $\nu$ directly. □

**Lemma 6** (Finite fibers)**.** *For any NNF formula $\psi_N$ of size $s$, $|\nu^{-1}(\psi_N) \cap \mathcal{R}(\varphi_0)| \leq 2^s$.*

*Proof.* Any $\psi$ with $\nu(\psi) = \psi_N$ is obtained from $\psi_N$ by un-pushing negations via reverse De Morgan or reverse quantifier negation at internal connective or quantifier nodes. Double-negation *introduction* is forbidden (one-way rule), so each of the at most $s$ internal nodes of $\psi_N$ admits at most a binary choice of whether a negation sits immediately above it, giving at most $2^s \leq 2^N$ pre-images. □

**Statement of Theorem 3 (Finiteness of the reachable set).** *For every formula $\varphi_0$ of finite size, the set of formulas reachable from $\varphi_0$ in each phase by applying the phase-oriented rules from Table 5.1 is finite.*

*Proof of Theorem 3.* We decompose $\mathcal{R}(\varphi_0)$ along the NNF projection $\nu$:

$$\mathcal{R}(\varphi_0) \;=\; \bigcup_{\psi_N \in \nu(\mathcal{R}(\varphi_0))} \Big(\nu^{-1}(\psi_N) \cap \mathcal{R}(\varphi_0)\Big).$$

By Lemma 5, $\nu(\mathcal{R}(\varphi_0)) \subseteq \mathcal{R}(\nu(\varphi_0))$. By Corollary 1, $\mathcal{R}(\nu(\varphi_0))$ is finite with cardinality $F_{\mathrm{NNF}} < \infty$, and every $\psi_N \in \mathcal{R}(\nu(\varphi_0))$ has size $s \leq N = [\![\nu(\varphi_0)]\!]$ (Lemma 4). By Lemma 6, each fiber $\nu^{-1}(\psi_N) \cap \mathcal{R}(\varphi_0)$ has at most $2^N$ elements. Therefore,

$$|\mathcal{R}(\varphi_0)| \;\leq\; F_{\mathrm{NNF}} \cdot 2^N \;<\; \infty,$$

completing the proof for the toward-CNF phase; the toward-DNF phase is identical using $[\![\cdot]\!]_{\mathrm{DNF}}$. □

Table A.1: Set operator translations to ASP. $S$ ranges over set predicates; $\bowtie$ over comparison operators.

| **Set Atom (source)** | **ASP Translation** |
|---|---|
| $(\backslash subset\ A\ \{B\})$ | $\forall \bar{x}.\ (a(\bar{x}) \rightarrow b(\bar{x}))$ |
| $(\backslash partition\ F\ \{\{A\ B\ \cdots\}\})$ | `:- f(X), #count{1,a:a(X); 1,b:b(X); ...} != 1.` |
| $\#count\ \{Set_p\}(v_1, \ldots, v_k) \bowtie n$ | `#count{`$v_1, \ldots, v_k$ `: p(`$v_1, \ldots, v_k$`)} ⋈ n` |
| $\#count\ \{Set_p\} \bowtie n$ | `#count{`$\bar{X}$ `: p(`$\bar{X}$`)} ⋈ n` |

## A.2 Set Operator Extensions

The base language of choice Horn clauses with FOL constraints operates over predicates and individual variables. To support set-level reasoning common in constraint modelling (cardinality bounds, partitions, subset relationships), we introduce a family of *set atoms* that may appear inside FOL constraint formulas. Each set atom is desugared—either before or during compilation—into a combination of FOL formulas and ASP aggregates.

**Notation.** A *set predicate* $Set_p$ denotes the extension of a predicate $p$: the set of tuples $\bar{x}$ for which $p(\bar{x})$ holds. When a set predicate carries a *projection tuple* $(v_1, \ldots, v_k)$, it selects specific argument positions. We write $dom_T$ for the domain predicate of type $T$.

Table A.1 summarises the translation of each set operator into ASP.

**Subset.** A subset atom $(\backslash subset\ A\ \{B\})$ asserts that the extension of $A$ is a subset of the extension of $B$. This is translated to the FOL constraint $\forall \bar{x}.\ (a(\bar{x}) \rightarrow b(\bar{x}))$, which then passes through the standard IC translation pipeline.

**Partition.** A partition atom $(\backslash partition\ F\ \{\{A\ B\ \cdots\}\})$ asserts that the extension of $F$ is partitioned into the extensions of $A, B, \ldots$ (disjoint and covering). This is compiled to a count constraint requiring that each element of $F$ belongs to exactly one part. When the partition co-occurs

with choice rules for each part sharing the same body, the compiler merges them into a single bounded choice rule, *e.g.*, `1{a(X); b(X)}1 :- f(X).`

**Count aggregation.** The most common set atom is a cardinality constraint, optionally with projection. When a projection tuple $(v_1, \ldots, v_k)$ is given, the translation counts the number of distinct tuples satisfying $p(v_1, \ldots, v_k)$. Universally quantified variables surrounding the count become *parameters*—they appear in the ASP rule body as domain guards ($dom_T(u_i)$) but are not counted. For example:

$$\forall R.\ \#count\ \{Set_queen\}(C) = 1$$

compiles to:

```
:- dom(R), #count{C : queen(R, C)} != 1.
```

Here $R$ is a parameter (grounded by `dom(R)`), while $C$ is the counted variable.

**Future work: set comprehensions.** Set comprehensions (let-bindings) introduce local environments for FOL sub-formulas and are naturally captured by auxiliary definite rules. Since the auxiliary predicate is used only within the scope of the binding, locality-aware inlining—merging the auxiliary back into the enclosing constraint—is a promising optimisation direction.

## A.3 Model Listings

### A.3.1 CNF Encoding for Graeco–Latin Squares

The CNF baseline used for the Graeco–Latin square experiment is generated from the following clause families. For an order-$n$ instance, atoms $l(R, C, A)$

and $g(R, C, A)$ denote that cell $(R, C)$ contains symbol $A$ in the two Latin squares, respectively, where $1 \leq R, C, A \leq n$.

$$\Phi_{cell}(s) = \bigwedge_{1 \leq R,C \leq n} \left( \bigvee_{1 \leq A \leq n} s(R, C, A) \right) \wedge \bigwedge_{\substack{1 \leq R,C \leq n \\ 1 \leq A_1 < A_2 \leq n}} (\neg s(R, C, A_1) \vee \neg s(R, C, A_2)),$$

$$\Phi_{row}(s) = \bigwedge_{\substack{1 \leq R,A \leq n \\ 1 \leq C_1 < C_2 \leq n}} (\neg s(R, C_1, A) \vee \neg s(R, C_2, A)),$$

$$\Phi_{col}(s) = \bigwedge_{\substack{1 \leq C,A \leq n \\ 1 \leq R_1 < R_2 \leq n}} (\neg s(R_1, C, A) \vee \neg s(R_2, C, A)),$$

$$\Phi_{latin}(s) = \Phi_{cell}(s) \wedge \Phi_{row}(s) \wedge \Phi_{col}(s).$$

Auxiliary atoms $p(R, C, A, B)$ encode $l(R, C, A) \wedge g(R, C, B)$:

$$\begin{aligned} \Phi_{orth} = & \bigwedge_{1 \leq A,B \leq n} \left( \bigvee_{1 \leq R,C \leq n} p(R, C, A, B) \right) \\ & \wedge \bigwedge_{1 \leq R,C,A,B \leq n} (\neg p(R, C, A, B) \vee l(R, C, A)) \\ & \wedge \bigwedge_{1 \leq R,C,A,B \leq n} (\neg p(R, C, A, B) \vee g(R, C, B)) \\ & \wedge \bigwedge_{1 \leq R,C,A,B \leq n} (\neg l(R, C, A) \vee \neg g(R, C, B) \vee p(R, C, A, B)). \end{aligned}$$

The symmetry-breaking clauses are:

$$\Phi_{sym} = \left( \bigwedge_{1 \leq C \leq n} l(1, C, C) \right) \wedge \left( \bigwedge_{2 \leq R \leq n} l(R, 1, R) \right) \wedge \left( \bigwedge_{1 \leq C \leq n} g(1, C, C) \right).$$

The complete CNF is:

$$\Phi_{GL} = \Phi_{latin}(l) \wedge \Phi_{latin}(g) \wedge \Phi_{orth} \wedge \Phi_{sym}.$$

### A.3.2 SetLah Model for Q-Synth

The following SetLah! model is the core encoding used for the Q-Synth experiment in § 5.5.3.

```
ExtDOM LQ: INT.
ExtDOM PQ: INT.
ExtDOM NC: INT.

ExtPred l_qubit: (LQ).
ExtPred p_qubit: (PQ).
ExtPred cp: (PQ, PQ).
```

```
ExtPred ord: (NC, NC).
ExtPred cl: (NC, LQ, LQ).

Const t: INT.

[
    m(t, L, P)? :- l_qubit(L), p_qubit(P).

    \forall L. (\impl l_qubit(L) (\exists P. m(t, L, P))).
    \forall L, P1, P2.
        (\impl (\and m(t, L, P1) m(t, L, P2)) (= P1 P2)).

    \forall P, L1, L2.
        (\impl (\and m(t, L1, P) m(t, L2, P)) (= L1 L2)).
]

[
    s(t, P, P')? :- cp(P, P'), t>0.

    \forall P1, P1', P2, P2'.
        (\impl
            (\and s(t, P1, P1') s(t, P2, P2') (> t 0))
            (= P1 P2)
        ).

    \forall P1, P1', P2, P2'.
        (\impl
            (\and s(t, P1, P1') s(t, P2, P2') (> t 0))
            (= P1' P2')
        ).

    \forall P, P', L.
        (\impl (\and cp(P, P') l_qubit(L) s(t, P, P') (> t 0))
            (\and
                (\impl m(t-1, L, P)  m(t, L, P'))
                (\impl m(t-1, L, P') m(t, L, P))
                (\impl m(t,   L, P') m(t-1, L, P))
                (\impl m(t,   L, P)  m(t-1, L, P'))
            )
        ).
]

[
    st(t, P)  :- s(t, P, P').
    st(t, P') :- s(t, P, P').

    \forall P, L.
        (\impl
```

```
            (\and (\neg st(t, P)) t>0)
            (\and
                (\impl m(t-1, L, P) m(t, L, P))
                (\impl m(t, L, P) m(t-1, L, P))
            )
        ).
]

[
    mp(t, P)? :- p_qubit(P).

    \forall P.
        (\and
            (\impl mp(t, P) (\exists L. m(t, L, P)))
            (\impl (\exists L. m(t, L, P)) mp(t, P))
        ).

    \forall P, P'.
        (\impl s(t, P, P') (\or mp(t, P) mp(t, P'))).
]

[
    cp_aux(P, P') :- cp(P, P').
    cp_aux(P', P) :- cp(P, P').
]

[
    lp(t, L, L')? :- cl(NC, L, L'), m(t, L, P), m(t, L', P').

    \forall L, L', P, P'.
        (\impl
            (\exists NC. cl(NC, L, L'))
            (\and
                (\impl
                    (\and m(t, L, P) m(t, L', P') cp_aux(P, P'))
                    lp(t, L, L')
                )
                (\impl
                    (\and m(t, L, P) m(t, L', P') (\neg cp_aux(P, P')))
                    (\neg lp(t, L, L'))
                )
            )
        ).
]

[
    clkey(t, I) :- cl(I, L1, L2).
]
```

```
[
    ac(t, I)? :- clkey(t, I).
    c(t, I)?  :- clkey(t, I).
    dc(t, I)? :- clkey(t, I).

    (\partition Set_clkey {{Set_c Set_ac Set_dc}}).

    (\neg (\exists I. ac(0, I))).

    \forall I, J.
        (\impl (\and ord(J, I) c(t, I)) (\or ac(t, J) c(t, J))).

    \forall I, J.
        (\impl (\and ord(I, J) c(t, I)) (\or dc(t, J) c(t, J))).

    \forall I.
        (\impl (\and ac(t, I) t>0) (\or c(t-1, I) ac(t-1, I))).

    \forall I, J.
        (\impl (\and ac(t, I) ord(J, I)) ac(t, J)).

    \forall I.
        (\impl (\and dc(t-1, I) t>0) (\or c(t, I) dc(t, I))).

    \forall I, J.
        (\impl (\and dc(t, I) ord(I, J)) dc(t, J)).

    \forall I.
        (\impl
            (\and dc(t, I) cl(I, L1, L2))
            (\or
                (\neg lp(t, L1, L2))
                (\exists J. (\and ord(J, I) dc(t, J)))
            )
        ).

    \forall I.
        (\impl (\and c(t, I) cl(I, L1, L2)) lp(t, L1, L2)).
]

[
    asm(t)?.
    (\impl asm(t) (\forall I. (\neg dc(t, I)))).
    (\impl (\forall I. (\neg dc(t, I))) asm(t)).
]
```