

UC San Diego

UC San Diego Electronic Theses and Dissertations

Title

Syntactic and Semantic Guidance for Automated Program Reasoning

Permalink

<https://escholarship.org/uc/item/3rn6282b>

ISBN

9798193291800

Author

Park, Kanghee

Publication Date

2026-10-13

Peer reviewed|Thesis/dissertation

UNIVERSITY OF CALIFORNIA SAN DIEGO

Syntactic and Semantic Guidance for Automated Program Reasoning

A dissertation submitted in partial satisfaction of the
requirements for the degree Doctor of Philosophy

in

Computer Science

by

Kanghee Park

Committee in charge:

Professor Loris D'Antoni, Chair
Professor Samuel Buss
Professor Ranjit Jhala
Professor Nadia Polikarpova

2026

Copyright
Kanghee Park, 2026
All rights reserved.

The Dissertation of Kanghee Park is approved, and it is acceptable in quality and form for publication on microfilm and electronically.

University of California San Diego

2026

DEDICATION

To my beloved parents, whose unconditional support and sacrifices gave me the opportunity to focus entirely on my studies. Everything I achieve is built on the foundation of your love.

To my advisor, Loris D'Antoni, for your invaluable mentorship. Thank you for your endless guidance, insightful advice, and wisdom—not only in my research but in life.

To my friends across South Korea, Madison, and San Diego, whose emotional support, warmth, and encouragement kept me going through every high and low.

TABLE OF CONTENTS

Dissertation Approval Page	iii
Dedication	iv
Table of Contents	v
List of Figures	ix
List of Tables	xii
Acknowledgements	xiii
Vita	xiv
Abstract of the Dissertation	xv
Chapter 1 Introduction	1
1.1 Thesis and Scope	2
1.2 Program Syntax and Semantics	3
1.3 A Layered View of Program Reasoning	5
1.4 Dissertation Overview	7
Part I Guiding Syntactic Constraints	10
Chapter 2 Syntactic Control of LLM Generation	11
2.1 Background	12
2.1.1 Formal Syntax	12
2.1.2 Grammar-Constrained Decoding	14
2.2 Efficient and Flexible Grammar-Constrained Decoding	15
2.3 Approach Overview	17
2.3.1 Lexing and Parsing	17
2.3.2 Using Parsing for Constrained Decoding	19
2.3.3 Our Approach	19
2.4 Offline Token Preprocessing	20
2.4.1 Lexer Preprocessing	20
2.4.2 Realizable Terminal Sequences	26
2.4.3 Parser Preprocessing	28
2.5 Online Token Masking	30
2.6 Experiments	33
2.7 Related Work	37
2.8 Conclusion	39
Chapter 3 Grammar-Aligned Decoding	41
3.1 Grammar-Aligned Decoding	42

3.2	Adaptive Sampling with Approximate Expected Futures	47
3.3	Experiments	52
3.4	Will ASAp still be more aligned than GCD after fine-tuning?	59
3.4.1	Experimental setup for fine-tuning	59
3.4.2	Additional Results	60
3.5	Related Work	61
3.6	Conclusion	63
Part II	Specification Synthesis	64
Chapter 4	Synthesizing Specifications	65
4.1	Problem Definition	69
4.2	An Algorithm for Synthesizing Best $\mathcal{L}$ -conjunctions	75
4.2.1	Positive and Negative Examples	75
4.2.2	Soundness and Precision	78
4.2.3	Synthesizing One Most Precise $\mathcal{L}$ -property	81
4.2.4	Synthesizing a Most-Precise $\mathcal{L}$ -conjunction	86
4.2.5	Completeness	90
4.3	Implementation	92
4.4	The Encoding of our Primitives in SKETCH	94
4.5	Evaluation	96
4.5.1	Application 1: Specification Mining	96
4.5.2	Application 2: Synthesizing Algebraic Specifications for Modular Synthesis	102
4.5.3	Application 3: Automating Sensitivity Analysis	105
4.5.4	Application 4: Enabling New Abstract Domains	107
4.5.5	Further Analysis of SPYRO's Performance	110
4.6	Related Work	112
4.7	Conclusion	116
Chapter 5	Specifications for Nondeterministic Programs	118
5.1	Motivating Examples	123
5.1.1	Over-Approximate Reasoning	124
5.1.2	Under-Approximate Reasoning	125
5.2	Strongest Consequences and Weakest Implicants	126
5.3	Relation to Program Logics	130
5.3.1	Relation to Hoare Logic	130
5.3.2	Relation to Reverse Hoare Logic and Incorrectness Logic	134
5.4	Counterexample-Guided Inductive Specification Synthesis	138
5.4.1	Synthesis from Positive and Negative Examples	138
5.4.2	Synthesizing One Strongest $\mathcal{L}$ -Consequence and One Weakest $\mathcal{L}$ -Implicant	144
5.4.3	Synthesizing a Best $\mathcal{L}$ -Conjunction and $\mathcal{L}$ -Disjunction	149
5.5	Counterexample-Guided Quantifier Instantiation	152

5.6	Implementation	159
5.7	Evaluation	162
5.7.1	Testing ASPIRE with Simple Programs	162
5.7.2	Application 1: Incorrectness Reasoning	164
5.7.3	Application 2: Reasoning about Concurrent Programs	169
5.7.4	Application 3: Solving Two-Player Games	170
5.8	Related Work	173
5.9	Conclusion	176
Part III	Modular Program Reasoning	178
Chapter 6	Modular System Synthesis	179
6.1	Illustrative Example	183
6.1.1	A Modular TicketVendor Implementation	183
6.1.2	The Input of Modular TicketVendor Synthesis	184
6.1.3	The Output of Modular TicketVendor Synthesis	185
6.1.4	Benefits of Modular System Synthesis	187
6.2	Modular System Design	188
6.2.1	Implementation-specific Specifications	189
6.2.2	Implementation-agnostic Specifications	190
6.3	Synthesis in Modular System Synthesis	191
6.3.1	Synthesis of Implementations	191
6.3.2	Synthesis of Implementation-agnostic Specifications	192
6.4	Implementation and Case-Study Evaluation	194
6.4.1	Implementation	194
6.4.2	Ticket-vendor Case Study	194
6.4.3	Case Studies from Mariano et al. [101]	200
6.4.4	Additional Case Studies Based on Mariano et al. [101]	202
6.4.5	Implementation Synthesis with JLibSketch	204
6.4.6	Limitations of MOSSKIT	206
6.5	Related Work	209
6.6	Conclusion	210
Chapter 7	Extracting Reusable Proof Structure	212
7.1	Background	213
7.1.1	The Calculus of Constructions	213
7.1.2	Proof Terms and Provability	216
7.1.3	Type Inference by Unification	217
7.2	Overview	218
7.2.1	Generalization by Abstraction and Unification	218
7.2.2	Choosing an Abstraction Boundary	220
7.3	Abstraction Synthesis	221
7.4	Top-Down Enumeration of Abstract Proof Terms	224
7.4.1	Safe Metavariable Refinements	224

7.4.2	Potentially Unsafe Metavariable Refinements	229
7.4.3	The Main Loop	232
7.5	Dependency-Guided Metavariable Instantiation	236
7.5.1	Metavariable Dependency Graph	237
7.5.2	Componentwise Safe Instantiation	238
7.5.3	Pruning the Enumeration	239
7.6	Related Work	242
7.7	Conclusion	243
Chapter 8	Conclusion	245
8.1	Summary of Contributions	245
8.2	Directions for Future Work	247
8.3	Closing Perspective	249
Appendix A	Specification Synthesis Evaluation Details	250
A.1	Specification-Mining Benchmarks	250
A.2	Synquid	252
A.3	Algebraic-Specification Benchmarks	258
A.4	Sensitivity-Analysis Benchmarks	263
A.5	Bit-Vector Polyhedra Benchmarks	269
A.6	Detailed Evaluation for Each Application	270
Appendix B	Specification Synthesis of Nondeterministic Program Evaluation Details . .	275
B.1	Type I Test Set: Mining Under-Approximated Specifications	275
B.2	Type II Test Set: Simple nondeterministic programs	279
B.3	Application 1: Incorrectness Reasoning	285
B.4	Application 2: Reasoning about Concurrent Programs	288
B.5	Application 3: Solving two-player games	293
Bibliography		297

LIST OF FIGURES

Figure 2.1.	Illustrative example of the approach implemented in GREATGRAMMA. . .	18
Figure 2.2.	A lexing transducer $\mathcal{T}_{\mathcal{A}}$ derived from FSA $\mathcal{A}$ in Fig. 2.1.	23
Figure 2.3.	Detokenizing transducer for vocabulary $\mathcal{V} = \{a, b, c, ab, ac, aba\}$	24
Figure 2.4.	A determinized token-level lexing transducer $\mathcal{T}_{\mathcal{A} \circ \mathcal{V}}$, which is formed by composing $\mathcal{T}_{\mathcal{V}}$ from Fig. 2.3 and $\mathcal{T}_{\mathcal{A}}$ from Fig. 2.2.	24
Figure 3.1.	CFG $\mathcal{G}_{sk}$ over tokens $\Gamma = \{0, 1\}$, written in Backus-Naur form (BNF) notation.	42
Figure 3.2.	Fragment of the conditional model distribution P for Ex. 1 depicted as a trie.	43
Figure 3.3.	Illustration of the trie built by ASAp after sampling 00000 as the first string (left) and after sampling 11111 as the second string (right).	48
Figure 3.4.	A SLIA problem with the explicitly defined grammar.	54
Figure 3.5.	An INV-BV problem with the implicitly defined grammar.	55
Figure 3.6.	$KL(Q_{ASAp} P)$ and $KL(Q_{GCD} P)$	56
Figure 3.7.	Expectations of $\tilde{Q}_{ASAp}$, $\tilde{Q}_{GCD}$, and P	57
Figure 3.8.	Scatter plots of $\tilde{Q}_{ASAp}$ and $\tilde{Q}_{GCD}$ vs. expectations of P after 2,000 samples.	58
Figure 4.1.	The role of examples and CHECKSOUNDNESS.	77
Figure 4.2.	Possible results produced by CHECKPRECISION.	80
Figure 4.3.	SKETCH generator genAP for the productions of nonterminal AP in the DSL in Equation 4.2.	95
Figure 4.4.	SKETCH code for CHECKPRECISION with positive and negative examples.	95
Figure 4.5.	$\mathcal{L}$ -properties synthesized by SPYRO. Some properties are rewritten as implications for readability.	100
Figure 4.6.	Each subfigure illustrates a bit-vector formula and the most precise bit-vector polyhedron computed by SPYRO (i.e., the inequalities above the formulas).	108
Figure 4.7.	Evaluation of the running time of SPYRO for different input sizes and optimizations.	112

Figure 5.1.	An over-approximation synthesis problem for identifying properties of <code>modhash</code> that hold for any choice of input x	123
Figure 5.2.	Synthesized $\mathcal{U}$ -implicants.	125
Figure 5.3.	The bounded domains of variables in problem Section 5.1.1	160
Figure 5.4.	Synthesized $\mathcal{U}$ -implicants	164
Figure 5.5.	A LOUD problem synthesizing <i>weakest possible preconditions</i> that cause a bug in the <code>remhash</code> function (i.e., returning a negative number).	165
Figure 5.6.	The DSL for <code>arit1</code>	168
Figure 5.7.	The DSLs and synthesized properties for the <code>philo</code> benchmark.	170
Figure 6.1.	Organization of the modular <code>TicketVendor</code> synthesis problem.	186
Figure 6.2.	Alternative implementation of the <code>Queue</code> module using list primitives instead of two stacks.	188
Figure 6.3.	Implementation of <code>snoc</code> supplied to SPYRO. Returning a value from a function is done by storing the value into a reference parameter of the function.	195
Figure 6.4.	Grammar for the domain-specific language in which SPYRO is to express an extracted <code>List</code> property.	195
Figure 6.5.	JLIBSKETCH sketch of <code>enq</code>	197
Figure 6.6.	JLIBSKETCH sketch of <code>enq</code> and <code>deq</code>	205
Figure 6.7.	JLIBSKETCH rewrite class <code>Stack</code> for the synthesis of <code>Queue</code>	207
Figure 6.8.	JLIBSKETCH harness corresponding to the implementation-specific specification ϕ_{imp}^{Queue} for <code>Queue</code> operations.	208
Figure 7.1.	The Calculus of Constructions with Non-cumulative Universes.	214
Figure 7.2.	Generalization of <code>even_numbers_not_coprime</code> by subterm abstraction and unification.	219
Figure A.1.	Each green and red cell represents a solution in 4-bit unsigned modular arithmetic of the indicated formula.	270
Figure B.1.	Resource allocator for two threads	291

Figure B.2.	Two threads in race2	292
-------------	----------------------------	-----

LIST OF TABLES

Table 2.1.	Offline and online processing times. X denotes when SYNCODE masked incorrect tokens.	40
Table 3.1.	Correctness of solutions for different algorithms.	59
Table 5.1.	Algorithm 10 run table	146
Table 5.2.	Algorithm 14 run table	155
Table 5.3.	Applications 1 to 3.	167
Table A.1.	Evaluation results of SPYRO for SyGuS and LIA specification-mining.	271
Table A.2.	Evaluation results of SPYRO for data structures and arithmetic specification-mining.	272
Table A.3.	Evaluation results of SPYRO for modular specification synthesis.	273
Table A.4.	Evaluation results of SPYRO for sensitivity analysis and abstract domains. .	274
Table B.1.	Evaluation of ASPIRE for mining specifications.	277
Table B.2.	Evaluation results of Type II test set.	281

ACKNOWLEDGEMENTS

I would like to acknowledge Professor Loris D’Antoni for his support as the chair of my committee. Through multiple drafts and many long nights, his guidance has proved to be invaluable. I would also like to thank the other members of my committee—Professor Samuel Buss, Professor Ranjit Jhala, and Professor Nadia Polikarpova—for their time, insightful feedback, and thoughtful questions, which helped sharpen and strengthen this dissertation.

Chapter 2, in full, is based on the paper from Kanghee Park, Timothy Zhou, and Loris D’Antoni, “Flexible and Efficient Grammar-Constrained Decoding” published in International Conference on Machine Learning, pp. 48262-48275. PMLR, 2025. The dissertation author is the primary investigator and author of this paper.

Chapter 3, in full, is based on the paper from Kanghee Park, Jiayu Wang, Taylor Berg-Kirkpatrick, Nadia Polikarpova, Loris D’Antoni. “Grammar-Aligned Decoding” published in Neural Information Processing Systems 37, pp. 24547-24568. 2024. The dissertation author is the primary investigator and author of this paper. The author sincerely thanks Paweł Parys for finding and fixing errors in the proof of Theorem 3.2.2.

Chapter 4, in full, is based on the paper from Kanghee Park, Loris D’Antoni, and Thomas Reps “Synthesizing Specifications” published in Proceedings of the ACM on Programming Languages 7(OOPSLA2), pp. 1787-1816. 2023. The dissertation author is the primary investigator and author of this paper.

Chapter 5, in full, is based on the paper from Kanghee Park, Xuanyu Peng, and Loris D’Antoni “LOUD: Synthesizing Strongest and Weakest Specifications” published in Proceedings of the ACM on Programming Languages 9(OOPSLA1), pp. 956-983. 2025. The dissertation author is the primary investigator and author of this paper.

Chapter 6, in full, is based on the paper from Kanghee Park, Keith J.C. Johnson, Loris D’Antoni, and Thomas Reps “Modular System Synthesis” published in 2023 Formal Methods in Computer-Aided Design, pp. 257-267. IEEE, 2023. The dissertation author is the primary investigator and author of this paper.

VITA

- 2021 Bachelor of Science, Bachelor of Engineering,
Pohang University of Science and Technology
- 2021–2022 Teaching Assistant, Department of Computer Science
University of Wisconsin Madison
- 2022–2024 Research Assistant, University of Wisconsin Madison
- 2024 Master of Science, University of Wisconsin Madison
- 2024–2026 Research Assistant, University of California San Diego
- 2026 Doctor of Philosophy, University of California San Diego

PUBLICATIONS

Kanghee Park, Loris D’Antoni, and Thomas Reps, Synthesizing Specifications, *Proceedings of the ACM on Programming Languages* 7(OOPSLA2), pp. 1787-1816. 2023.

Kanghee Park, Keith J.C. Johnson, Loris D’Antoni, and Thomas Reps. Modular System Synthesis. In *2023 Formal Methods in Computer-Aided Design*, pp. 257-267. IEEE, 2023.

Kanghee Park, Jiayu Wang, Taylor Berg-Kirkpatrick, Nadia Polikarpova, Loris D’Antoni. Grammar-Aligned Decoding. *Advances in Neural Information Processing Systems* 37, pp. 24547-24568. 2024.

Kanghee Park, Xuanyu Peng, Loris D’Antoni. LOUD: Synthesizing Strongest and Weakest Specifications. *Proceedings of the ACM on Programming Languages* 9(OOPSLA1), pp. 956-983. 2025.

Kanghee Park, Timothy Zhou, Loris D’Antoni. Flexible and Efficient Grammar-Constrained Decoding. In *International Conference on Machine Learning*, pp. 48262-48275. PMLR, 2025.

ABSTRACT OF THE DISSERTATION

Syntactic and Semantic Guidance for Automated Program Reasoning

by

Kanghee Park

Doctor of Philosophy in Computer Science

University of California San Diego, 2026

Professor Loris D’Antoni, Chair

In the past few years, the landscape of software development has been reshaped. Large Language Models have made generating code, proofs, tests, and documentation cheap and abundant. A generated artifact may look plausible yet be subtly wrong. At the pace generation now runs, manual review cannot close the gap. Generation is cheap; review is the bottleneck. This dissertation investigates how to establish formal guarantees for code at scale, with minimal human oversight.

The first part of this dissertation studies mechanisms for enforcing strict syntactic constraints during LLM generation. In particular, we focus on Grammar-Constrained Decoding (GCD), which integrates formal grammar parsers directly into the decoding process. To make this

approach practical for modern LLMs with large vocabularies, we develop techniques to improve the efficiency of grammar-aware decoding. We also identify a fundamental theoretical limitation of GCD: although it guarantees syntactic validity, it can distort the model’s learned probability distribution. To address this limitation, we introduce the Grammar-Aligned Decoding (GAD) problem, which seeks to align the sampling distribution with the desired structural constraints.

While syntax can often be captured by explicit grammars, semantic correctness is substantially harder to specify. The second part of this dissertation addresses the problem of synthesizing precise semantic specifications from source code. We introduce a counterexample-guided synthesis algorithm that generates sound and precise program specifications within a target domain-specific language. We further extend this algorithm to nondeterministic programs, enabling the synthesis of both over-approximate and under-approximate specifications.

The third part of this dissertation builds on these component-level specifications to synthesize complex systems compositionally. Rather than verifying or generating a large system monolithically, our approach constructs larger programs from existing sub-programs and verified lemmas. Each synthesis task is reduced to a smaller subproblem of the same general form, keeping the search space and proof obligations bounded at the component level. Together, these techniques provide a robust path toward the scalable, specification-guided construction of verified software systems.

Chapter 1

Introduction

Automating programming has long been an aspiration of computer science. Over the past few years, large language models (LLMs) have steadily moved work away from writing every line by hand. A single prompt now returns what once took days to compose: a working function, a data pipeline, even an entire prototype. Increasingly, programmers describe what they want and read what comes back.

Cheap generation does not finish the job. A program can compile, run, and pass a demo while quietly doing something other than what was asked, and someone still has to catch the gap. Automation has changed which of these tasks is the bottleneck: producing a plausible candidate is now cheap, while deciding whether to trust one remains as expensive as ever. When generation outpaces human review, confidence becomes the scarce resource.

Testing checks only the inputs someone thought to try. Code review depends on a human reading at human speed. Neither of them keeps pace with a generator that produces candidates faster than anyone can read them, and neither says what has been established when a candidate passes. Cheap generation calls for checks that are themselves automatic and that, when they succeed, actually guarantee something.

This is what formal methods offer. Formal methods interpret programs as formal objects, so that claims about them can be checked mechanically rather than judged by inspection. A parser can decide whether a generated string is a well-formed program; a verifier can decide

whether an implementation satisfies a precise description of its intended behavior, called a *specification*. Such checks are especially valuable in a generative setting: the generator itself need not be trusted, so long as every accepted result is validated by a sound, automatic procedure.

In practice, however, formal methods have long faced problems of their own. First, someone must say what *correct* means. Writing a specification requires both domain knowledge and formal logic. This rare combination is one reason formal methods have long remained the province of specialists. Second, rejecting bad candidates does not make good candidates likely. If the generator rarely produces anything acceptable, a perfect checker only confirms failure faster. Better is to steer generation so that unacceptable candidates are never produced at all. Third, the space of programs, specifications, and proofs that automated tools must explore grows quickly with the size of the system, so that even fully mechanical checks can become intractable.

Historically, this third problem was itself the bottleneck of formal methods. Today, however, search is no longer where the main difficulty lies. Recent advances in large language models have driven this shift: LLMs can now search for certificates of correctness at a scale not previously possible. The bottleneck has moved to *controlling* what the automation is asked to reason about and *structuring* the spaces in which it searches.

1.1 Thesis and Scope

The common idea across this dissertation is that *formal structure makes large search and reasoning spaces tractable*. A small, human-provided formal object, placed between the automation and a large space, restricts attention to the part that matters. It turns an enormous task into one an automated tool can finish and a sound checker can verify.

This dissertation addresses problems across different layers of automated program reasoning. Part I constrains what an LLM emits so that generated code respects a target grammar. Part II reduces the manual effort of writing formal specifications by synthesizing them directly from code. Part III exploits modularity to keep each verification subproblem within the reach of

automated tools, using module contracts that hide implementation detail and proof abstractions that expose reusable logical structure.

The scope is deliberately narrow. Scalable verification sometimes evokes fully autonomous, end-to-end tools for arbitrary code, but the mechanisms developed here target specific bottlenecks under concrete assumptions. Structural guidance comes from the developer: grammars for permitted output languages, domain-specific languages for relevant semantic properties, module decompositions for architectural boundaries. The resulting guarantees are necessarily relative to these inputs, the underlying formal semantics, and the capabilities of the concrete solvers or proof assistants.

These formal objects share a common role: each is small, human-provided, and enough to turn an unbounded task into one an automated tool can finish and a sound checker can verify. What they constrain differs: grammar operates on syntax, the others on semantics. Section 1.3 examines each in turn.

1.2 Program Syntax and Semantics

Before turning to the framework the dissertation develops, this short section fixes the vocabulary used throughout to talk about programs: what they look like, and what they mean. Readers familiar with formal programming-language theory can safely skim.

Programs are governed by two essential aspects: *syntax*, which dictates the rules for forming valid structural constructs, and *semantics*, which assigns precise computational behavior to those constructs.

The term *syntax* often carries two distinct meanings depending on the context of the discussion. In formal programming language theory, syntax typically refers to the *abstract syntax*, the hierarchical, tree-like constructs used to build a program mathematically. For instance, in the Simply Typed Lambda Calculus (STLC), the abstract syntax of a program expression e is

inductively defined by the following grammar:

$$e := x \mid \lambda x:\tau. e \mid e_1 e_2$$

This notation specifies that an expression e can either be a variable x , a function abstraction $\lambda x:\tau. e$ that binds a variable x of type τ in a sub-expression e , or an application $e_1 e_2$ that applies an expression e_1 to another expression e_2 . For example, an expression applying a function f to a bound variable x is written as $\lambda x:\tau. f x$. Crucially, this grammar defines a recursive tree structure rather than a flat text sequence. The application $f x$ is mathematically a tree node with two child leaves: the variables f and x .

In practice, programmers do not interface with a computer by directly constructing these abstract syntax trees. They write programs by typing linear sequences of characters, and a *parser* is responsible for determining whether a given string is syntactically valid and, if so, converting it into the corresponding tree structure. In this practical context, syntax refers to the *concrete syntax*. The set of all valid sequences of characters that can be successfully parsed into well-formed programs is a *language*.

Once a program is deemed syntactically valid, its execution behavior is dictated by its *semantics*, which provides a rigorous mathematical meaning for each syntactic construct. Semantics can be defined in various ways, such as operational, denotational, or axiomatic. This dissertation frequently models the semantics of a program expression e as a mathematical relation $\llbracket e \rrbracket(\sigma, \sigma')$, indicating that executing the syntactically valid program e on an initial input state σ successfully terminates and produces an output state σ' .

For example, consider a simple imperative program expression e given by the assignment $y := x + 1$. A program state can be understood as a mapping from variable names to their values. If the initial state σ assigns the value 5 to variable x , that is $\sigma = \{x \mapsto 5\}$, executing e updates the environment to produce a final state $\sigma' = \{x \mapsto 5, y \mapsto 6\}$. In this case, the relation $\llbracket y := x + 1 \rrbracket(\sigma, \sigma')$ holds.

1.3 A Layered View of Program Reasoning

The dissertation moves through four kinds of formal objects. One of them, the grammar, operates on syntax; the other three operate on semantics. Together they span the layers of automated program reasoning, and each addresses the same basic tension between an expressive search space and a tractable reasoning problem.

Syntactic Structure

A program must be a well-formed artifact before its behavior can be considered. In traditional software development, a parser enforces this syntactic structure *post-hoc*: given a completed program, it either produces an abstract syntax tree (AST) or raises a syntax error. During generation, the underlying grammar can intervene much earlier. By evaluating the prefix at each step, it identifies valid continuations and prevents the generator from selecting tokens that lead to ungrammatical states.

Enforcing a grammar can also change what the underlying model is effectively sampling. Removing invalid next tokens and renormalizing the remaining probabilities guarantees grammaticality, but it does not preserve the model’s relative preferences among complete valid outputs. Syntactic guidance therefore has two obligations. It should rule out malformed artifacts efficiently, and it should make clear how the intervention changes the generator’s distribution. The first part of this dissertation studies both obligations.

Semantic Specification

Syntactic validity ensures an artifact can be parsed, but guarantees nothing about its behavior. Semantic reasoning instead focuses on execution properties: what outputs may follow from an input, what must always hold, what can go wrong, and what conditions make a result possible.

Specifications do not prescribe how to compute a desired result; they define what constitutes the correct behavior of a program. A specification is not necessarily equivalent to the

full semantics of a concrete implementation. For instance, a specification for a list-sorting algorithm might require that elements with smaller keys appear earlier and that the output is a valid permutation of the input, while leaving the relative order of elements with equal keys unspecified. This choice permits both stable and unstable sorting implementations under the same contract.

This dissertation treats a user-provided domain-specific language (DSL) as a formal description of the abstraction boundary. The DSL defines the vocabulary of interest, including the predicates, operators, and formula structures that may appear in a synthesized specification. Synthesis within the DSL then reduces to a search problem whose difficulty is driven by the size of the property space and by the strength of the semantic reasoning available to prune it.

For deterministic programs, the central question is what strongest expressible facts follow from a computation. For nondeterministic programs, one must distinguish facts that hold for every execution from facts witnessed by at least one execution. A useful semantic guidance mechanism must support both over-approximate statements about what must hold and under-approximate statements about what may occur.

Module Interfaces

Large systems are rarely understood or constructed as monolithic programs. They are divided into modules so that one component can be changed without requiring every client to understand its representation. A module interface, or contract, establishes the correctness boundary between an implementation and its clients: the implementation must provide the promised behavior, and clients may rely on that behavior but not on hidden details.

Information hiding provides a crucial *trust boundary* for program reasoning. Without it, higher-level tasks must analyze all accumulated implementation details, leading to search space explosion. Reasoning exclusively through trusted, implementation-agnostic contracts allows automated tools to bound their search space to the current layer. Modularity is thus a key method for controlling formal reasoning rather than a passive software-engineering habit.

Correctness Proof

Given a specification and an implementation, verifying correctness requires a formal certificate showing that the code satisfies its specification. Formal methods can check such certificates soundly, so any accepted certificate is trustworthy. The remaining challenge is producing one: automated search is expensive and often incomplete.

Classical approaches combine enumerative search over candidate proofs with symbolic reasoning through solvers, and both scale poorly as programs and their underlying theories grow more complex [12, 34, 145, 93]. Frontier LLMs have begun to change this, writing formal proofs of program correctness directly in interactive proof assistants such as Rocq [149] and Lean [109].

Correctness alone is not enough. A proof that type-checks may still be too large to audit or too specialized to reuse. Machine-generated proofs frequently show these problems, resolving theorems through long sequences of specialized operations without factoring out reusable lemmas. A checked proof can indeed be more general than the specific theorem it establishes. Selected pieces of the proof can be replaced with placeholders, and the surrounding argument still works for any pieces that fit those placeholders. The result is a template that adapts to any new setting where the placeholders can be proved.

1.4 Dissertation Overview

The dissertation has three parts. The first develops grammars, the second develops specifications, and the third develops both module interfaces and proof abstractions.

Part I: Guiding Syntactic Constraints.

The first part asks how a formal language can participate directly in LLM decoding.

Chapter 2 introduces GREATGRAMMA, a grammar-constrained decoding algorithm that precomputes, for each lexer state, the terminal sequences realizable by the LLM vocabulary and uses that representation for efficient online masking. With a clean modular structure, the algorithm makes offline preprocessing substantially faster than prior work while retaining low

online overhead.

Chapter 3 shows that token masking, though it guarantees grammaticality, can distort the model’s distribution over complete valid outputs. The chapter formalizes this problem as *grammar-aligned decoding*: sampling from the LLM distribution conditioned on grammaticality. It introduces ASAp, an adaptive algorithm that estimates the expected future grammaticality of each prefix and converges toward the aligned target.

Part II: Specification Synthesis.

The second part addresses semantic specifications, which cannot be captured by an off-the-shelf grammar.

Chapter 4 presents SPYRO, a framework for synthesizing sound, most-precise specifications in a user-supplied DSL. Rather than requiring a programmer to state a formal specification, SPYRO searches for strongest consequences of a semantic query that are expressible in the chosen language. Its counterexample-guided algorithm synthesizes individual strongest properties, distinguishes negative examples that must be rejected from those that may be rejected, and combines incomparable properties into a best conjunction.

Chapter 5 extends specification synthesis to nondeterministic programs and to both over- and under-approximate reasoning. The LOUD framework supports strongest over-approximations, which describe behavior that must hold, and weakest under-approximations, which describe behavior that may occur for some execution. Its implementation, ASPIRE, combines the specification synthesis loop with counterexample-guided quantifier instantiation to handle existentially quantified queries.

Part III: Modular Program Reasoning.

The third part builds on component-level specifications to construct and reason about larger systems.

Chapter 6 presents MOSS, a framework for modular system synthesis based on the information hiding principle. The framework identifies two complementary synthesis problems:

synthesizing a module implementation from lower-level contracts, and synthesizing a contract from a completed module. Alternating these operations allows a system to be constructed layer by layer while hiding lower-level representation choices and keeping each synthesis subproblem bounded to the current layer.

Chapter 7 addresses proof reuse. It formalizes *proof abstraction* as the replacement of selected subterms of a checked, dependently typed proof with metavariables, and defines a preorder that captures refinements known to preserve provability. A constructor-guided, top-down enumerator distinguishes refinements that preserve provability without branching from refinements whose safety depends on additional commitments, and a metavariable dependency graph identifies choices that must be resolved together. From a single checked proof, the enumerator produces typed templates whose remaining obligations state exactly what must be proved to instantiate the same argument elsewhere.

Part I

Guiding Syntactic Constraints

Chapter 2

Syntactic Control of LLM Generation

Traditional program synthesis has long relied on rigorous enumerative search techniques applied directly to the Abstract Syntax Tree (AST). By methodically exploring the space of possible program structures, these classical approaches ensure that every generated candidate strictly adheres to the underlying grammatical rules of the target language. Because the search space itself is defined by the production rules of the grammar, any program discovered through this technique naturally obeys all syntactic constraints. In this paradigm, producing structurally invalid code is fundamentally impossible, as the generation engine is structurally bound by the language’s syntax.

In contrast, code generation driven by LLMs operates on a fundamentally different, statistical paradigm. Rather than constructing a program via hierarchical trees and structural rules, LLMs generate code autoregressively as a linear sequence of tokens. Because their outputs are determined by statistical probabilities derived from vast training corpora rather than explicit structural boundaries, LLMs possess no inherent guarantee of syntactic correctness. While they are highly adept at mimicking the syntax of widely used languages like Python, they are essentially guessing the next most likely text fragment, leaving them prone to generating structurally invalid code.

This architectural difference becomes a significant liability when attempting to generate code for non-standard grammars or proprietary Domain-Specific Languages (DSLs). When

an LLM is tasked with writing code in a custom language that is scarcely represented in its training data, its statistical intuition fails, frequently resulting in syntax errors and compilation failures. Without the strict, grammar-driven boundaries provided by AST-based enumerative search, the model struggles to internalize and enforce unfamiliar syntactic constraints. To address this limitation, recent research has explored using parsers to control the LLM generation process, thereby forcing the model’s output to strictly follow defined syntactic constraints.

2.1 Background

2.1.1 Formal Syntax

Formally, the syntax of a program is defined as a set of valid strings, which we refer to as a *language*. To ground this mathematically, we first define an *alphabet* Σ as a finite set of symbols. A *string* is a finite sequence of symbols drawn from Σ , with the set of all possible strings denoted by Σ^* . Consequently, a language $\mathcal{L}$ is simply a subset of Σ^* . To specify and reason about these languages, formal language theory categorizes them into different classes, such as those defined by regular expressions and context-free grammars (CFGs).

Regular Expression

Let Σ be a finite alphabet of characters. A *regular expression* r over Σ is a syntactic construct inductively defined by the following grammar:

$$r ::= \emptyset \mid \varepsilon \mid c \mid r_1 r_2 \mid r_1 \cup r_2 \mid r^*$$

where $c \in \Sigma$ represents a literal character, $r_1 r_2$ denotes concatenation, $r_1 \cup r_2$ (often denoted as $r_1 \mid r_2$) denotes alternation (or union), and r^* denotes the Kleene star operation (representing zero or more repetitions of r).

Every regular expression r describes a formal language, denoted by $\mathcal{L}(r) \subseteq \Sigma^*$, which is the set of all strings that match the pattern. The language is defined recursively based on its

syntax: $\mathcal{L}(\emptyset) = \emptyset$, $\mathcal{L}(\varepsilon) = \{\varepsilon\}$, $\mathcal{L}(c) = \{c\}$, $\mathcal{L}(r_1 r_2) = \{w_1 w_2 \mid w_1 \in \mathcal{L}(r_1), w_2 \in \mathcal{L}(r_2)\}$, $\mathcal{L}(r_1 \cup r_2) = \mathcal{L}(r_1) \cup \mathcal{L}(r_2)$, and $\mathcal{L}(r^*) = \bigcup_{i \geq 0} \mathcal{L}(r)^i$. While regular expressions provide a declarative way to specify these sets of strings, it is often necessary to evaluate them operationally, which brings us to the definition of automata.

Finite-State Automata

A *finite-state automaton* (FSA) is defined as a tuple $\mathcal{A} = (\Sigma, Q, q_0, \delta, F)$ where Σ is the input alphabet, Q is the set of states, $q_0 \in Q$ is the initial state, $\delta \subseteq Q \times (\Sigma \cup \{\varepsilon\}) \times Q$ is the set of transitions, and $F \subseteq Q$ is the set of accepting states. Each transition (q, c, q') , also denoted by $q \xrightarrow{c} q'$, indicates that, from state q , upon reading the input symbol c , the automaton transitions to state q' .

Given a transition relation δ of automaton $\mathcal{A}$, the extended transition $\delta^* \subseteq Q \times \Sigma^* \times Q$ is the smallest relation defined by (i) $(q, \varepsilon, q) \in \delta^*$ and (ii) $(q, wc, q') \in \delta^*$ whenever $(q, w, q'') \in \delta^*$ and $(q'', c, q') \in \delta$ for some $q'' \in Q$. We also denote $(q, w, q') \in \delta^*$ by $q \xrightarrow{w} q'$. A string $w \in \Sigma^*$ is accepted by automaton $\mathcal{A}$ when there exists $q' \in F$ such that $q \xrightarrow{w}^* q' \in \delta^*$.

However, in this dissertation, we use the term *accepted* in a broader sense than its standard definition. We say that a string w is *acceptable* in state q if there exists q' such that $q \xrightarrow{w} q' \in \delta^*$, meaning that a valid transition for w exists from q .

Context-Free Grammar

A *context-free grammar* (CFG) $\mathcal{G}$ is a tuple $(\Gamma, \mathcal{N}, S, \mathcal{R})$ where Γ is a finite set of terminal symbols (e.g. constants, variable names, and keywords), $\mathcal{N}$ is a finite set of non-terminal symbols, $S \in \mathcal{N}$ is a start nonterminal, $\mathcal{R}$ is a set of production rules $A \rightarrow \alpha_1 \dots \alpha_n$ where $A \in \mathcal{N}$ and $\alpha_i \in \mathcal{N} \cup \Gamma$.

Formally, a grammar $\mathcal{G}$ defines a *single-step derivation* relation on sequences of symbols $\alpha, \beta, \gamma \in (\mathcal{N} \cup \Gamma)^*$: $\alpha A \gamma \Rightarrow \alpha \beta \gamma$ if $A \rightarrow \beta \in \mathcal{R}$. The reflexive transitive closure of this relation is called *derivation* and written $\Rightarrow^*$. A sequence of terminals w is a *sentence* if it is derivable from S ; the set of all sentences is called the *language* of the grammar $\mathcal{G}$, that is, $\mathcal{L}(\mathcal{G}) = \{w \in$

$\Gamma^* \mid S \Rightarrow^* w\}$.

In addition, we define the *prefix language* of $\mathcal{G}$ as the set of all prefixes of sentences in $\mathcal{L}(\mathcal{G})$, that is, $\mathcal{L}_{\text{prefix}}(\mathcal{G}) = \{w \in \Gamma^* \mid \exists v \in \Gamma^*. wv \in \mathcal{L}(\mathcal{G})\}$.

Pushdown Automata

Just as finite-state automata serve as the operational counterpart to regular expressions, pushdown automata act as the computational machine model for context-free grammars. A *pushdown automaton* is a tuple $\mathcal{P} = (\Sigma, \Pi, Q, q_0, Z_0, \delta, F)$ where Σ , Q , q_0 and F are as in their FSA definitions, Π is the stack alphabet, $Z_0 \in \Pi$ is the initial stack symbol, and $\delta \subseteq Q \times (\Sigma \cup \{\epsilon\}) \times \Pi^* \times Q \times \Pi^*$ is the set of transitions. Each transition $(q, c, \alpha, q', \beta)$ specifies that, in state q , upon reading the input symbol c and matching the top stack symbols to α , the PDA transitions to state q' and replaces α with the sequence β on the stack.

Finite State Transducer

A finite-state transducer (FST) is a tuple $\mathcal{T} = (\Sigma, \Gamma, Q, q_0, \delta, F)$ where all components are analogous to those of a FSA but each FST transition $q \xrightarrow{c:T_1 \dots T_k} q'$ in δ denotes that when reading a character c from state q , the FST moves to a new state q' and also outputs the sequence $T_1 \dots T_k$ of elements over the output alphabet Γ .

2.1.2 Grammar-Constrained Decoding

An (autoregressive) language model defines a probability distribution P on the set of all strings $w \in \Gamma^*$ over a vocabulary of tokens Γ via a product of left-to-right next-token conditional distributions $P(w_1 \dots w_n) = \prod_{i=1}^n P(w_i \mid w_{1:i-1})$.

Constrained decoding guides the output of Large Language Models (LLMs) by greedily enforcing user-given constraints in highly structured settings. *Grammar-constrained decoding* (GCD) [55] refers to the specific case where the constraint is given as a formal grammar that the LLM’s output must conform to. This is done by applying parsing algorithms to build an automaton that interfaces with the LLM’s decoding algorithm to mask away *all tokens that will*

provably lead to outputs not in the grammar. For example, GCD can be used to ensure that an LLM generates programs that only use a specific set of function names.

Algorithm 1: CONSTRAINEDDECODING

Input: Model M , Checker C , Tokenized prompt x

```

1  $\mathcal{V} := M.\text{vocabulary};$ 
2 repeat
3    $m := C(x; \mathcal{V});$ 
4    $\text{logits} := M(x);$ 
5    $t_{\text{next}} := \text{sample}(\text{apply\_mask}(m, \text{logits}));$ 
6    $x := x.\text{append}(t_{\text{next}});$ 
7 until  $t_{\text{next}} = \text{EOS};$ 
8 return  $x;$ 

```

Alg. 1 describes the abstract algorithm for general constrained decoding. When given a sequence of tokens $t_{1:k}$, CONSTRAINEDDECODING (Alg. 1) uses a checker C during the decoding process to mask out any next token t_{k+1} that would result in a prefix $t_{1:k+1}$ for which no possible completion satisfies the constraint. Specifically, given a prefix $t_{1:k}$ and vocabulary $\mathcal{V} \subseteq \Sigma^+$, a checker C computes a Boolean mask $C(t_{1:k}; \mathcal{V}) = m \in \{0, 1\}^{|\mathcal{V}|}$, where a value of 1 denotes a viable token (i.e., one for which there exists a sentence completion that can lead to constraint satisfaction), and 0 denotes an invalid one (i.e., one for which no sentence completion can lead to constraint satisfaction). The mask is then applied to the logits [36] produced by the language model to exclude invalid tokens from consideration.

2.2 Efficient and Flexible Grammar-Constrained Decoding

Parsing algorithms for Context-Free Grammars (CFG) achieve efficiency by processing input strings in two phases. Terminals—e.g., variable names or string literals—are recognized by a lexer in a preprocessing phase, while the grammatical structure of how terminals can be arranged—e.g., that the body of a loop should be a sequence of statements—is enforced by an automaton operating over lexed terminals. A key challenge with implementing GCD algorithms is that *the tokens used by subword tokenizers in LLMs do not align with the terminals used by*

parsers.

Because of this misalignment, GCD approaches either incur high *online token-masking overhead* (>1 second per token), or high *offline preprocessing costs* (>30 minutes) to precompute a lookup table that relates LLM tokens to terminals [16, 152]. Thus, existing GCD algorithms are impractical for domains where the constraining grammar frequently changes. Examples of such domains include *program synthesis* [4], where a grammar is provided for every program one may try to synthesize, and *grammar prompting* [154], where grammars are predicted from a given prompt to then guide the LLM towards a particular output structure.

In this chapter, we introduce a new approach for grammar-constrained decoding that is both *flexible*—i.e., handles diverse grammars without incurring prohibitive offline preprocessing costs—and *efficient*—i.e., maintains state-of-the-art efficiency in online token masking. The key innovation is a combined analysis of the LLM token vocabulary and set of CFG terminals. The analysis efficiently precomputes a lexer-state-dependent mapping between sequences of CFG tokens and individual LLM tokens. This precomputation allows the decoder to efficiently identify valid LLM tokens at decoding time while avoiding wasteful processing of token combinations that are not *realizable* within the given LLM vocabulary.

We implement our approach in a tool, GREATGRAMMA, which compares favorably to existing GCD approaches. GREATGRAMMA achieves in average 17.71x speedup in offline preprocessing compared to related approaches [152] while maintaining state-of-the-art online masking efficiency (5-32ms per token). Furthermore, our evaluation reveals that GCD implementations from existing published papers contain completeness bugs (e.g., they mask tokens that should not be masked or do not terminate when preprocessing simple grammars), whereas our approach lends itself to an elegant implementation consisting of simple modules.

This work makes two contributions:

- A flexible (low offline overhead) and efficient (low online token-masking overhead) algorithm for grammar-constrained decoding (Sec. 2.4-2.5).

- A tool implementing our algorithm, called GREATGRAMMA, that is up to 17.71x faster than other tools in offline preprocessing while retaining low online token-masking overhead (Sec. 2.6).

2.3 Approach Overview

In this section, we give some brief background about parsing (Sec. 2.3.1) and grammar-constrained decoding (Sec. 2.3.2), overview the high-level structure of our decoding approach (Sec. 2.3.3), and discuss our improvements over prior work.

2.3.1 Lexing and Parsing

While it is possible to build parsers that operate directly on input text, practical parsing tools first perform a preprocessing pass using a lexer for efficiency. A *lexer* is built from a set of regular expressions which define terminals (e.g. identifiers, keywords, or integer literals). Fig. 2.1a illustrates two regular expressions defining two terminals B (strings consisting of an a followed by a sequence of b’s) and C (strings consisting of an a followed by a sequence of c’s).

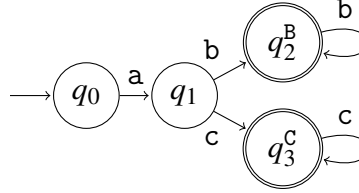
The lexer then takes as input a string and tokenizes it so that substrings matching regular expressions for terminals are grouped together. Here, our terminals consist of an a followed by either a sequence of b or c. For instance, the string “abaccab” lexes as `ab acc ab`.

The pass by the lexer ensures that the parser can be defined directly over terminals instead of individual characters. This separation of concerns avoids mixing character-level processing with higher-level grammar rules, making both components easier to implement and maintain. In particular, the grammar for the language can be defined at the *terminal* level. In our example, the grammar shown in Fig. 2.1e accepts sequences of terminals of the form `BCBCBC . . .` (where B and C can be any strings matching those terminals).

One way of defining a parser is as an automaton: it takes as input a sequence of terminals and either accepts or rejects the sequence. To handle all context-free grammars the automaton needs to be a pushdown automaton (PDA) that is equipped with a stack. In our example, the

B: ab+
C: ac+

(a) Terminals and regular expressions.



(b) Lexing automaton obtained from terminal definitions in Fig. 2.1a.

"a", "b", "c", "ab", "ac", "aba"

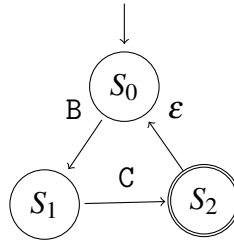
(c) Subword vocabulary for the LLM.

	a	b	c	ab	ac	aba
q_0	B, C	$\emptyset$	$\emptyset$	B	C	BB, BC
q_1	$\emptyset$	B	C	$\emptyset$	$\emptyset$	$\emptyset$
q_2	BB, BC	B	$\emptyset$	BB	BC	BBB, BBC
q_3	CB, CC	$\emptyset$	C	CB	CC	CBB, CBC

(d) Token spanner table computed from lexing automaton (Fig. 2.1b) and subword vocabulary (Fig. 2.1c).

$S := BC \mid BCS$

(e) Grammar $\mathcal{G}_{BC}$.



(f) Automaton for the grammar in Fig. 2.1e.

Figure 2.1. Illustrative example of the approach implemented in GREATGRAMMA.

simple stack-free automaton shown in Fig. 2.1f is enough to describe all valid sequences of terminals.

2.3.2 Using Parsing for Constrained Decoding

Going back to grammars, a parser either accepts or rejects a string. A simple extension is to make this process online—the parser rejects as soon as it consumes a token that will ensure the input string fails to parse. Intuitively, the key idea behind *grammar-constrained decoding* is that the parser primitive for checking if a token will be allowed or not by the parser can be used to build the checker C .

However, there is a key problem with using a parser as a checker for constrained decoding: a parser reads *language* level tokens (which this dissertation calls *terminals* to avoid confusion), while an LLM outputs tokens according to its subword vocabulary. The LLM tokens may span multiple or only fragments of language terminals, complicating the decoding process. This problem is known as *token misalignment* [122], and is the chief source of complexity for GCD implementations. For example some of the LLM tokens in Fig. 2.1c correspond to terminals whereas others can span multiple terminals.

2.3.3 Our Approach

Fig. 2.1 illustrates the overall structure of our approach, which is based on a data structure we introduce called a *token spanner table*. Intuitively, the token spanner table stores what sequences of terminals could be emitted by the lexer if it reads an LLM token from a given state.

Our GCD approach is split into two parts: an offline preprocessing phase in which the table is constructed and an online phase where the table is then used to generate token masks during decoding.

The offline portion of our algorithm takes two inputs: the LLM vocabulary $\mathcal{V}$ (Fig. 2.1c) and the definitions (as regexes) for terminals (Fig. 2.1a). It then constructs a finite state automaton (FSA) with a set of states Q representing the lexer (Fig. 2.1b). For example, state q_2 represents

a state where the lexer has already read the substring ab and can (optionally) read more b 's to emit a B terminal. We use this automaton together with the LLM's vocabulary to build the token spanner table (Fig. 2.1d). The keys of the token spanner table are pairs (q, v) where $q \in Q$ is a state of the lexer automaton and $v \in \mathcal{V}$ is an LLM token. The (q, v) entry of the token spanner table contains the set of sequences of terminals that can eventually be produced if the lexer is in state q and is fed in one additional LLM token v . For example, given state q_2 and LLM token aba we can either produce a sequence of terminals BBB or BBC —this is because we are in a lexer state corresponding to having started a B token, lex a complete B token from ab , and then start an arbitrary new token.

In the online portion, our algorithm uses the token spanner table together with the parser to construct masks stating which LLM tokens are valid at each step. During decoding, the algorithm tracks the state of the lexer (Fig. 2.1b) and the state of the parser PDA (Fig. 2.1f) on the prefix the LLM has generated so far. The algorithm then analyzes the state the parser is in to determine the possible sequences of terminal the parser could consume next. Then, the algorithm consults the token spanner table using these sequences and the current state of the lexer to determine what LLM tokens should be masked and kept.

2.4 Offline Token Preprocessing

Our approach starts by preprocessing the lexer to efficiently construct a lookup table that relates LLM tokens to terminal sequences (Sec. 2.4.1) and vice versa (Sec. 2.4.2). The preprocessed lexer is then used to analyze the parser to determine what terminal sequences are actually possible sequences in the grammar (Sec. 2.4.3).

2.4.1 Lexer Preprocessing

Let Σ denote the set of string characters, Σ^* denote the set of strings over this alphabet, and Γ denote the set of terminals (i.e., grammar-level tokens). A *lexer* is a function Lex that given an input string $w \in \Sigma^*$, returns a pair $(T_1 \dots T_k, w_r)$, where $T_1 \dots T_k \in \Gamma^*$ is a sequence

of terminals and $w_r \in \Sigma^*$ is the suffix of w that has not been lexed (i.e., mapped to language terminals) yet.

Typically lexers resolve ambiguity by making some simplifying assumptions that also help improve efficiency and avoid backtracking. We use the same assumptions and describe them next.

Maximal Munch Principle and Lookahead

Consider a language that contains two different tokens for the increment operator `++` and the addition operator `+`. Although the input string `++` could be tokenized as two separate `+` addition tokens, in practice lexers prioritize the longer valid token to resolve ambiguity (and which usually captures the intended syntax of the programming language). This behavior is called the *maximal munch principle*: the lexer matches the longest possible substring starting at the current position that aligns with a defined token pattern.

Under a strict interpretation of the maximal munch principle, if the lexer reaches the end of the input stream while processing a partial triple-quoted Python string `"""a"`, the lexer should tokenize the input as two strings `"""` and `"a"`. However, supporting such cases would require either waiting until the end of the input string to produce any tokens or allowing backtracking. As such, in practice many lexers (including Python's) will raise an error and stop lexing if the scanned prefix cannot be tokenized as a single terminal. This greedy behavior disallows some strings, but guarantees that the lexer can resolve all tokenization ambiguities by inspecting only the next character at each step.

Definition 2.4.1 (1-lookahead). *A lexer Lex is 1-lookahead if for every string $w \in \Sigma^*$ and valid continuation $c \in \Sigma$ of w , whenever $\text{Lex}(w) = (T_1 \dots T_k, r)$ then $\text{Lex}(wc)$ is either $(T_1 \dots T_k T_{k+1}, c)$ for some $T_{k+1} \in \Gamma$ or $(T_1 \dots T_k, rc)$.*

Terminals are specified as a set of regular expressions. It is oftentimes convenient to work with a *lexing automaton*, which is the finite state automaton (FSA) that accepts strings matching any terminal definition [102]. We refer the reader to paragraph 2.1.1 for formal definitions of the

semantics of FSA, but recall that an FSA is a tuple $\mathcal{A} = (\Sigma, Q, q_0, \delta, F)$ where Σ is the alphabet, Q is the set of states with initial state $q_0 \in Q$, δ contains transitions of the form $q \xrightarrow{c} q'$, and $F \subseteq Q$ is the set of final states.

Formally, a *lexer specification* is given as a finite set of pairs, each consisting of an automaton $\mathcal{A}^i = (Q^i, \Sigma^i, q_0^i, \delta^i, F^i)$ and a terminal symbol $T^i \in \Gamma$. Given a lexer specification $\{(\mathcal{A}^i, T^i)\}_i$, a corresponding (1-lookahead) *partial lexer* is a partial function $\text{Lex} : \Sigma^* \rightarrow \Gamma^* \times \Sigma^*$ which is defined as following:

- $\text{Lex}(\varepsilon) = (\varepsilon, \varepsilon)$
- Given $\text{Lex}(w) = (T_1 \dots T_k, w_r)$, the value of $\text{Lex}(wc)$ is
 - (1) $(T_1 \dots T_k T^j \$, \varepsilon)$ if $c = \text{EOS}$ and $w_r \in \mathcal{L}(\mathcal{A}^j)$ for some j
 - (2) $(T_1 \dots T_k, w_r c)$ if $c \neq \text{EOS}$ and $w_r c \in \mathcal{L}_{\text{prefix}}(\mathcal{A}^i)$ for some i
 - (3) $(T_1 \dots T_k T^j, c)$ if $c \neq \text{EOS}$ and $w_r \in \mathcal{L}(\mathcal{A}^j)$ but $w_r c \notin \mathcal{L}_{\text{prefix}}(\mathcal{A}^i)$ for all i
 - (4) $\perp$ otherwise

Given a context-free grammar $\mathcal{G} = (\Gamma, \mathcal{N}, S, \mathcal{R})$ with a separate lexer $\text{Lex} : \Sigma^* \rightarrow \Gamma^* \times \Sigma^*$, we say that a token sequence $w \in \mathcal{V}^*$ is a *sentence* if there exists a derivable sequence of terminals $T_1 \dots T_k$ such that $\text{Lex}(w) = (T_1 \dots T_k \$, \varepsilon)$. Formally,

$$\mathcal{L}^{\text{Lex}}(\mathcal{G}) = \{w \in \mathcal{V}^* \mid \exists T_1 \dots T_k. \text{Lex}(w) = (T_1 \dots T_k \$, \varepsilon) \wedge T_1 \dots T_k \$ \in \mathcal{L}(\mathcal{G})\}$$

We also define the *prefix language* of $\mathcal{G}$ with separate lexer Lex as the set of all prefixes of sentences in $\mathcal{L}^{\text{Lex}}(\mathcal{G})$, that is, $\mathcal{L}_{\text{prefix}}^{\text{Lex}}(\mathcal{G}) = \{w \in \Sigma^* \mid \exists v \in \Sigma^*. wv \in \mathcal{L}^{\text{Lex}}(\mathcal{G})\}$.

Lexing Transducer

A 1-lookahead maximal munch lexer can be defined from a lexing automaton as follows: The input is processed character-by-character by transitioning through the FSA's states. When

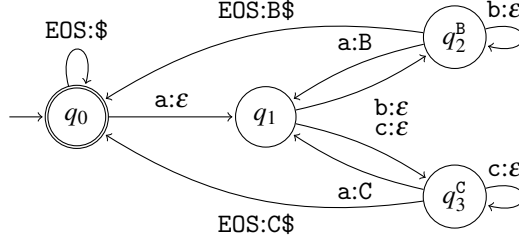


Figure 2.2. A lexing transducer $\mathcal{T}_{\mathcal{A}}$ derived from FSA $\mathcal{A}$ in Fig. 2.1.

no valid transition exists for the next character c , the lexer checks whether the current state corresponds to a valid language token. If it does and the tokenizer has at this point produced a pair $(T_1 \dots T_k, w_r)$, the not-yet tokenized string w_r is tokenized with token T_{k+1} corresponding to the reached state, the FSA is reset to its initial state q_0 (and the tokenizer state $(T_1 \dots T_k T_{k+1}, \epsilon)$ with the empty string ϵ), and the character c is consumed as the starting character of a new token q_0 . If the current state does not correspond to a valid token or if c cannot be consumed at q_0 , then c is invalid. Invalid characters inform what tokens should be masked during constrained decoding.

Algorithm 2: BUILDLEXINGFST

Input: FSA $\mathcal{A} = (\Sigma, Q, q_0, \delta, F)$, Output alphabet Γ ; // Terminal Map $T : F \rightarrow \Gamma$

Output: FST $\mathcal{T}_{\mathcal{A}} = (\Sigma, \Gamma, Q, q_0, \delta_{\text{FST}}, F_{\text{FST}})$

- 1 $\delta_{\text{FST}} := \{q \xrightarrow{c:\epsilon} q' \mid q \xrightarrow{c} q' \in \delta\}$, $F_{\text{FST}} := \{q_0\}$;
 - 2 **for** state q that recognizes language token $T \in \Gamma$ **do**
 - 3 **for** (c, q') s.t. $\exists q'' . q \xrightarrow{c} q'' \notin \delta$ and $q_0 \xrightarrow{c} q' \in \delta$ **do**
 - 4 $\delta_{\text{FST}} := \delta_{\text{FST}} \cup \{q \xrightarrow{c:T} q'\}$;
 - 5 $\delta_{\text{FST}} := \delta_{\text{FST}} \cup \{q \xrightarrow{\text{EOS}:T\$} q_0\} \cup \{q_0 \xrightarrow{\text{EOS}:\$} q_0\}$;
 - 6 **return** $\mathcal{T}_{\mathcal{A}} = (\Sigma, \Gamma, Q, q_0, \delta_{\text{FST}}, F_{\text{FST}})$;
-

This process can be formalized as a finite-state transducer (FST), an extension of a finite-state automaton that can produce output terminals when reading characters. Given the original lexing automaton $\mathcal{A}$ representing valid tokens, we write $\mathcal{T}_{\mathcal{A}}$ to denote the *lexing transducer*, the FST corresponding to $\mathcal{A}$. The construction of $\mathcal{T}_{\mathcal{A}}$ from $\mathcal{A}$ is formalized in Alg. 2, but at a high level the process simply adds transitions to handle cases where no valid transition exists for

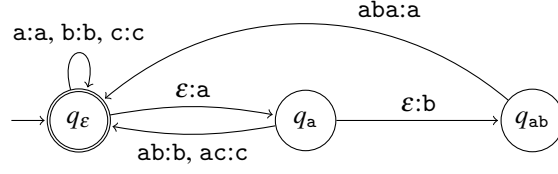


Figure 2.3. Detokenizing transducer for vocabulary $\mathcal{V} = \{a, b, c, ab, ac, aba\}$.

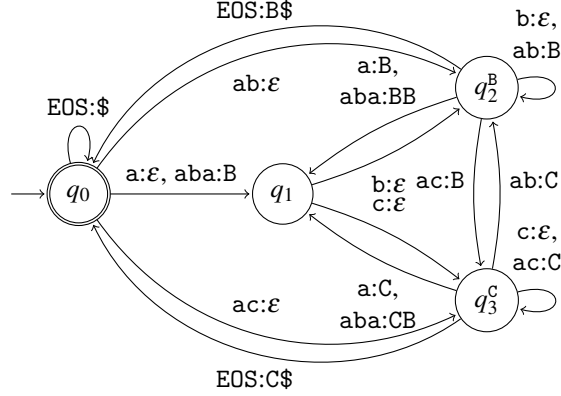


Figure 2.4. A determinized token-level lexing transducer $\mathcal{T}_{\mathcal{A} \circ \mathcal{V}}$, which is formed by composing $\mathcal{T}_{\mathcal{V}}$ from Fig. 2.3 and $\mathcal{T}_{\mathcal{A}}$ from Fig. 2.2.

the current character c , outputting terminals and exiting final states. Fig. 2.2 shows the lexing transducer derived from the FSA in Fig. 2.1.

LLM Token to Terminals

Processing LLM tokens character-by-character with the lexing transducer at runtime would incur significant overhead. To address this problem, we instead construct a token-to-terminal transducer by composing the character-to-terminal lexing transducer with the detokenizing transducer introduced by Koo et al. [87]. A detokenizing transducer simply maps LLM tokens to their corresponding sequence of text characters (i.e., the input alphabet is $\mathcal{V}$ and the output alphabet is Σ). A detokenizing transducer is nondeterministic and can contain ϵ -transitions that produce outputs without consuming inputs.

Fig. 2.3 illustrates the detokenizing transducer $\mathcal{T}_{\mathcal{V}}$ derived from the vocabulary $\mathcal{V} = \{a, b, c, ab, ac, aba\}$ in Fig. 2.1. Note that as an optimization common prefixes of tokens form a trie-like structure (e.g., state q_a denotes the state reached when producing the first a for all

Algorithm 3: BUILDDETOKENIZINGFST

Input: Vocabulary $\mathcal{V} \subseteq \Sigma^+$
Output: FST $\mathcal{T}_{\mathcal{V}} = (\mathcal{V}, \Sigma, Q, q_0, \delta, F)$
1 $Q := \{q_\varepsilon\}, F := \{q_\varepsilon\}, q_0 := q_\varepsilon, \delta := \emptyset;$
2 **for** $c_1 \dots c_k \in \mathcal{V}$ **do**
3 $q_{prev} := q_\varepsilon;$
4 **for** $i = 1$ **to** $k - 1$ **do**
5 $Q := Q \cup \{q_{c_1 \dots c_i}\};$
6 $\delta := \delta \cup \{q_{prev} \xrightarrow{\varepsilon:c_i} q_{c_1 \dots c_i}\};$
7 $q_{prev} := q_{c_1 \dots c_i};$
8 $\delta := \delta \cup \{q_{prev} \xrightarrow{c_1 \dots c_k:c_k} q_\varepsilon\};$
9 **return** $\mathcal{T}_{\mathcal{V}} = (\mathcal{V}, \Sigma, Q, q_0, \delta, F);$

tokens that start with that character), reducing computational overhead by reusing shared prefix computation. Fig. 2.4 depicts the combined token-level lexing transducer $\mathcal{T}_{\mathcal{A} \circ \mathcal{V}} = \mathcal{T}_{\mathcal{A}} \circ \mathcal{T}_{\mathcal{V}}$, which is the determinized FST capturing the sequential functional composition of the two transducers—i.e., the output of $\mathcal{T}_{\mathcal{V}}$ is fed as input to $\mathcal{T}_{\mathcal{A}}$. This token-to-terminal transducer enables efficient lookup of valid next tokens and produced terminal symbols in each state.

The lexing automaton $\mathcal{A}$ is defined as the product automaton of automata $\mathcal{A}^i$ specified in the lexer definition $\{(\mathcal{A}^i, T^i)\}$.

Lemma 2.4.2. *Let $\mathcal{T}_{\mathcal{A}} = (Q, \Sigma, \Gamma, q_0, \delta, F)$ be a lexing transducer for the lexer specification $\{(\mathcal{A}^i, T^i)\}_i$. Assume $\varepsilon \notin \mathcal{L}(\mathcal{A}^i)$ for all i and each $\mathcal{A}^i$ does not have any dead state. Then, for any string $w \in \Sigma^*$, $\text{Lex}(w) = (T_1 \dots T_k, w_r)$ if and only if $q_0 \xrightarrow{w:T_1 \dots T_k}^* q' \in \delta^*$ and $q_0 \xrightarrow{w_r:\varepsilon}^* q' \in \delta^*$ for some $q' \in Q$.*

Proof. The proof proceeds by induction. For the base case, let $w = \varepsilon$. Then $\text{Lex}(\varepsilon) = (\varepsilon, \varepsilon)$ and $q_0 \xrightarrow{\varepsilon:\varepsilon}^* q_0 \in \delta^*$. This is the only possible transition for empty input, as $\varepsilon \notin \mathcal{L}(\mathcal{A}^i)$ for each i .

Otherwise, we consider the remaining cases in the definition of Lex. Cases (1)-(3) are by construction. Assume $\text{Lex}(wc) = (T_1 \dots T_k, w_r)$, then by the induction hypothesis, $q_0 \xrightarrow{w:T_1 \dots T_k}^* q' \in \delta^*$ and $q_0 \xrightarrow{w_r:\varepsilon}^* q' \in \delta^*$ for some $q' \in Q$. The final case applies when $\text{Lex}(wc) = \perp$, indicating that no valid tokenization of wc can be produced. By assumption, our component

automata are constructed without dead states. This ensures that there doesn't exist $q' \in Q$ such that $q_0 \xrightarrow{w_r:T_1 \dots T_k}^* q' \in \delta^*$ when $w_r \notin \mathcal{L}_{\text{prefix}}(\mathcal{A}^i)$ for all i . Conversely, if there doesn't exist $q' \in Q$ such that $q_0 \xrightarrow{w:T_1 \dots T_k}^* q' \in \delta^*$, then by construction, cases (1)-(3) cannot hold so $\text{Lex}(wc)$ must be $\perp$. $\square$

The correctness of the detokenizing transducer is proven in the Appendix of [87]. The correctness of the token-level lexing transducer follows from the correctness of transducer composition.

Theorem 2.4.3. *Let $\mathcal{T}_{\mathcal{A} \circ \mathcal{V}} = (Q, \mathcal{V}, \Gamma, q_0, \delta, F)$ be a token-level lexing transducer for the lexer specification $\{(\mathcal{A}^i, T^i)\}_i$ and vocabulary $\mathcal{V} \subseteq \Sigma^+$. Then $q_0 \xrightarrow{w:T_1 \dots T_k}^* q' \in \delta^*$ if and only if $\text{Lex}(w) = (T_1 \dots T_k, w_r)$ for some $w_r \in \Sigma^*$ and $q' \in Q$ such that $q_0 \xrightarrow{w_r:\varepsilon}^* q'$.*

2.4.2 Realizable Terminal Sequences

Now that we have defined the formal machinery behind lexing, we are ready to explain how LLM tokens can be mapped to possible sequences of terminals.

When the transducer $\mathcal{T}_{\mathcal{A} \circ \mathcal{V}}$ in Fig. 2.4 consumes the LLM token `aba` from the initial state q_0 , it produces the terminal `B` and moves to state q_1 . If we inspect the grammar $\mathcal{G}_{\text{BC}}$ in Fig. 2.1e, we can deduce that the parser, which receives as input sequences of tokens, expects/requires the next language token to be `C`. Since $\mathcal{T}_{\mathcal{A} \circ \mathcal{V}}$ in state q_1 does not immediately produce any output when consuming `b`, the generated terminal sequence so far (i.e., `B`) is still a valid prefix according to the grammar $\mathcal{G}_{\text{BC}}$. However, after transitioning to q_2 , no possible path can generate `C` next.

As illustrated by the above example, for each transition $q \xrightarrow{t:T_1 \dots T_k} q'$ in the lexing transducer $\mathcal{T}_{\mathcal{A} \circ \mathcal{V}}$, we should check whether there is a terminal T such that (i) T can be generated along some path from q' , and (ii) $T_1 \dots T_k T$ is accepted by the grammar. This observation leads to the following definition, which describes which terminal sequences need to be considered by the parser.

Algorithm 4: BUILDINVERSETOKENSPANNERTABLE

Input: Combined FST $\mathcal{T}_{\mathcal{A} \circ \mathcal{V}} = (\mathcal{V}, \Gamma, Q, q_0, \delta, F)$

Output: Realizable token sequences $Re_{\mathcal{A} \circ \mathcal{V}}$,

Inverse lookup table $T_{\text{inv}} : Re_{\mathcal{A} \circ \mathcal{V}} \times Q \rightarrow 2^{\mathcal{V}}$

```
1  $Re_{\mathcal{A} \circ \mathcal{V}} := \emptyset, T_{\text{inv}}(\cdot, \cdot) := \emptyset;$ 
2 for  $q \xrightarrow{t:T_1 \dots T_k} q' \in \delta$  do
3   for  $T$  recognized at  $q'$ , that is reachable from  $q'$  do
4      $Re_{\mathcal{A} \circ \mathcal{V}} := Re_{\mathcal{A} \circ \mathcal{V}} \cup \{T_1 \dots T_k T\};$ 
5      $T_{\text{inv}}(q, T_1 \dots T_k T) := T_{\text{inv}}(q, T_1 \dots T_k T) \cup \{t\};$ 
6 return  $Re_{\mathcal{A} \circ \mathcal{V}}, T_{\text{inv}};$ 
```

Definition 2.4.4 (Producible Terminals). *Given a state q , the set of producible terminals $Prod(q)$ is defined as the set*

$$\{T_1 \in \Gamma \mid q \xrightarrow{w:T_1 \dots T_k}^* q' \in \delta^* \text{ for some } q' \in Q, w \in \Sigma^*\}$$

Definition 2.4.5 (Realizable Terminal Sequences). *Given a token vocabulary $\mathcal{V}$ and FSA $\mathcal{A}$, let δ be the set of transitions in the token-level lexing transducer $\mathcal{T}_{\mathcal{A} \circ \mathcal{V}}$. The set of realizable terminal sequences $Re_{\mathcal{A} \circ \mathcal{V}}$ is defined as*

$$Re_{\mathcal{A} \circ \mathcal{V}} = \{T_1 \dots T_k T \mid \exists q, q', t. q \xrightarrow{t:T_1 \dots T_k} q' \in \delta \text{ and } T \in Prod(q')\}.$$

Note that the LLM vocabulary $\mathcal{V}$ contains all characters in Σ , ensuring that $\Sigma^* = \mathcal{V}^*$. Therefore, any next terminal producible in $\mathcal{T}_{\mathcal{A}}$ is also producible in the combined transducer $\mathcal{T}_{\mathcal{A} \circ \mathcal{V}}$ and vice versa. This equivalence allows us to simplify producibility checks: instead of analyzing the large combined transducer $\mathcal{T}_{\mathcal{A} \circ \mathcal{V}}$, we need only compute reachability to accepting states within $\mathcal{T}_{\mathcal{A}}$ to determine producible next terminals.

Inverse Token Spanner Table

Alg. 4 computes the set of realizable terminal sequences and constructs the key data structure we use to perform constrained decoding.

Definition 2.4.6 (Inverse Token Spanner Table). *Given a lexer state $q \in Q_{\mathcal{A} \circ \mathcal{V}}$ and terminal sequence $T_1 \dots T_k T \in Re_{\mathcal{A} \circ \mathcal{V}}$, the entry $T_{inv}(q, T_1 \dots T_k T)$ in the inverse token spanner table T_{inv} is the set of tokens that generates $T_1 \dots T_k T$ from state q . Formally,*

$$T_{inv}(q, T_1 \dots T_k T) = \{t \mid q \xrightarrow{t:T_1 \dots T_k} q' \in \delta \text{ and } T \in Prod(q')\}$$

Note that, for a given state q' , the set of producible terminals T in the lexing transducer can be precomputed using reachability algorithm [51]. This table allows a decoder to determine which LLM tokens can result in a given realizable terminal sequence $T_1 \dots T_k T$ being produced from a specific lexer state.

Going back to the example in Fig. 2.1, the table in Fig. 2.1d illustrates the terminal sequences produced by the lexing transducer for each pair of state and token. The set of all realizable terminal sequences, $Re_{\mathcal{A} \circ \mathcal{V}}$, is obtained by taking the union of all entries in this table.

Building on Fig. 2.1d, we can also derive an inverse token spanner table, T_{inv} . For a given state q and terminal sequences α , the table T_{inv} provides the set of tokens that can generate the sequence α from the state q . For example, the terminal sequence BC is generated by the token aba in state q_0 (i.e., $aba \in T_{inv}(q_0, BC)$). The same sequence is generated by the token ac in state q_2 (i.e., $ac \in T_{inv}(q_2, BC)$).

By construction of inverse token spanner table, the following proposition holds.

Proposition 2.4.7. *Given a token sequence $w \in \mathcal{V}^*$ such that $q_0 \xrightarrow{w:T_1 \dots T_k}^* q'$, any $v \in \mathcal{V}$ and $T \in \Gamma$, the token v is in the inverse token spanner table entry $T_{inv}(q', T_{k+1} \dots T_m T)$ if and only if $q_0 \xrightarrow{wv:T_1 \dots T_k T_{k+1} \dots T_m}^* q''$ and $T \in Prod(q'')$.*

2.4.3 Parser Preprocessing

As with lexer preprocessing, our goal is to also preprocess the parser to avoid iterating over every terminal sequence generated by the lexer at runtime. The challenge with pushdown

automata (PDAs), compared to lexers (NFAs), is their requirement for a stack to parse context-free grammars. This reliance on the stack prevents us from fully determining at preprocessing time whether a given terminal sequence is acceptable from a particular state, as the stack's contents must be examined during execution. However, we can still identify many terminal sequences that will always be accepted or always rejected, independent of the current stack.

To compute which terminal sequences will always be accepted or always rejected, we directly compose the detokenizing transducer $Re_{\mathcal{A} \circ \mathcal{V}}$ (Definition 2.4.5), with the PDA $\mathcal{P}$ produced by a parser generator (and where transitions operate over single terminals) [2] to obtain a new pushdown automaton $\mathcal{P} \circ \mathcal{I}_{Re_{\mathcal{A} \circ \mathcal{V}}}$ where transitions operate over terminal sequences.

This last pushdown automaton can efficiently determine valid sequences of terminal symbols from each parser state. We note that preprocessing both the lexer and parser in tandem is a key feature that distinguishes our work from prior work [16, 152].

Stack invariance provides a sufficient condition for knowing when a sequence of terminals is accepted.

Proposition 2.4.8 (Stack Invariance). *If a PDA $\mathcal{P}$ accepts an input sequence w in state q with stack configuration γ , then w is also accepted in the same state q when the stack configuration is $\gamma' \cdot \gamma$ for some γ' (i.e., when γ appears at the top of the stack with additional symbols beneath it).*

It follows that a terminal sequence accepted by the PDA starting with an empty stack configuration is accepted under any stack configuration.

The following proposition shows how to construct a stack-free finite-state automaton that overapproximates the set of sequences accepted by a PDA.

Proposition 2.4.9 (Overapproximation via FSA). *Consider an FSA $\mathcal{A}_{\mathcal{P}}$ obtained by removing all stack operations from a PDA $\mathcal{P}$. If an input sequence w is not accepted by $\mathcal{A}_{\mathcal{P}}$ in state q , then w cannot be accepted by $\mathcal{P}$ in state q with any stack configuration.*

Following Proposition 2.4.9, a terminal sequence is always rejected if it is rejected by the FSA obtained by removing all stack operations.

The above reasoning can be formalized by computing the set of *always-accepted tokens* A and of *context-dependent terminal sequences* D . Given a lexer state $q^{\mathcal{A}}$ and a parser state $q^{\mathcal{P}}$, we denote by $A(q^{\mathcal{A}}, q^{\mathcal{P}})$ the set of tokens that are accepted regardless of the stack configuration γ , and by $D(q^{\mathcal{A}}, q^{\mathcal{P}})$ the set of terminal sequences that may be accepted by some configuration γ .

Algorithm 5: PREPROCESSPARSER

Input: PDA $\mathcal{P}$, Realizable sequences $Re_{\mathcal{A} \circ \mathcal{V}}$,
FSA $\mathcal{A}$, Inverse token spanner table T_{inv}
Output: Always-accepted token table A
Context-dependent sequence table D

```

1  $\mathcal{A}_{\mathcal{P}} := \text{REMOVESTACKOPERATIONS}(\mathcal{P});$ 
2 for  $q^{\mathcal{P}} \in Q_{\mathcal{P}}$  do
3    $\bar{A}(q^{\mathcal{P}}) := \{\alpha \in Re_{\mathcal{A} \circ \mathcal{V}} \mid q \text{ with stack } \varepsilon \text{ accepts } \alpha\};$ 
4    $\bar{R}(q^{\mathcal{P}}) := \{\alpha \in Re_{\mathcal{A} \circ \mathcal{V}} \setminus \bar{A}(q) \mid \mathcal{A}_{\mathcal{P}} \text{ rejects } \alpha \text{ in } q\};$ 
5    $\bar{D}(q^{\mathcal{P}}) := Re_{\mathcal{A} \circ \mathcal{V}} \setminus \bar{A}(q) \setminus \bar{R}(q);$ 
6   for  $q^{\mathcal{A}} \in Q_{\mathcal{A}}$  do
7      $A(q^{\mathcal{A}}, q^{\mathcal{P}}) := \bigcup_{\alpha \in \bar{A}(q)} T_{\text{inv}}(q^{\mathcal{A}}, \alpha);$ 
8      $D(q^{\mathcal{A}}, q^{\mathcal{P}}) := \{\alpha \in \bar{D}(q) \mid T_{\text{inv}}(q^{\mathcal{A}}, \alpha) \neq \emptyset\};$ 
9 return  $A, D;$ 

```

Alg. 5 describes how to preprocess a parser to build a table of always-accepted tokens A and context-dependent sequences D .

Note that one key source of efficiency resulting from our approach is that many transitions in the combined transducer $\mathcal{T}_{\mathcal{A} \circ \mathcal{V}}$ produce the same terminal sequence $T_1 \dots T_k T$, making $|Re_{\mathcal{A} \circ \mathcal{V}}|$ smaller than $|\mathcal{V}|$ or $|\delta|$. Thus, the set of realizable terminal sequences $Re_{\mathcal{A} \circ \mathcal{V}}$ enables efficient precomputation of what sequences of terminals the parser should consider.

2.5 Online Token Masking

With a preprocessed inverse token spanner table T_{inv} , along with the table of always-accepted tokens A and context-dependent sequences D , we are now ready to describe the online part of our grammar-constrained decoder (Alg. 6).

Algorithm 6: COMPUTEVALIDTOKENMASK

Input: PDA $\mathcal{P}$, FSA $\mathcal{A}$, Inverse token spanner table T_{inv}

Always-accepted token table A ,

Context-dependent sequence table D ,

Lexer state $q^{\mathcal{A}}$, Parser state $q^{\mathcal{P}}$, Stack configuration γ

Output: Set of allowed tokens $\mathcal{V}_{\text{allowed}}$

```

1  $\mathcal{V}_{\text{allowed}} := A(q^{\mathcal{A}}, q^{\mathcal{P}});$ 
2 for  $\alpha \in D(q^{\mathcal{A}}, q^{\mathcal{P}})$  do
3   if  $\alpha$  is accepted by  $\mathcal{P}$  in state  $q^{\mathcal{P}}$  with stack  $\gamma$  then
4      $\mathcal{V}_{\text{allowed}} := \mathcal{V}_{\text{allowed}} \cup T_{\text{inv}}(q^{\mathcal{A}}, \alpha);$ 
5 return  $\mathcal{V}_{\text{allowed}};$ 

```

At each decoding step, Alg. 6 analyzes the lexer state $q^{\mathcal{A}}$, parser state $q^{\mathcal{P}}$, and the current stack configuration γ to produce the set of exactly all LLM tokens $\mathcal{V}_{\text{allowed}}$ that can lead to a sequence accepted by the grammar.

Although we construct the combined PDA $\mathcal{P} \circ \mathcal{T}_{Re_{\mathcal{A} \circ \mathcal{V}}}$ for preprocessing what terminal sequences a token can result in, using the same automaton for checking at decoding time whether LLM tokens can lead to valid parsing sequences would be inefficient. PDAs cannot always be determinized and one would have to compute reachability for nodes in $\mathcal{P} \circ \mathcal{T}_{Re_{\mathcal{A} \circ \mathcal{V}}}$ under different stack configurations to determine valid language token sequences. This computation would effectively involve testing all sequences in $Re_{\mathcal{A} \circ \mathcal{V}}$ at decoding time, making the preprocessing of the parser meaningless. Therefore, Alg. 6 uses the set of always-accepted tokens A whenever possible to avoid traversing the PDA and only analyzes what terminal the PDA can accept at decoding time for context-dependent sequences described by D . The set of next legal tokens is then obtained by looking up, for each sequence of terminals accepted by the grammar, the set of tokens that can lead to that sequence. This is done by consulting the inverse token spanner table T_{inv} .

Lemma 2.5.1. *Consider a string $w \in \mathcal{V}^*$ with an extended transition $q \xrightarrow{w:T_1 \dots T_k}^* q'$ through the lexing transducer, where $T_1 \dots T_k$ is incomplete prefix of $\mathcal{L}(\mathcal{G})$ (i.e., $T_1 \dots T_k \in \mathcal{L}_{\text{prefix}}(\mathcal{G}) \setminus \mathcal{L}(\mathcal{G})$). If $w \in \mathcal{L}_{\text{prefix}}^{\text{Lex}}(\mathcal{G})$ then there must exist some terminal $T \in \text{Prod}(q')$ such that $T_1 \dots T_k T \in \mathcal{L}(\mathcal{G})$.*

$\mathcal{L}_{\text{prefix}}(\mathcal{G})$.

Proof. Suppose $w \in \mathcal{L}_{\text{prefix}}^{\text{Lex}}(\mathcal{G})$. By definition of the prefix language, there exists a non-empty suffix $v \in \mathcal{V}^+$ such that $wv \in \mathcal{L}^{\text{Lex}}(\mathcal{G})$. Consequently, $\text{Lex}(wv)$ must yield a complete token sequence ending in an EOS token, say $(T_1 \dots T_k T_{k+1} \dots T_m \$, \epsilon)$, where $T_1 \dots T_k T_{k+1} \dots T_m \$ \in \mathcal{L}(\mathcal{G})$.

Given our assumption that $T_1 \dots T_k \notin \mathcal{L}(\mathcal{G})$, it follows that $T_{k+1} \dots T_m$ cannot be empty. Specifically, this means T_{k+1} must exist and be producible from the state q' , i.e., $T_{k+1} \in \text{Prod}(q')$.

Therefore, by combining $T_1 \dots T_k$ with the next token T_{k+1} , we form a valid prefix of a complete token sequence $T_1 \dots T_k T_{k+1} \in \mathcal{L}_{\text{prefix}}(\mathcal{G})$. $\square$

Theorem 2.5.2 (Completeness). *Let $\mathcal{P} = (\Gamma, \Pi, Q^{\mathcal{P}}, q_0^{\mathcal{P}}, Z_0, \delta^{\mathcal{P}}, F^{\mathcal{P}})$ be a pushdown automaton such that $\mathcal{L}(\mathcal{P}) = \mathcal{L}(\mathcal{G})$. Given a string $w \in \mathcal{L}_{\text{prefix}}^{\text{Lex}}(\mathcal{G})$ and a token $v \in \mathcal{V}$, the token v is not masked by Alg. 6 (i.e., $v \in \mathcal{V}_{\text{allowed}}$) if the concatenation wv belongs to $\mathcal{L}_{\text{prefix}}^{\text{Lex}}(\mathcal{G})$.*

Proof. By Thm. 2.4.3, the lexing transducer emitted $T_1 \dots T_k$ and reached a state q' such that $q_0 \xrightarrow{w_r:\epsilon}^* q'$. If $wv \in \mathcal{L}_{\text{prefix}}^{\text{Lex}}(\mathcal{G})$, then $q_0 \xrightarrow{wv:T_1 \dots T_k T_{k+1} \dots T_m} q''$ for some state q'' , which implies $q' \xrightarrow{T_{k+1} \dots T_m} q''$. By Lemma 2.5.1, there exists $T \in \text{Prod}(q'')$ such that $T_1 \dots T_k T_{k+1} \dots T_m T \in \mathcal{L}_{\text{prefix}}(\mathcal{G})$. Therefore, $v \in T_{\text{inv}}(q', T_{k+1} \dots T_m T)$, and hence $v \in \mathcal{V}_{\text{allowed}}$. $\square$

Lemma 2.5.3. *If $T \in \text{Prod}(q)$, then there exist $q' \in Q$ and $v \in \mathcal{V}^*$ such that $q \xrightarrow{v:T}^* q' \in \delta^*$.*

Proof. Since $T \in \text{Prod}(q)$, there exist $q'' \in Q$ and $v \in \mathcal{V}^*$ such that $q \xrightarrow{v:T\alpha}^* q'' \in \delta^*$. Note that $\Sigma \subseteq \mathcal{V}$. A key property of the lexing transducer is that it produces at most one terminal for each input character. Consequently, there must exist a prefix v_p of v and a state $q' \in Q$ such that $q \xrightarrow{v_p:T}^* q' \in \delta^*$. $\square$

Lemma 2.5.4. *Assume that if there exists some $v \in \mathcal{V}^*$ such that $q_0 \xrightarrow{w_r v:T} q$, then for any terminal sequence $\alpha \in \Gamma^*$, there also exists some $v' \in \mathcal{V}^*$ such that $q_0 \xrightarrow{w_r v':T\alpha} q'$. Consider a string $w \in \mathcal{V}^*$ with an extended transition $q \xrightarrow{w:T_1 \dots T_k}^* q'$ through the lexing transducer, where $T_1 \dots T_k$*

is incomplete prefix of $\mathcal{L}(\mathcal{G})$ (i.e., $T_1 \dots T_k \in \mathcal{L}_{\text{prefix}}(\mathcal{G}) \setminus \mathcal{L}(\mathcal{G})$). If there exists $T \in \text{Prod}(q')$ such that $T_1 \dots T_k T \in \mathcal{L}_{\text{prefix}}(\mathcal{G})$ then $w \in \mathcal{L}_{\text{prefix}}^{\text{Lex}}(\mathcal{G})$.

Proof. Suppose there exists $T \in \text{Prod}(q')$ such that $T_1 \dots T_k T \in \mathcal{L}_{\text{prefix}}(\mathcal{G})$. By the definition of prefix language, there exists $\alpha \in \Gamma^*$ such that $T_1 \dots T_k T \alpha \in \mathcal{L}(\mathcal{G})$. Furthermore, by Lemma 2.5.3, for such a T , there exists a state $q'' \in Q$ and a string $v \in \mathcal{V}^*$ such that $q' \xrightarrow{v:T}^* q'' \in \delta^*$. Now, by the assumption, there also exists a string $v' \in \mathcal{V}^*$ such that for some state $q''' \in Q$, $q_0 \xrightarrow{wv':T_1 \dots T_k T \alpha}^* q''' \in \delta^*$. Here, since α must end with $\$,$ it implies that q''' must be the initial state q_0 by construction. Thus, we have $\text{Lex}(wv') = (T_1 \dots T_k T \alpha, \varepsilon)$. Since $T_1 \dots T_k T \alpha \in \mathcal{L}(\mathcal{G})$, this final result implies that $w \in \mathcal{L}_{\text{prefix}}^{\text{Lex}}(\mathcal{G})$. $\square$

Theorem 2.5.5 (Soundness). *Let $\mathcal{P} = (\Gamma, \Pi, Q^{\mathcal{P}}, q_0^{\mathcal{P}}, Z_0, \delta^{\mathcal{P}}, F^{\mathcal{P}})$ be a pushdown automaton such that $\mathcal{L}(\mathcal{P}) = \mathcal{L}(\mathcal{G})$. Given a string $w \in \mathcal{L}_{\text{prefix}}^{\text{Lex}}(\mathcal{G})$ and a token $v \in \mathcal{V}$, the token v is not masked by Alg. 6 (i.e., $v \in \mathcal{V}_{\text{allowed}}$) only if the concatenation wv belongs to $\mathcal{L}_{\text{prefix}}^{\text{Lex}}(\mathcal{G})$.*

Proof. By Thm. 2.4.3, the lexing transducer emitted $T_1 \dots T_k$ and reached a state q' such that $q_0 \xrightarrow{w_r:\varepsilon}^* q'$. If $v \in \mathcal{V}_{\text{allowed}}$, then there must exist some $T \in \Gamma$ such that $v \in T_{\text{inv}}(q', T_{k+1} \dots T_m T)$. By the construction of inverse token spanner table, this implies that the concatenation $T_1 \dots T_k T_{k+1} \dots T_m T$ is in the prefix language $\mathcal{L}_{\text{prefix}}(\mathcal{G})$. Furthermore, this means that there exist some $q'' \in Q$ such that $q' \xrightarrow{T_{k+1} \dots T_m T} q''$ and $T \in \text{Prod}(q'')$. Therefore, by Lemma 2.5.4, the concatenation wv belongs to $\mathcal{L}_{\text{prefix}}^{\text{Lex}}(\mathcal{G})$. $\square$

2.6 Experiments

We implemented our algorithms for computing token masks in a tool called GREATGRAMMA. A testament to the simplicity of our approach is that GREATGRAMMA is implemented in just 900 lines of Python code built on top of the LALR parser provided by the LARK library [139]. Specifically, we used LARK to parse regular expressions and context-free grammars, and used LARK's LALR parser to generate a parse table representing a deterministic PDA for parsing sequences of tokens in a given context-free grammar.

In this section, we evaluate GREATGRAMMA by answering the following two questions:

- **RQ1:** How does the *offline preprocessing overhead* of GREATGRAMMA compare to existing GCD approaches?
- **RQ2:** How does the *online per-token decoding overhead* of GREATGRAMMA compare to existing GCD approaches?

Because all GCD approaches in theory produce the same token masks and only differ in execution time, we do not need to evaluate GREATGRAMMA’s performance on specific downstream tasks; the effectiveness of GCD has already been evaluated in prior work [55, 16, 117].

Models and Grammars.

We conduct our experiments using three different tokenizers: Llama (32K tokens), Llama-3 (128K), and Qwen-2 (151K). We consider simplified Go, Java, and Python grammars from prior work on GCD [152]. We choose these grammars for three reasons: they are large grammars (87-99 terminals, 109-145 nonterminals, 353-520 production rules, 7,441-9,319 bytes), they capture real programming languages that are used in existing applications of constrained decoding, and they illustrate well the trade-off between offline preprocessing and online masking times. While other smaller grammars appear in existing work on GCD, the tools we compare against all take slightly different grammar formats with various restrictions (e.g., no left recursion was allowed in XGRAMMAR), and we had to manually translate all grammars we considered between these formats to perform our evaluation (a very time-consuming task).

Baselines.

We compare GREATGRAMMA against OUTLINES [157], SYNCODE [152], and XGRAMMAR [44], the three GCD tools that can nominally handle grammars of practical sizes.

Measures.

For each combination of tokenizer and grammar, we measured the time taken to preprocess the grammar for that tokenizer and the average time taken by each GCD implementation to

produce a token at inference time. To fairly evaluate per-token computation time, we wanted to ensure that all three tools followed the same token selection process and recorded the time required to compute the mask at each step—i.e., the online overhead. We manually built 5 programs in each grammar and used them to decide what sequence of tokens the decoder was going to produce—i.e., for each example program, the decoder produced each token in the program in order and computed the token masks at each step. Following this setup, we could exactly measure the same online average per-token overhead across all GCD approaches.

Results.

Tab. 2.1 reports the results. We do not include XGRAMMAR in Tab. 2.1 because it encountered errors during either preprocessing or decoding for all the grammars we considered—it failed to compile the Java grammar, and entered an infinite loop or encountered a segmentation fault during decoding for Python and Go grammars.

On average (geomean), GREATGRAMMA’s *offline preprocessing* is 17.71x faster than SYNCODE, but 30.01x slower than OUTLINES.

For certain terminal sequences, SYNCODE incorrectly masked a valid token (we suspect the source of this imprecision is that SYNCODE only unrolls future terminal sequences up to a fixed bound k). When considering the remaining data points, GREATGRAMMA’s *online masking* is on average (geomean) 2.73x faster than SYNCODE and 550.57x faster than OUTLINES.

Discussion.

In summary, GREATGRAMMA emerges as the new state-of-the-art GCD approach. GREATGRAMMA is 17.71x faster at offline preprocessing than SYNCODE, the only other GCD tool with acceptable online overhead (**RQ1**), and exhibits the lowest online masking overhead than any other GCD tool (**RQ2**).

While OUTLINES has the lowest preprocessing offline overhead, its seconds-per-token online overhead does not meet the needs of most practical applications of LLMs. OUTLINES’ online overhead is due to the fact that its CFG module verifies the acceptability of each token at

runtime.

The improvement in offline pre-processing over SYNCODE is expected as our new approach targets the inefficiency of SYNCODE’s algorithm for token alignment. The slight improvement in online token-mask computation over SYNCODE is likely due to how GREATGRAMMA only consults the PDA for context-dependent sequences, while SYNCODE does so for all terminal sequences up to a bound.

While of course we cannot guarantee that GREATGRAMMA is free of bugs, the simplicity of our approach makes its implementation easier (just 900 lines of code).

Furthermore, GREATGRAMMA’s offline preprocessing times have allowed us to heavily test our implementation without incurring multi-hour testing times.

Limitations.

The current implementation of GREATGRAMMA works under the lexing assumptions described in Sec. 2.4.1: maximal munch principle and 1-lookahead lexing. It therefore does not support languages for which lexing requires more than 1-lookahead or instances where the same input sequence must be lexed differently depending on the parsing context.

For example, in Java, the end of the nested generic `List<List<T>>` is erroneously tokenized by a lexer operating under maximal munch as `>>`—i.e., the bitwise right-shift operator. However, in this context the parser instead expects two consecutive `>` terminals denoting the closure of a generic type. (The Java grammar we used in our evaluation, sourced from the SYNCODE benchmark, does not support generic types.)

To resolve this type of lexing ambiguity, the lexer must consider the current state of the parser, a task that for arbitrary regular expressions requires the lexer to perform unbounded lookahead. It is possible to modify our approach to handle practically occurring ambiguities by allowing the lexing transducer to nondeterministically generate both possible tokenizations (e.g., a single `>>` terminal or two separate `>` terminals), specifically for such cases that do not require arbitrary lookahead, and enabling the parser to select the valid interpretation based on

grammatical constraints.

For a lexer requires $k > 1$ lookahead, a k -lookahead lexing transducer can be implemented by encoding the state and its k -lookahead as a single combined state, similar to an $LR(k)$ parser [85]. However, this approach can significantly increase the size of the resulting transducer.

While our implementation uses a deterministic PDA as its basis, our algorithm can be modified to handle nondeterminism by tracking multiple active states concurrently. This extension is applicable to nondeterministic PDAs without ε -transition loops that push stack symbols. Handling multiple states involves calculating the possible next tokens for each state individually and then taking the union of these allowed tokens.

2.7 Related Work

There are many different implementations of grammar constrained decoding [55, 157, 100, 16, 152, 44, 57], but we focus our comparison on the ones that can handle realistic grammars (e.g., those of modern programming languages).

SYNCODE [152] is the only that can provide low online overhead for the large grammars in our evaluation. SYNCODE speculatively unrolls future terminal sequences up to a fixed bound (i.e., every terminal sequence in Γ^k) and precomputes a table similar to our inverse token spanner table but *for all* sequences. Our evaluation shows that our table can be computed significantly faster since it only contains the set of realizable terminal sequences.

DOMINO [16] precomputes a tree of terminal sequences similar to our lexing transducer. However, at decoding time, DOMINO must traverse the entire tree because the set of realizable terminal sequences varies *for each* lexer state. In contrast, our approach computes a single *global* set of realizable terminal sequences $Re_{\mathcal{A} \circ \mathcal{V}}$, which we then use to preprocess the parser. DOMINO’s implementation is not publicly available, but their paper reports 20s for offline pre-processing and 22% online overhead for an extremely simplified C grammar with approximately 70 rules when using the smaller Llama tokenizer. For the same tokenizer, GREATGRAMMA can

preprocess *much larger grammars* (353-520 rules) with similar times (25-35s).

XGRAMMAR [44] reduces runtime checks by preprocessing context-independent tokens for character-based nondeterministic PDAs, but mentions the misalignment problem in their related work. By preprocessing the set of realizable terminal sequences and the inverse token spanner table, we efficiently integrate their context-independent token caching approach with a parser that uses a separate lexer. Furthermore, our overapproximation via FSA can identify more always-rejected sequences compared to their expanded suffix analysis, as our method can also follow rule-reduction edges by treating them as ε -transitions.

Existing research on the effectiveness of constrained decoding in structured generation shows mixed results. Some studies [55, 16] demonstrate that GCD can improve downstream task performance. However, other recent work [148, 117] highlights that GCD can also negatively impact the quality of generated outputs. Our approach is orthogonal to these methods and can be integrated with techniques proposed by Park et al. [117] or Banerjee et al. [11] to mitigate such negative effects.

Comparison with LLGUIDANCE [108]

After we completed this work, Moskal et al. [108] released LLGUIDANCE, a fast GCD tool written in Rust that incorporates a memory-optimized tokenizer trie, a derivative-based lazy lexer, and a highly optimized Earley parser. LLGUIDANCE shares some similarities with our compositional approach, but it relies on derivative-based parsing instead of automaton- and transducer-based constructions.

The evaluation provided on LLGUIDANCE’s website reports impressive performance (in the order of micro to milliseconds for both offline pre-processing and online per-token overhead) on benchmarks like JSONSchemaBench [56]. However, we could not compare this tool to our approach because LLGUIDANCE (v.0.7.27) reports the “ParserTooComplex” error when applied to the grammars used in our experiments. However, we believe that some of the optimizations employed by LLGUIDANCE can also be beneficial for GREATGRAMMA.

2.8 Conclusion

We present GREATGRAMMA, a tool for grammar-constrained decoding that is both flexible (low offline overhead) and efficient (low online overhead). GREATGRAMMA precomputes a succinct and easy-to-generate data structure that only captures terminal sequences that are *realizable* by LLM tokens in a given lexer state. This data structure also speeds up online masking compared to similar approaches by further reducing the number of inputs the parser has to check during decoding.

GREATGRAMMA is built with simple primitives that are already supported by existing parsing libraries and lends itself to an easy implementation consisting of just 900 lines of Python. This simple implementation is easier to test (we have identified that other tools often crash or produce incorrect outputs) and opens the door to many future possible research directions—e.g., finding ways to leverage the PDA stack to perform more aggressive precomputation or implementing decoding directly on GPUs.

Acknowledgments

Chapter 2, in full, is based on the paper from Kanghee Park, Timothy Zhou, and Loris D’Antoni, “Flexible and Efficient Grammar-Constrained Decoding” published in International Conference on Machine Learning, pp. 48262-48275. PMLR, 2025. The dissertation author is the primary investigator and author of this paper.

Table 2.1. Offline and online processing times. X denotes when SYNCODE masked incorrect tokens.

Model ($ V $)	Grammar	Offline preprocessing (s)			Online per-token overhead (ms)		
		OUTLINES	SYNCODE	GREAT GRAMMA	OUTLINES	SYNCODE	GREAT GRAMMA
Llama (32,000)	Go	2.51	556.10	24.76	3,213.05	X	5.08
	Java	2.49	579.48	33.79	3,450.67	46.71	6.29
	Python	4.89	768.33	35.40	3,245.10	62.40	7.11
Llama-3 (128,256)	Go	2.43	2,212.95	106.07	12,974.43	X	21.11
	Java	2.46	2,294.26	166.87	13,725.58	49.03	24.69
	Python	4.86	2,789.31	170.71	13,167.83	71.49	30.17
Qwen-2 (151,665)	Go	2.38	2,635.00	132.27	14,756.47	X	21.32
	Java	2.39	2,734.27	260.80	15,235.25	50.32	25.12
	Python	4.81	3,268.09	155.32	15,146.38	70.87	32.30

Chapter 3

Grammar-Aligned Decoding

Constrained decoding addresses the inefficiency of rejection sampling by directly “forcing” the LLM output to satisfy a given constraint. Specifically, as explored in the previous chapter, grammar-constrained decoding (GCD) utilizes automata to perform on-the-fly token masking. This mechanism preemptively filters out any tokens that would provably result in grammatically invalid output during generation.

While GCD does not incur the overhead of rejection sampling—i.e., the generated output is always in the language of the grammar—we show that GCD and in general all forms of structured decoding introduce a new problem: **structured decoding distorts the LLM’s learned language distribution**, effectively hindering the LLM’s capabilities.

This chapter introduces and formalizes *grammar-aligned decoding* (GAD), the problem of sampling from an LLM so that the outputs (1) are guaranteed to adhere to a given grammar, and (2) are unbiased *wrt.* the LLM’s distribution. Although exact GAD is intractable in general (similar to rejection sampling), we propose a new adaptive decoding algorithm for *approximate* GAD, which starts off as GCD and gradually converges to the LLM’s distribution, and thus allows trading off between efficiency and accuracy. The algorithm, which we dub *Adaptive Sampling with Approximate Expected Futures* (ASAp), is built “on top” of existing constrained decoding algorithms. Whereas GCD approaches simply mask out tokens that lead to non-grammatical sequences for a given prefix, ASAp remembers for all sampled prefixes the probability associated

$$\begin{aligned}
S &::= 00000 \mid 1A_2 \\
A_i &::= 0A_{i+1} \mid 1A_{i+1}, \text{ for } i = 2, 3, 4 \\
A_5 &::= 0 \mid 1
\end{aligned}$$

Figure 3.1. CFG $\mathcal{G}_{sk}$ over tokens $\Gamma = \{0, 1\}$, written in Backus-Naur form (BNF) notation. This grammar accepts the string 00000 and all length-5 strings that start with a 1.

with masked-out tokens and uses it to upper bound the probability of grammaticality. By updating this bound when more samples are observed, the decoding algorithm converges to the desired probability distribution—i.e., it samples outputs from the LLM-induced probability conditioned on the outputs being accepted by the grammar. The idea works for any structured decoding approach and not just for GCD, but in this chapter we focus our evaluation on constraints expressed via grammars.

We evaluate ASAp on two structured prediction tasks: formal program synthesis and constituency parsing. Our experiments on program synthesis and NLP tasks show that GCD techniques generate outputs that are grammatical but unlikely according to the LLM, while with ASAp, the likelihood of the generated outputs improves over time, converging to the target constrained LLM—i.e., GAD better respects the LLM while still enforcing the constraints.

3.1 Grammar-Aligned Decoding

In this section, we formalize the problem of *grammar-aligned decoding* (GAD) as decoding from an autoregressive language model while enforcing the output sequence to be accepted by a given context-free grammar. We also demonstrate the limitations of existing approaches to this problem.

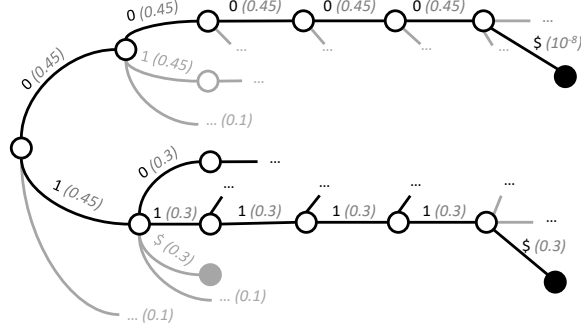


Figure 3.2. Fragment of the conditional model distribution P for Ex. 1 depicted as a trie. Each node corresponds to a prefix $w_{1:i-1}$, and each edge is annotated with the next token w_i and its conditional probability $P(w_i | w_{1:i-1})$. Filled nodes are complete strings. Grayed out parts of the trie are outside of the grammar $\mathcal{G}_{sk}$.

Grammar-Aligned Decoding

Given a model distribution P and a CFG $\mathcal{G}$, *grammar-aligned decoding (GAD)* is the task of sampling from the distribution $Q^{P,\mathcal{G}}$ that is *proportional* to P but *restricted* to sentences in $\mathcal{G}$:

$$Q^{P,\mathcal{G}}(w) = \frac{\mathbb{1}[w \in \mathcal{L}(\mathcal{G})] \cdot P(w)}{\sum_{w'} \mathbb{1}[w' \in \mathcal{L}(\mathcal{G})] \cdot P(w')}$$

When P and $\mathcal{G}$ are clear from context, we will write $Q(w)$ instead of $Q^{P,\mathcal{G}}(w)$.

Example 1 (GAD). Consider the distribution P that arises from prompting an LLM to “generate a binary string that ends with a 1”. We expect P to assign high probability to strings of the form $(0 | 1)^* 1$ —i.e. those that satisfy the prompt (*Mixtral-8x7B-Instruct-v0.1* (temperature=1) generates binary strings that end with a 1 around 90% of the time.) A snippet of a possible distribution P is depicted in Fig. 3.2.

Suppose we constrain the model’s output to the language of the grammar $\mathcal{G}_{sk}$ in Fig. 3.1, which only accepts strings of length 5. Moreover, $\mathcal{G}_{sk}$ only accepts one string that starts with 0, i.e., 00000, which does not end with 1. In Fig. 3.2, the grayed out parts of the trie are tokens that lead to sequences outside of the grammar $\mathcal{G}_{sk}$. According to the definition of GAD, the target sampling distribution $Q^{P,\mathcal{G}_{sk}}$ should assign: (i) high probability to all eight strings of the form

$1w_2w_3w_41$ —which conform both to the grammar and the prompt; (ii) low probability to the string 00000 —which conforms to the grammar but not the prompt; and (iii) zero probability to all other strings.

Exact GAD

Can one exactly sample from $Q^{P,\mathcal{G}}$? Rejection sampling, which repeatedly draws from P until a sample lands in $\mathcal{L}(\mathcal{G})$, provably yields exact samples according to $Q^{P,\mathcal{G}}$, but if P assigns most of its mass outside of $\mathcal{L}(\mathcal{G})$, it is intractably slow, especially if the prompt is not including information about the grammar (see [154]). For Ex. 1, rejection sampling would be highly inefficient because the model would generate many strings that are not of length five.

In contrast, exact sampling from P is efficient because its joint distribution is represented by a product of easily computed left-to-right conditionals, enabling ancestral sampling (i.e., generating tokens left to right, conditioned on already generated tokens). Can we similarly factor Q into a product of left-to-right conditionals $Q^{P,\mathcal{G}}(w_i | w_{1:i-1})$, to enable ancestral sampling?

For simplicity, let us assume that P is a distribution over sequences of exactly length n (although, in practice, language models can produce ‘stop’ tokens which allow for a valid distribution on sequences of all lengths). The exact conditionals of $Q^{P,\mathcal{G}}$ are given by:

$$\begin{aligned} Q^{P,\mathcal{G}}(w_i | w_{1:i-1}) &\propto \sum_{w_{i+1:n}} \left[\mathbb{1}[w \in \mathcal{L}(\mathcal{G})] \cdot \prod_{j=i}^n P(w_j | w_{1:j-1}) \right] \\ &\propto P(w_i | w_{1:i-1}) \cdot \mathbb{E}_{P(w_{i+1:n} | w_{1:i})} [\mathbb{1}[w \in \mathcal{L}(\mathcal{G})]] \end{aligned} \quad (3.1)$$

Thus, exact left-to-right sampling from $Q^{P,\mathcal{G}}$ consists of sampling from model conditionals $P(w_i | w_{1:i-1})$, with an additional weighting term $c(w_{1:i}) = \mathbb{E}_{P(w_{i+1:n} | w_{1:i})} [\mathbb{1}[w \in \mathcal{L}(\mathcal{G})]]$ that considers the grammar.

We refer to $c(w_{1:i})$ as *expected future grammaticality* (EFG), i.e. the probability that a continuation of $w_{1:i}$ sampled from P lands in $\mathcal{L}(\mathcal{G})$. Using this notation, we can write the exact

left-to-right sampling conditional explicitly as:

$$Q^{P,\mathcal{G}}(w_i \mid w_{1:i-1}) = \frac{P(w_i \mid w_{1:i-1}) \cdot c(w_{1:i})}{\sum_{w'_i} P(w'_i \mid w_{1:i-1}) \cdot c(w_{1:i-1}, w'_i)} \quad (3.2)$$

To see why computing this conditional is intractable, consider using dynamic programming to compute $c(w_{1:i})$ by marginalizing over a product of potential functions: the set of model conditionals and an indicator potential for the grammar. While the indicator potential can be factorized across rules in the grammar, the model's contribution generally does not factorize: in practice, the final conditional probability $P(w_n \mid w_{1:n-1})$ is a global potential function, defined by a non-linear neural network touching every variable. Thus, the main goal of this chapter is to develop effective approximations to the EFG $c(w_{1:i})$, which would enable us to compute the left-to-right conditionals of Q .

Limitations of Grammar-Constrained Decoding

Existing work [154, 157] has proposed *grammar-constrained decoding* (GCD) as a way to efficiently sample from an autoregressive language model subject to grammar constraints. Although the exact details of these techniques vary depending on class of grammars they support, the common thread is that they rely on an *incremental parser*, which can efficiently check whether a given string w is a prefix of a sentence in the grammar, i.e., $w \in \mathcal{L}_{\text{prefix}}(\mathcal{G})$. When given a sentence $w_{1:i-1}$, GCD techniques use this parser during decoding to mask out any next token w_i that results in a prefix $w_{1:i}$ for which no completion will produce a sequence in the grammar. Using the trie in Fig. 3.2 as an example, one can think of GCD as sampling a path through the trie by selecting only among the black outgoing edges from every node, proportional to their conditional probabilities in the diagram (e.g. the first token is 0 or 1 with equal probability).

In terms of the GAD problem, we can view GCD as approximating the exact left-to-right conditionals $Q^{P,\mathcal{G}}(w_i \mid w_{1:i-1})$ by the conditional distribution $\tilde{Q}_{\text{GCD}}(w_i \mid w_{1:i-1})$, defined as

follows:

$$\tilde{Q}_{\text{GCD}}(w_i \mid w_{1:i-1}) = \frac{P(w_i \mid w_{1:i-1}) \cdot \mathbb{I}[w_{1:i} \in \mathcal{L}_{\text{prefix}}(\mathcal{G})]}{\sum_{w'_i} P(w'_i \mid w_{1:i-1}) \cdot \mathbb{I}[w_{1:i-1}, w'_i \in \mathcal{L}_{\text{prefix}}(\mathcal{G})]}$$

Though not originally formulated in this way, we can view recent work on GCD [154, 157] as forming a binary approximation $\mathbb{I}[w_{1:i} \in \mathcal{L}_{\text{prefix}}(\mathcal{G})]$ to the EFG $c(w_{1:i})$. In other words, while GCD considers the *possibility* of future grammaticality, it makes no attempt to integrate the model’s likelihood to estimate *expected* future grammaticality, which can lead to substantial bias in the sampling distribution—i.e., every EFG such that $c(w_{1:i}) > 0$ will simply be approximated via the value 1.

Example 2 (GCD). *Consider again the GAD problem from Ex. 1, where our target sampling distribution $Q^{P, \mathcal{G}_{sk}}$ assigns high probability to strings that both start and end with a 1 and a low probability to the string 00000. However, we observe that GCD [54] generates strings ending with a 1 only 30% of the time—i.e., GCD has effectively ruined the LLM’s ability to follow the prompt by biasing sampling towards 00000, an incorrect output.*

When generating the first token (0 or 1), the GCD algorithm does not know how many grammatical strings can start with each character and, more importantly, how likely these strings are under P . Since both tokens 0 and 1 have the possibility of leading to a grammatical string, GCD will estimate their expected future grammaticality as 1, and choose each of them roughly half of the time (since $P(0) \approx P(1)$). Once GCD has chosen 0, however, it becomes “trapped” in the part of the search space where the only grammatical string is the low-probability sequence 00000.

Ex. 2 illustrates how existing GCD approaches can hinder the language model’s abilities to explore the space of possible outputs according to the learned distribution, thus highlighting the importance of designing a better approximation to the EFG $c(w_{1:i})$; this is addressed in the next section.

3.2 Adaptive Sampling with Approximate Expected Futures

In this section, we propose an adaptive sampling algorithm that iteratively builds better approximations of the future grammaticality of a sequence. Our procedure operates by sampling *repeatedly*, each time bounding lost probability mass to provably ungrammatical areas of the search space in order to better guide the next sampling iteration. As a result, our algorithm converges over many iterations to exact samples from the constrained LLM distribution, allowing for a flexible trade-off between efficiency and accuracy.

Overview of the Algorithm

GCD approaches poorly approximate the desired distribution because they greedily sample prefixes without worrying about the EFG. When sampling the first token in Ex. 2, GCD simply uses the likelihood for tokens 0 and 1 assigned by the LLM without considering the probability that these next tokens would result in grammatical completions if sampling were unconstrained—i.e. without incorporating the critical EFG re-weighting terms that are necessary for unbiased sampling from the constrained LLM distribution. However, if GCD ends up sampling 0 as the first token for Ex. 2, it will necessarily sample the string 00000 since no other sequences starting with 0 are allowed by the grammar. We can “learn” from this result: the true probability mass assigned to all grammatical sequences starting with a 0 is not 0.45 as the LLM’s next token probability would have us believe; instead, the total grammatical mass in this section of the search space is the joint probability of the single string 00000, which is the much lower value of $0.45^5 * 10^{-8}$ as depicted in Fig. 3.3. In other words, simply by sampling 00000, we can better approximate (in this case, exactly) the EFG of tokens along this path.

The key insight behind our algorithm, which we call ASAp, is that we can iterate this process of discovering lost grammatical probability mass by repeatedly sampling and revising transition weights after each sample is produced. More formally, we can think of this procedure as starting with GCD’s over-approximation to each EFG $c(w_{1:i})$ term, and then, through repeated sampling and discovery of mass assigned to non-grammatical completions, reducing each

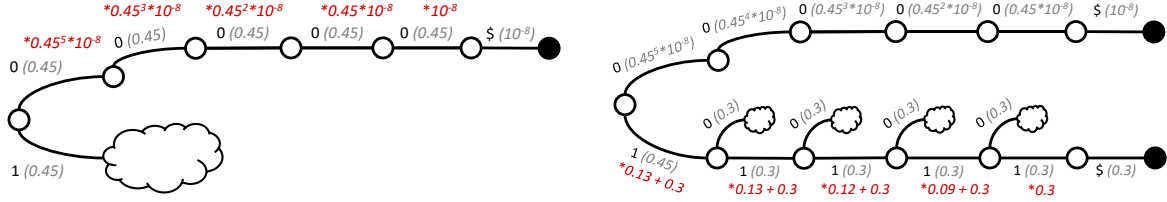


Figure 3.3. Illustration of the trie built by ASAP after sampling 00000 as the first string (left) and after sampling 11111 as the second string (right). EFG updates after each iteration are shown in red.

overapproximation to make it more accurate. In the limit, the approximations converge to exact EFG estimates and unbiased sampling.

Two possible first iterations of the ASAP algorithm are depicted in Fig. 3.3. In the first iteration (left of Fig. 3.3), after sampling the sequence 00000, the algorithm directly addresses the issue that arose in Ex. 2 by attempting to better approximate the probability mass of potential grammatical completions of each prefix of 00000 (red quantities). For example, the expected future grammaticality of the prefix 0000 is now $0.45 * 10^{-8}$ —i.e., the algorithm effectively “looks ahead” to determine that only one valid (but low probability) string 0\$ that can follow 0000. The ideas developed in GCD allow us to efficiently compute, for a given string, the likelihood of the next tokens that will immediately result in non-grammaticality.

If we only sample one string from the LLM, we cannot hope to do better than GCD in terms of sampling faithfully in a grammar-aligned way. However, if we were to now sample once more, we could now better direct our sampling strategy. In the second iteration (right of Fig. 3.3), the string 11111 is sampled and the expected future grammaticality is updated (red quantities). Note that at this point the probabilities assigned to the string 00000 from the earlier iteration have already been updated.

By repeating the above approach multiple times (i.e., by producing more samples), the ASAP algorithm produces precise approximations of the expected future grammaticalities and thus better samples from the constrained LLM.

Algorithm Formalization

The key quantity that the algorithm approximates based on past samples is the *expected future grammaticality* (EFG) $c(w_{1:i}) = \mathbb{E}_{p(w_{i+1:n}|w_{1:i})}[\mathbb{1}[w \in \mathcal{L}(\mathcal{G})]]$. At iteration $m + 1$, our algorithm uses the set of samples $S = \{s_1, \dots, s_m\}$ observed so far to compute an overapproximation $\tilde{c}_S(w_{1:i})$ of $c_S(w_{1:i})$ for every possible string $w_{1:i}$. The overapproximation is inductively defined:

$$\begin{aligned} \tilde{c}_S(w_{1:i}) &= \mathbb{1}[w_{1:i} \in \mathcal{L}_{\text{prefix}}(\mathcal{G})] && \text{no string in } S \text{ starts with } w_{1:i} \\ \tilde{c}_S(w_{1:i}) &= \sum_{w_{i+1}} P(w_{i+1} \mid w_{1:i}) \cdot \tilde{c}_S(w_{1:i+1}) && \text{otherwise} \end{aligned} \quad (3.3)$$

Intuitively, if no samples in S start with the prefix $w_{1:i}$, then $\tilde{c}_S(w_{1:i})$, the overapproximation of EFG is simply whether the string is or is not a valid prefix in the grammar—i.e. the same overapproximation used by GCD. If, on the other hand, we *have* encountered the prefix $w_{1:i}$ before in previous samples in S , the overapproximation uses the next token likelihoods that were computed during the previous sampling runs of the algorithm to compute a better estimate of EFG.

For example, in Fig. 3.3, once we have sampled the sequences 00000 and 11111, we have that $\tilde{c}_S(0000) = 0.45 * 10^{-8}$ and $\tilde{c}_S(110) = 1$ (i.e., we have not seen a sample with the prefix 110 yet).

Theorem 3.2.1. $\forall w_{1:i} \in \Sigma^*, \tilde{c}_S(w_{1:i}) \geq c(w_{1:i})$.

Proof. To see that $\tilde{c}_S(w_{1:i})$ is indeed an upper bound on $c(w_{1:i})$, consider two cases: First, suppose $w_{1:i}$ is not a prefix of any string in S . In this case, $c(w_{1:i}) \leq 1 = \tilde{c}_S(w_{1:i})$. Second, we need to prove that $\forall w_{1:i} \in \text{prefix}(S), c(w_{1:i}) \leq \tilde{c}_S(w_{1:i})$, where $\text{prefix}(S)$ is the set of (finitely many) prefixes of string in S . We proceed by induction, starting from the longest prefixes. No matter whether we are in the base case or in the inductive step, for any w_{i+1} , the string $w_{1:i+1}$ either is not a prefix of S , where by the first case we obtain $c(w_{1:i+1}) \leq \tilde{c}_S(w_{1:i+1})$, or it is a longer prefix of S , for which the same inequality holds by the induction hypothesis. Observing

that $c(w_{1:i})$ factorizes similarly to $\tilde{c}_S(w_{1:i})$, this leads us to the following inequality:

$$\begin{aligned} c(w_{1:i}) &= \sum_{w_{i+1}} P(w_{i+1} \mid w_{1:i}) \cdot c(w_{1:i+1}) \\ &\leq \sum_{w_{i+1}} P(w_{i+1} \mid w_{1:i}) \cdot \tilde{c}_S(w_{1:i+1}) = \tilde{c}_S(w_{1:i}) \end{aligned} \quad \square$$

The sampling procedure itself proceeds autoregressively like GCD, but using the iteratively updated EFG estimates we have just defined, $\tilde{c}_S$. Specifically, the left-to-right sampling conditional for our procedure, $\tilde{Q}_S(w_i \mid w_{1:i-1})$, after having previously sampled the strings in S , is defined as follows:

$$\tilde{Q}_S(w_i \mid w_{1:i-1}) = \frac{P(w_i \mid w_{1:i-1}) \cdot \tilde{c}_S(w_{1:i})}{\sum_{w'_i} P(w'_i \mid w_{1:i-1}) \cdot \tilde{c}_S(w_{1:i-1}, w'_i)} \quad (3.4)$$

Algorithm 7: ASAp algorithm

```

1 Initialize  $S := \{\}$ ,  $\tilde{c}_S(\cdot) := 1$ ;
2 for  $m \leq M$  do
3   Draw  $w_{1:n} \sim \tilde{Q}_S$  via ancestral sampling;
4    $S := S \cup \{w_{1:n}\}$ ;
5   for  $i$  in  $(n-1) \dots 1$  do
6     for  $w'$  in  $\{w' \mid w_{1:i} \cdot w' \notin \mathcal{L}_{\text{prefix}}(\mathcal{G})\}$  do
7        $\tilde{c}_S(w_{1:i} \cdot w') := 0$ ;
8      $\tilde{c}_S(w_{1:i}) := \sum_{w'} P(w' \mid w_{1:i}) \cdot \tilde{c}_S(w_{1:i} \cdot w')$ ;
9 return Final sample  $w_{1:n}$ ;
```

Our overall algorithm, which is presented in Algorithm 7, then proceeds iteratively, using past samples to improve subsequent samples. Whenever the sample set S is updated with a new sample $w_{1:n}$, the overapproximation $\tilde{c}$ is updated for the prefixes of $w_{1:n}$. The update begins at the end of the sequence and proceeds backward toward the start, by the recursive definition

in Eq. 3.3. In the listing, we assume that we are only interested in the final sample, but in our evaluation we will analyze whether the algorithm induces the desired distribution.

Next we provide a proof that this algorithm converges to exact estimates of EFG in the limit of infinite iterations, and therefore to exact samples from the constrained LLM distribution. The theorem assumes almost sure termination of ancestral sampling in the unconstrained LLM distribution P —i.e., the LLM eventually terminates.

Theorem 3.2.2. *Let $S_m = \{s_1, \dots, s_m\}$ be the set of recorded samples up to the m -th iteration of ASAp. Then, $\forall w_{1:i} \in \mathcal{L}_{\text{prefix}}, \tilde{c}_{S_m}(w_{1:i}) \rightarrow c(w_{1:i})$ as $m \rightarrow \infty$ almost surely.*

Proof. Fix an arbitrary sequence $w_{1:i} \in \mathcal{L}_{\text{prefix}}$. Let $\mathcal{L}(w_{1:i})$ be the set of all strings w in $\mathcal{L}$ extending the fixed prefix $w_{1:i}$. For every m , let $X_m \subseteq \mathcal{L}_{\text{prefix}}$ be the set of shortest extensions $w_{1:j} \in \mathcal{L}_{\text{prefix}}$ of $w_{1:i}$ that *have not yet been encountered* in the first m samples, that is, are not prefixes of any string in S_m . It can be easily seen that

$$\begin{aligned} \tilde{c}_{S_m}(w_{1:i}) - c(w_{1:i}) &= \sum_{w_{1:j} \in X_m} P(w_{i+1:j} \mid w_{1:i}) \cdot (1 - c(w_{1:j})) \\ &\leq \sum_{w_{1:j} \in X_m} P(w_{i+1:j} \mid w_{1:i}), \end{aligned} \tag{3.5}$$

Now consider an arbitrarily small $\varepsilon > 0$. Because the infinite sum $\sum_{w \in \mathcal{L}(w_{1:i})} P(w_{i+1:\infty} \mid w_{1:i})$ equals 1, there exists a *finite* subset $V_\varepsilon \subseteq \mathcal{L}(w_{1:i})$ such that $\sum_{w \in V_\varepsilon} P(w_{i+1:\infty} \mid w_{1:i}) > 1 - \varepsilon$ (we just take appropriately long initial prefix of the infinite sum); in other words, $\sum_{w \in \mathcal{L}(w_{1:i}) \setminus V_\varepsilon} P(w_{i+1:\infty} \mid w_{1:i}) < \varepsilon$. Suppose that for some m we have the property that for every $w \in V_\varepsilon$,

1. if $w \in \mathcal{L}$, then $w \in S_m$, and
2. if $w \notin \mathcal{L}$, then its longest prefix $w_{1:i}$ that belongs to $\mathcal{L}_{\text{prefix}}$ is a prefix of some $w' \in S_m$.

Coming back to Eq. 3.5, we observe that the sum $\sum_{w_{1:j} \in X_m} P(w_{i+1:j} \mid w_{1:i})$ can be rewritten as the sum of $P(w_{i+1:\infty} \mid w_{1:i})$ over all strings $w \in \mathcal{L}(w_{1:i})$ having a prefix in X_m . But strings $w \in V_\varepsilon \cap \mathcal{L}$ do not have a prefix in X_m , because by (1) they were already sampled. Strings $w \in V_\varepsilon \setminus \mathcal{L}$ also

do not have a prefix in X_m , because by (2) all their prefixes were either encountered or are not in $\mathcal{L}_{\text{prefix}}$ (while $X_m \subseteq \mathcal{L}_{\text{prefix}}$). Thus, by the definition of V_ε , the sum $\sum_{w_{1:j} \in X_m} P(w_{i+1:j} \mid w_{1:i})$ is bounded by ε . Having this for every $\varepsilon > 0$, we obtain convergence.

It remains to prove that indeed almost surely there exists a value of m such that (1) and (2) hold for every word in V_ε . Because V_ε contains only finitely many strings, it is enough to prove this for every string $w \in V_\varepsilon$ (and then take a maximum of the finitely many obtained values of m). Thus, fix such a string $w \in V_\varepsilon$; also fix a step number m , together with the current set of previous samples S_m . In case (1) (i.e., when $w \in \mathcal{L}$), let $n = \text{len}(w)$, and in case (2) (i.e., when $w \notin \mathcal{L}$), let n be the length of the longest prefix $w_{1:n}$ of w that belongs to $\mathcal{L}_{\text{prefix}}$. In both cases, we need to prove that the probability that the string sampled in the next step will start with $w_{1:n}$ is not smaller than some positive constant, independent of m and S_m (for case (1) notice that after sampling the first $\text{len}(w)$ letters of w we have no choice, only w can be sampled). Take thus some $j \in \{1, \dots, n\}$, suppose that we have already sampled $w_{1:j-1}$, and we are now going to sample the j -th letter of the word. Once again, because there are finitely many positions j in consideration, it is enough to prove that for each such j the probability of sampling w_j as the next letter is not smaller than some positive constant. But notice that $\tilde{c}_{S_m}(w_{1:j}) \geq c(w_{1:j}) > 0$ (the second inequality because $w_{1:n} \in \mathcal{L}_{\text{prefix}}$) and $\tilde{c}_{S_m}(w_{1:j-1} \cdot a) \leq 1$ for every letter a . It follows that the probability of sampling w_j as the j -th letter, that is $\tilde{Q}_S(w_j \mid w_{1:j-1})$, is at least

$$\frac{P(w_i \mid w_{1:i-1}) \cdot c(w_{1:i})}{\sum_{w'_i} P(w'_i \mid w_{1:i-1})},$$

which is a positive constant independent from m and S_m , as needed. $\square$

3.3 Experiments

We implemented the ASAp algorithm as an extension of the Transformers-CFG implementation of GCD [54]. When the LLM generates a sequence $w_{1:n}$, the ASAp algorithm keeps track of the original LLM's probability $P(w_i \mid w_{1:i-1})$ for $1 \leq i \leq n$ and the set of allowed next

tokens $\{w_i \mid w_{1:i-1}, w'_i \in \mathcal{L}(\mathcal{G})\}$ determined by the incremental parser in the Transformers-CFG library. After the LLM finishes generating a sequence, our implementation of ASAp updates the overapproximation $\tilde{c}_S$ from the end of sequence by back-propagating the quantity 1 minus probability of the tokens that will for sure lead to non-grammatical sequences. The implementation of ASAp updates $\tilde{c}_S(w_{1:n-1}, w'_n)$ for all possible tokens w'_n , and then moves on to update $\tilde{c}_S(w_{1:n-2}, w'_{n-1}) \dots, \tilde{c}_S(w_1, w'_2), \tilde{c}_S(w'_1)$ using Equation (3.3).

Datasets and Models.

We consider the benchmark from Example 2 and three structured-decoding tasks. Two of our tasks involve solving Syntax-Guided Synthesis Problems (SyGuS) [4]. SyGuS is a standardized format where one provides a logical specification and a context free grammar of first-order terms and the goal is to synthesize a term in the grammar that satisfies the specification. SyGuS is a natural fit for GAD and we consider two tasks from the standard SyGuS benchmarks where grammars vary from benchmark to benchmark: strings with linear integer arithmetic (SLIA) and conditional inverse generation with bit-vector arithmetic (INV-BV). In the former, the grammar is used to restrict what constant strings one can use when building string-manipulating programs and in the latter the grammar is used to restrict constant bit-vectors and operations used to build invariants. Fig. 3.4 and Fig. 3.5 provide examples of SLIA and INV-BV problems. For both families of benchmarks, our prompts consist of 3 in-context examples of the form (specification, solution) and the grammar is then provided as a constraint for GAD. Our third task is the constituency parsing (CP) task already used in prior GCD work [55] where the grammar is used to help the model produce well-parenthesized parse trees for English sentences.

Due to constrained resources and needing to run inference multiple times to measure whether the distribution $\tilde{Q}$ is faithful to Q , we randomly select 15 SLIA problems, 15 INV-BV problems, and 6 CP problems. We select the open-source Mistral-7B [80] for evaluation due to its superior reasoning and code generation capabilities.

```

; Determines what terms can appear
(set-logic SLIA)
; The function to synthesize
(synth-fun f ((name String)) String
  ; The grammar for f to be synthesized in
  ((Start String (S))
    (S String
      (name " " ".")
      (str.++ S S)
      (str.at S I)
      (str.replace S S S)
      (str.substr S I I)))
    (I Int
      (0 1 2 (+ I I) (- I I)
        (str.len S)
        (str.indexof S S I))))))

; Specifications to satisfy
(constraint (= (f "Nancy_FreeHafer") "N.F.))
(constraint (= (f "Andrew_Cencici") "A.C.))
(constraint (= (f "Jan_Kotas") "J.K.))
(constraint (= (f "Mariya_Sergienko") "M.S.))

(a) SLIA/initials-small

```

```

Start ::= S
S ::= name | " " | "."
      | str.++ S S | str.at S I
      | str.replace S S S
      | str.substr S I I
I ::= 0 | 1 | 2 | + I I | - I I
      | str.len S | str.indexof S S I

(b) Grammar for f

```

Figure 3.4. (a) A SLIA problem in which the grammar for the target function is explicitly defined. (b) The explicitly defined grammar for `f` written in BNF notation. The goal of problem is to find an implementation for `synth-fun` functions that satisfies all the `constraints` within a specified grammar—i.e., to find implementation of `f` in the grammar (b)

```

; Determines what terms can appear
(set-logic BV)
; The function to synthesize
(synth-fun inv
  ((s (BitVec 4)) (t (BitVec 4)))
  (BitVec 4))
; Helper functions
(define-fun min () (BitVec 4)
  (bvnot (bvlshr (bvnot #x0) #x1)))
(define-fun max () (BitVec 4) (bvnot min))
(define-fun l
  ((s (BitVec 4)) (t (BitVec 4))) Bool
  (bvsgt (bvnot (inv s t)) t))
(define-fun SC
  ((s (BitVec 4)) (t (BitVec 4))) Bool
  (distinct t max))

; Specifications to satisfy
; with universally quantified variables
(declare-var s (BitVec 4))
(declare-var t (BitVec 4))
(constraint (=> (SC s t) (l s t)))

(a) INV-BV/find_inv_bvsle_bvlshr1_4bit

```

```

Start ::= BV
BV ::= s | t
      | #x0 | #x7 | #x8
      | bvneg BV | bvnot BV
      | bvadd BV BV | bvsub BV BV
      | bvand BV BV | bvlor BV BV
      | bvlshl BV BV | bvlshr BV BV

```

(b) Grammar for inv

Figure 3.5. (a) An INV-BV problem in which the grammar for the target function `inv` is implicitly defined. (b) The implicitly defined grammar for `inv` written in BNF notation. The grammar is implicitly defined by primitives of BV logic and parameters of `inv`. The goal of problem is to find an implementation for `synth-fun` functions that satisfies all the constraints within a specified grammar—i.e., to find implementation of `inv` in the grammar (b).

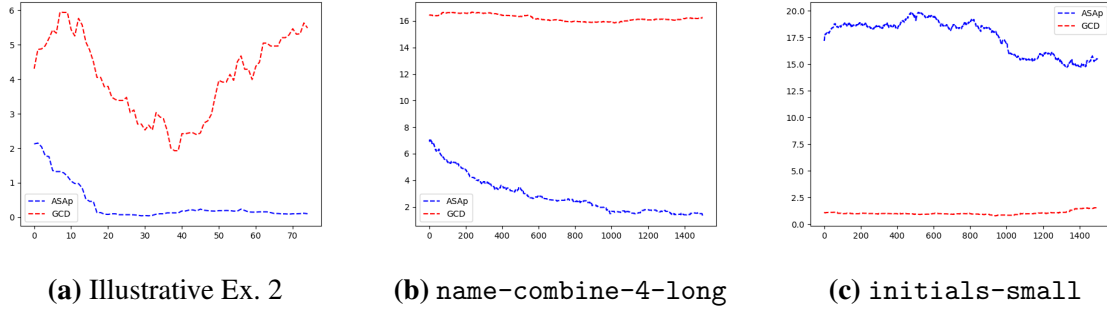


Figure 3.6. $KL(Q_{ASAp}||P)$ and $KL(Q_{GCD}||P)$

Measures.

We run both algorithms for 2,000 iterations/sample on each benchmark.

To assess convergence to the target distribution, we measure the Kullback–Leibler (KL) divergence between the distributions of GCD and ASAp from the target distribution Q for a given number of samples. Because the ideal GAD distribution $Q_{P,\mathcal{G}}$ is proportional to the original LLM’s distribution P for sequences allowed by a grammar $\mathcal{G}$, we can use the LLM’s distribution P on all observed samples as an estimate $Q_{P,\mathcal{G}}$. The quantity $KL(Q||P)$ only differs by a constant from the KL divergence between empirical distributions and the ideal GAD distribution:

$$KL(\tilde{Q}||P) = \mathbb{E}_{\tilde{Q}} \left[\log \frac{\tilde{Q}}{P} \right] = \mathbb{E}_{\tilde{Q}} \left[\log \frac{\tilde{Q}}{C \cdot Q_{P,\mathcal{G}}} \right] = \mathbb{E}_{\tilde{Q}} \left[\log \frac{\tilde{Q}}{Q_{P,\mathcal{G}}} \right] - \log C = KL(\tilde{Q}||Q_{P,\mathcal{G}}) - \log C$$

where $C = \sum_w \mathbb{1}[w \in \mathcal{L}(\mathcal{G})]P(w)$. Thus, $KL(\tilde{Q}||P)$ can be used to quantify the alignment between the empirical distributions of GCD and ASAp with the ideal GAD distribution.

For example, Fig. 3.6a shows convergence results for the first 75 iterations on the illustrative Ex. 2—i.e., the KL divergence for $\tilde{Q}_{ASAp}$ quickly converges to 0 whereas the one for $\tilde{Q}_{GCD}$ doesn’t.

We also compare the empirical expectations of the variables $\tilde{Q}_{GCD}$, $\tilde{Q}_{ASAp}$, and P . For example, Fig. 3.7a shows convergence results for the first 75 iterations on the illustrative Ex. 2—i.e., $\tilde{Q}_{ASAp}$ converges to the right expectation.

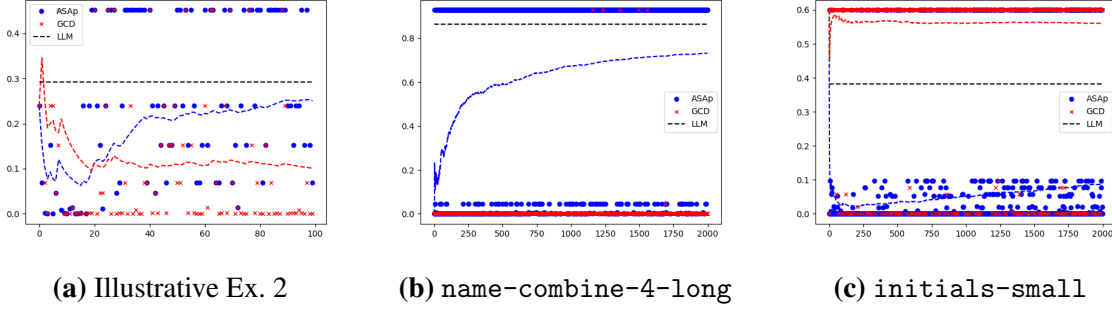


Figure 3.7. Expectations of $\tilde{Q}_{ASAp}$, $\tilde{Q}_{GCD}$, and P

Results.

Fig. 3.6b and Fig. 3.7b illustrate a benchmark in which our ASAp algorithm quickly converges to the target distribution. Fig. 3.6 depicts the KL divergence of a sliding window of size 500 (e.g., the points at $x=800$ denote the KL divergence of the samples 800-1300). Fig. 3.7 depicts all the samples from the experiment, as well as how the expectations converge (a point at $x=i$ denotes the empirical expectation on the first i samples). For this case the expectation for GCD stays very close to 0.

Similarly, Fig. 3.6c and Fig. 3.7c illustrate a benchmark in which our ASAp algorithm converges slowly. In this case, both ASAp and GCD are far from the target expectation (Fig. 3.7c), but because GCD happens to be biased towards the most likely outcome, it exhibits better KL divergence.

To better understand how the algorithms respectively converge, Fig. 3.8 plots for each benchmark category the expectations for each benchmark computed by GCD and ASAp against the target expectation of P after 2,000 iterations. The sum of least square difference between expectations computed by GCD and the expectations of P are 2.259 (SLIA), 1.852 (INV-BV4), and 0.109 (CP). The sum of squared differences between expectations computed by ASAp and the expectations of P are 1.242 (SLIA), 0.802 (INV-BV4), and 0.159 (CP). While we have too few points for CP to draw conclusions, the expectations computed by ASAp are much closer to the ones computed by GCD on SLIA and INV-BV.

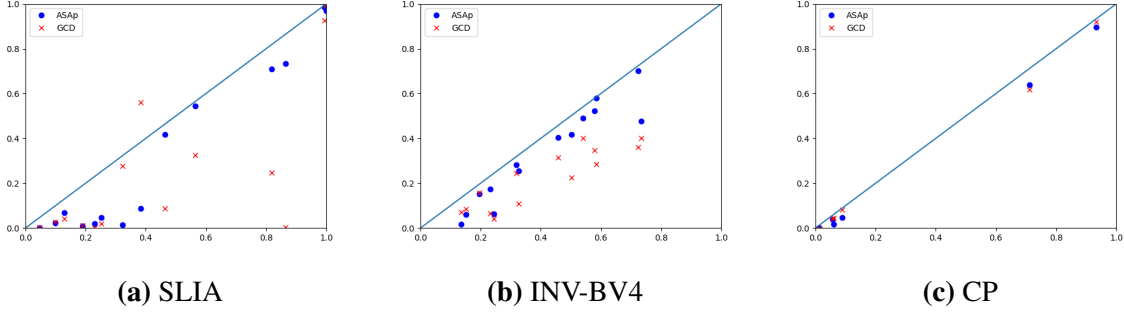


Figure 3.8. Scatter plots of $\tilde{Q}_{ASAp}$ (●) and $\tilde{Q}_{GCD}$ (×) vs. expectations of P after 2,000 samples. Proximity to the diagonal indicates proximity to the actual expectation—e.g., a ● at (0.45,0.4) indicates a benchmark where the empirical expectation of P was 0.45 and $\tilde{Q}_{ASAp}$ had converged to an expectation of 0.4 after 2,000 iterations.

While our work is interested in the theoretical convergence of the ASAp algorithm, we also report how the GCD and ASAp differ for solving the SLIA and INV-BV4 tasks—i.e., how many of the sampled programs are correct solutions to the given problem. GCD and ASAp solve approximately the same set of problems (there is just one SLIA benchmark for which ASAp returns a valid solution on one sample and GCD never does so). ASAp produces correct samples 38% more often than GCD (geomean), whereas for SLIA benchmarks that both tools can solve, ASAp produces correct samples 73% less often than GCD (geomean). Detailed results can be found in Tab. 3.1. These results are in line with the fact ASAp shows faster convergence on INV-BV4 benchmarks. For example, for the benchmark illustrated in Fig. 3.6b, ASAp returns the correct solution for 1588 samples, whereas GCD only returns the correct solution 12 times, whereas for the benchmark in Fig. 3.6c, ASAp returns the correct solution 69 times and GCD 363 times.

Table 3.1. Correctness of solutions for different algorithms.

	Benchmark	Correct ASAP	Correct GCD
SLIA	phone-3-long	0	0
	name-combine-2_short	171	319
	name-combine-2-long-repeat	0	0
	name-combine-4-short	20	11
	name-combine-4-long	1588	12
	lastname	285	1526
	firstname	1960	1997
	firstname_small	1754	1997
	reverse-name-long	1981	1859
	univ_1	67	40
	univ_1_short	605	1859
	univ_2_short	0	0
	dr-name	357	1654
	initials_small	69	363
	initials_long	540	1584
	initials_long-repeat	3	0
INV-BV	find_inv_ne_bvudiv1_4bit	0	0
	find_inv_bvugt_bvashr0_4bit	83	49
	find_inv_eq_bvlshr0_4bit	635	228
	find_inv_eq_bvand_4bit	1599	1305
	find_inv_bvule_bvurem0_4bit	1813	1710
	find_inv_bvsgt_bvor_4bit	11	10
	find_inv_bvugt_bvneg_4bit	84	36
	find_inv_bvule_bvurem1_4bit	143	227

3.4 Will ASAp still be more aligned than GCD after fine-tuning?

3.4.1 Experimental setup for fine-tuning

We adhere to the established QLoRA finetuning pipeline and create task-specific datasets of INV-BV4 and CP for instruction tuning. In line with our methodology, we incorporate in-context examples in the instruction tuning dataset to enhance the models’ performance in in-context learning. For each task, we independently finetune Mistral-7B, resulting in two versions of the model (for INV-BV4 and CP). We employ a standard train-validation-test split

of 70-10-20%. Instruction tuning is conducted on the training set, and model selection is based on the lowest validation loss. Key hyperparameters include a learning rate of $2e-4$, a warmup ratio of 0.03, a maximum sequence length of 2048, LoRA alpha of 32, LoRA dropout of 0.05, and LoRA rank of 64. The best checkpoints were at 328 and 536 steps for INV-BV and CP, respectively.

3.4.2 Additional Results

No significant differences in convergence rates post fine-tuning.

In Section 3.3, we evaluate ASAp and GCD on the base model Mistral-7B. A natural extension of this evaluation is determining whether ASAp retains its advantages over GCD after fine-tuning the base model on task-specific datasets, which optimizes the LLM for higher grammatical accuracy from the start.

In our fine-tuning step, we want to teach the LLM to assign higher probabilities to grammatical outputs for the specific task DSL. We randomly selected two INV-BV problems (`find_inv_bvsge_bvneg_4bit` and `find_inv_bvsge_bvgt_4bit` for INV-BV) and four CP problems (`CP_re_ptb_215`, `CP_re_ptb_434`, `CP_re_ptb_1627` and `CP_re_ptb_1643` for CP) from the test set, and instruction-tuned input-output pairs of prompt and output programs in the training set for the base model Mistral-7B. We obtained two fine-tuned models, one for INV-BV and one for CP.

We tested GCD and ASAp on the fine-tuned Mistral-7B on the randomly left-out problems and checked the convergence rates of the KL-divergence. The results from fine-tuned Mistral-7B did not show significant differences in terms of convergence compared to the base Mistral-7B. As done in Section 3.3, we computed the expectation for each benchmark obtained via GCD and ASAp after 2,000 iterations and compared it against the target expectation $Q^{P, \mathcal{G}}$ of GAD. The sum of squared differences between expectations computed by GCD and the expectations of $Q^{P, \mathcal{G}}$ are 0.677 (INV-BV4), 0.278 (CP), while ASAp achieved lower errors: 0.051 (INV-BV4), 0.201 (CP), indicating that ASAp more closely aligned with the exact GAD expectations. We

did not include SLIA as we did not have sufficient data for further fine-tuning.

Discussion and Limitations.

As predicted by our theorems, on most benchmarks the ASAp algorithm converges to the desired distribution P whereas GCD does not improve over time (i.e., it exhibits the bias described in this chapter).

While ASAp has no strong effect on solving downstream tasks, we observe that on instances where the convergence is prominent, ASAp ends up sampling correct solutions more often than GCD, which is what we expect when the LLM has “learned” how to solve the given task.

The key limitation of our work is the current slow convergence of the ASAp algorithm. In some benchmarks, even after 2,000 iterations the KL divergence barely improves and even though the expectation of $\tilde{Q}_{ASAp}$ is improving, it converges very slowly.

We highlight that the contributions of this chapter are discovering and formalizing the bias of existing constrained decoding approaches and proposing the first converging algorithm to address this problem. Now that we have identified the problem, there are many “low-hanging fruits” to improve our sampling strategy, which are great targets for future work—e.g., using forms of targeted beam search to bootstrap our sample set to better explore grammar paths and avoid sampling similar strings.

3.5 Related Work

Constrained Decoding

Past work has extensively explored *constrained decoding* algorithms, which modify the original decoding process of LLMs to ensure the output adheres to a user-specified regular [103, 157] or context-free language [15, 43, 55, 122, 135, 138, 147, 152] in a discrete space. Other works enforce hard output constraints using dynamic monitoring and verification methods [1, 95, 154] or by modifying beam search techniques to impose lexical constraints, which require

specific keywords to appear in the generated text [7, 72, 75, 99, 98, 124]. At a high level, these methods involve running the LLM decode in parallel with a monitoring scheme (e.g., parsing algorithms for CFGs) to identify which next tokens or beams can produce valid output sequences that meet the constraints. The decoder then masks out any tokens that would lead to invalid sequences, sampling only from the permissible ones.

To incorporate sequence-level soft semantic or contextual constraints, Amini et al. [5], Kumar et al. [88], Li et al. [94], Qin et al. [125] have applied gradient-based sampling techniques that relax those constraints to differentiable ones, used them as classifiers to further guide the decoding process. While these works guarantee that the decoded output meets the specified constraints (whether in the form of grammar, monitoring schemes, or differentiable functions), they often operate greedily and introduce bias into the output distribution in the way that has been discussed in this chapter. Depending on the application one considers, this problem may or may not affect downstream tasks, but as we have argued in this chapter, the bias can be quite prominent and sometimes affect downstream performance. Our adaptive decoding algorithm improves decoding over time by analyzing how previous samples led to nongrammaticality.

Constraint-Aligned Decoding

This chapter formally defines the problem of aligning the output distribution of an LLM in the presence of a constraint. We focus our attention on constraints expressed as grammars, but our definitions and algorithm apply to any constraint for which possible satisfaction (in our case grammaticality) can be evaluated in a left-to-right manner.

In some settings, one is interested in generating multiple outcomes with an LLM to approximate a distribution of interest [77, 127]—e.g., to generate a random number or a set of good test cases for a program. As we have shown, constrained decoding can heavily skew the LLM’s distribution and result in biasing the model towards certain constraint-matching sequences. While our work is at this point theoretical, now that the problem of aligning an LLM’s distribution with constraints has been defined, we expect advances in how sampling is

performed to quickly converge to better distributions faster (e.g., using beam search to quickly explore possible paths instead of just sampling).

This work formalized the likelihood misalignment problem of existing GCD approaches and provided an initial solution with provable asymptotic guarantees. Building on this work, subsequent studies have explored faster-converging methods [119, 6] and evaluated their empirical impact on performance [11, 92].

3.6 Conclusion

We have introduced a new analysis of the ideal target for constrained sampling from an LLM using a grammar, which we call grammar-aligned decoding (GAD). We proposed a new algorithm for GAD which we call ASAP that iteratively builds better approximations to the critical re-weighting term required for GAD: the expected future grammaticality. We analyzed the convergence of our proposed algorithm and demonstrated its effectiveness in relation to existing grammar-constrained decoding techniques on a set of benchmark code generation tasks. We analyzed and evaluated our approach using constraints enforced by a context-free grammar; however, extensions of our approach might be applied to more general classes of constraints for LLM decoding.

Acknowledgments

Chapter 3, in full, is based on the paper from Kanghee Park, Jiayu Wang, Taylor Berg-Kirkpatrick, Nadia Polikarpova, Loris D’Antoni. “Grammar-Aligned Decoding” published in Neural Information Processing Systems 37, pp. 24547-24568. 2024. The dissertation author is the primary investigator and author of this paper. The author sincerely thanks Paweł Parys for finding and fixing errors in the proof of Theorem 3.2.2.

Part II

Specification Synthesis

Chapter 4

Synthesizing Specifications

Specifications do not prescribe how to compute a desired result; rather, they define what constitutes the correct behavior of a program. Consequently, a specification is not necessarily equivalent to the full semantics of a concrete implementation. For instance, a specification for a list-sorting algorithm might require that elements with smaller keys appear earlier and that the output is a valid permutation of the input. However, it may intentionally leave the relative order of elements with equal keys unspecified, thereby permitting both stable and unstable sorting implementations.

Recently, the importance of formal specifications has grown significantly, as Large Language Models (LLMs) can synthesize implementations directly when provided with checkable specifications. The central difficulty is that high-quality specifications are expensive to author and maintain. As implementations evolve and intended behaviors are refined, manually written contracts risk becoming stale. Writing effective specifications requires domain expertise to identify key properties of interest, faithfully capture correctness, and enforce constraints strong enough to reject invalid implementations. Although numerous techniques have been proposed to automate specification generation, existing approaches remain restricted in scope and limited in the expressiveness of the properties they can infer. Furthermore, many rely on dynamic analysis—yielding likely specifications that hold across observed execution traces, but remain unsound in the general case.

In this chapter, we present the first customizable framework for synthesizing provably sound, most-precise (i.e., “best”) specifications from a given set of function definitions. Our framework can be used to mine specifications from code, but also to enable several applications where obtaining precise specifications is crucial—e.g., generating algebraic specifications for modular program synthesis [101], automating sensitivity analysis of programs [33], and enabling abstract interpretation for new abstract domains [161]. The engine/primitive that drives the framework is an algorithm for the following problem: *Given a query Ψ posed about a set of function definitions, find a most-precise conjunctive formula—expressed in the user-supplied, domain-specific language (DSL) $\mathcal{L}$ —that is implied by Ψ .*

Our algorithm synthesizes a set of properties $\{\varphi_i\}$ —each expressed in $\mathcal{L}$ —that are consequences of Ψ . Each φ_i is a most-precise $\mathcal{L}$ -property for Ψ : there is no $\mathcal{L}$ -property for Ψ that is strictly more precise than φ_i . Furthermore, the set $\{\varphi_i\}$ is exhaustive: the conjunction of the properties $\bigwedge_i \varphi_i$ is a best $\mathcal{L}$ -conjunction—i.e., no $\mathcal{L}$ -properties can be added to make the conjunction more precise.

This primitive is quite flexible. For instance, if the user desires a specification of the *input-output behavior* of a function `foo`, then Ψ is “ $out = \text{foo}(in)$.” Here the objective of specification synthesis is to perform a kind of “projection” operation that provides information about the behavior of `foo` solely in terms of the variables visible on entry and exit (i.e., the ones visible to a client of `foo`). On the other hand, if the user wants a specification of the properties of *combinations* of the stack operations `push` and `pop`, then Ψ is $os_1 = \text{push}(s_1, x_1) \wedge (os_2, x_2) = \text{pop}(s_2)$. In this case, when given an appropriate DSL $\mathcal{L}$, our tool SPYRO synthesizes the $\mathcal{L}$ -properties (i) $\text{eq}(os_1, s_2) \Rightarrow x_1 = x_2$ (the element ox obtained after $(os, ox) = \text{pop}(\text{push}(s, x))$ is the pushed element x), (ii) $\text{eq}(os_1, s_2) \Rightarrow \text{eq}(s_1, os_2)$ (the stack os obtained after $(os, ox) = \text{pop}(\text{push}(s, x))$ is the original stack s), and (iii) $\text{eq}(os_2, s_1) \wedge x_1 = x_2 \Rightarrow \text{eq}(os_1, s_2)$ (after $\text{pop}(s)$, a push of the popped element restores stack s).

Our approach is different from existing specification-mining algorithms in that it meets three important objectives: (i) **Expressiveness:** The synthesized specifications are not limited

to a fixed type, but are customizable through the user-provided DSL. (ii) **Soundness:** The synthesized specifications are sound for all inputs to the function definitions, not just for a specified set of test cases.¹ (iii) **Precision:** The synthesized specifications are precise in that no specification in the DSL $\mathcal{L}$ is more precise than the synthesized ones—both at the level of each individual $\mathcal{L}$ -property φ_i and at the level of the synthesized best $\mathcal{L}$ -conjunction $\bigwedge_i \varphi_i$.²

The Key Challenge

Finding a sound $\mathcal{L}$ -property for Ψ is trivial: “*true*” is always one. However, finding a *best* $\mathcal{L}$ -property is difficult because it is a kind of optimization problem, requiring SPYRO to find a *most-precise* solution. Proving most-preciseness of an $\mathcal{L}$ -property requires showing that synthesizing a more precise $\mathcal{L}$ -property is an *unrealizable* problem—i.e., no such property exists.

Key Ideas

Our algorithm is based on a form of counterexample-guided inductive synthesis (CEGIS) that iteratively accumulates positive and negative examples of Ψ . We build upon the algorithm for synthesizing abstract transformers proposed by Kalita et al. [81] and generalize it to the problem of synthesizing specifications. Our algorithm uses three key ideas (the first and third of which differ from Kalita et al.). First, although the overall goal of the framework is to obtain a specification as a conjunctive formula, the core algorithm that underlies the synthesis process “dives below” this goal and focuses on synthesizing a most-precise *individual conjunct* (i.e., a most-precise $\mathcal{L}$ -property for Ψ). The smaller size of individual $\mathcal{L}$ -properties compared to a full conjunctive $\mathcal{L}$ -conjunction is better aligned with the capabilities of synthesis tools. Second, to handle competing objectives of soundness and precision, the algorithm treats negative examples specially. Some negative examples *must* be rejected by the $\mathcal{L}$ -property we are

¹As explained in §4.1, the framework requires a definition of the semantics of the function symbols that appear in query Ψ (e.g., `push`, `pop`, `reverse`, etc.) and DSL $\mathcal{L}$. The specification obtained with our framework is *sound with respect to the supplied semantics*, but our implementation sometimes uses bounded or approximate semantics.

²Readers familiar with symbolic methods for abstraction interpretation [129] will recognize that our problem is an instance of the *strongest-consequence problem*. Given a formula Φ in logic $\mathcal{L}_1$ (with meaning function $\llbracket \cdot \rrbracket_1$), find the strongest formula Ψ in a different logic $\mathcal{L}_2$ (with meaning function $\llbracket \cdot \rrbracket_2$) such that $\llbracket \Phi \rrbracket_1 \subseteq \llbracket \Psi \rrbracket_2$.

synthesizing and some *may* be rejected. Third, to speed up progress, the algorithm accumulates *must* negative examples *monotonically*, so that once a sound $\mathcal{L}$ -property is identified, *may* negative examples change status to *must* negative examples, and the algorithm only searches for (better) $\mathcal{L}$ -properties that also reject those examples.

Our Framework

The core algorithm is a CEGIS loop that handles some negative-example classifications as “maybe” constraints, and guarantees progress via monotonic constraint hardening until an $\mathcal{L}$ -property is found that is both sound and precise. By repeatedly calling the core algorithm to synthesize incomparable $\mathcal{L}$ -properties, a most-precise $\mathcal{L}$ -conjunction is created.

The core algorithm relies on three simple primitives.

SYNTHESIZE: synthesizes an $\mathcal{L}$ -property that accepts a set of positive examples and rejects a set of negative examples; it returns $\perp$ if no such property exists.

CHECKSOUNDNESS: checks if the current $\mathcal{L}$ -property is sound; if is not, CHECKSOUNDNESS returns a new positive example that the property fails to accept.

CHECKPRECISION: checks if the current $\mathcal{L}$ -property is precise; if it is not, CHECKPRECISION returns a new $\mathcal{L}$ -property that accepts all positive examples, rejects all negative examples, and rejects one more negative example (which is also returned).

Our current implementation of each primitive relies on satisfiability modulo theory (SMT) solvers, limiting the scope of our framework. Nevertheless, as long as one has implementations for such primitives, the algorithm is sound. When the DSL $\mathcal{L}$ is finite, the algorithm is also complete.

Contributions

Our work makes the following contributions:

- A formal framework for the problem of synthesizing best $\mathcal{L}$ -conjunctions (§4.1).

- An algorithm for synthesizing best $\mathcal{L}$ -conjunctions (§4.2).
- A tool that we implemented to support our framework, called SPYRO. There are two instantiations of SPYRO: SPYRO[SKETCH] and SPYRO[SMT], which have different capabilities (see §4.3).
- An evaluation of SPYRO on a variety of benchmarks, showcasing four different applications of SPYRO (§4.5): mining specifications [97], generating algebraic specifications for modular program synthesis [101], automating sensitivity analysis of programs [33], and enabling abstract interpretation for new abstract domains [161].

§4.4 contains implementation details. §4.6 discusses related work. §4.7 concludes.

4.1 Problem Definition

In this section, we define the problem addressed by our framework. Throughout this chapter, we use a running example in which the goal is to synthesize interesting consequences of the following query, which allows obtaining properties of up to two calls of the list-reversal function.

$$ol_1 = \text{reverse}(l_1) \wedge ol_2 = \text{reverse}(l_2), \quad (4.1)$$

In particular, we are interested in identifying properties that are consequences of query formula (4.1) and are expressible in the DSL defined by the following grammar $\mathcal{L}_{list}$:

$$\begin{aligned} B &:= \top \mid AP \mid AP \vee AP \mid AP \vee AP \vee AP \\ AP &:= \text{isEmpty}(L) \mid \neg \text{isEmpty}(L) \mid \text{eq}(L, L) \mid \neg \text{eq}(L, L) \mid S \{ \leq \mid < \mid \geq \mid > \mid = \mid \neq \} S + \{0 \mid 1\} \\ S &:= 0 \mid \text{len}(L) \\ L &:= l_1 \mid l_2 \mid ol_1 \mid ol_2 \end{aligned} \quad (4.2)$$

An $\mathcal{L}$ -property is a property expressible in a DSL $\mathcal{L}$. We say that an $\mathcal{L}$ -property is *sound* if it is a consequence of—i.e., implied by—the given query formula Ψ . The goal of

our framework is to synthesize a set of incomparable *sound and most-precise* $\mathcal{L}$ -properties (i.e., a conjunctive specification), not just any $\mathcal{L}$ -properties. For example, the $\mathcal{L}_{list}$ -property $\text{len}(l_1) \leq \text{len}(ol_1)$ is sound but not most-precise ($\text{len}(l_1) = \text{len}(ol_1)$ is a more precise sound $\mathcal{L}_{list}$ -property).

In the rest of this section, we describe what a user of the framework has to provide to solve this problem, and what they obtain as output. The user needs to provide the following inputs:

Input 1: Query. The query formula consists of a finite set of atomic formulae $\Psi = \{\psi_1, \dots, \psi_n\}$ (denoting their conjunction). Each atomic formula σ_i is of the form $o^i = f^i(x_1^i, \dots, x_n^i)$, where o^i is an output variable, each x_j^i is an input variable, and f^i is a function symbol. In our running example, the query is given in Eq. (4.1).

Input 2: Grammar of $\mathcal{L}$ -properties. The grammar of the DSL $\mathcal{L}$ in which the synthesizer is to express properties. In our example, the DSL $\mathcal{L}_{list}$ is defined in Eq. (4.2).

Input 3: Semantics of function symbols. A specification of the semantics of the function symbols that appear in query Ψ (e.g., `reverse`) and in the DSL $\mathcal{L}$ (e.g., `len`).

We assume semantic definitions are given in—or can be translated to—formulas in some logic fragment. For pragmatic reasons, in our implementation semantic definitions are given as code that is then automatically transformed into first-order formulas. For instance, the semantics of `reverse` is given as a program in the SKETCH programming language [145] from which we automatically extract the following bounded semantics $\varphi_{\text{reverse}}^k(l, ol)$ for a given bound $k > 0$:

$$\begin{aligned} \varphi_{\text{reverse}}^0(l, ol) &:= \perp \\ \varphi_{\text{reverse}}^n(l, ol) &:= [\varphi_{\text{eq}}(l, []) \Rightarrow \varphi_{\text{eq}}(ol, [])] \wedge \\ &\quad \exists v_{hd}, l_{tl}, l' [\varphi_{\text{eq}}(l, v_{hd} :: l_{tl}) \Rightarrow \varphi_{\text{reverse}}^{n-1}(l_{tl}, l') \wedge \varphi_{\text{snoc}}(l', v_{hd}, ol)] \text{ if } n > 0 \end{aligned} \tag{4.3}$$

We discuss this limitation—i.e., that the semantics is bounded—and how we mitigate it in Section 4.5.

Let V_Ψ be the set of all variables in Ψ . We use ψ to denote the formula that exactly characterizes the space of valid models over the variables V_Ψ in Ψ . For example, let $\phi_{\text{reverse}}(l, ol)$ be the formula that exactly characterizes the result of reversing a list l and storing the result in ol —e.g., $(l, ol) = ([1, 2], [2, 1])$ is a valid model of ϕ_{reverse} . We use $\llbracket \phi \rrbracket$ to denote the set of models of a formula ϕ . Then, in our example $\llbracket \psi(l_1, ol_1, l_2, ol_2) \rrbracket = \llbracket \phi_{\text{reverse}}(l_1, ol_1) \wedge \phi_{\text{reverse}}(l_2, ol_2) \rrbracket$. Henceforth, we omit variables in formulas when no confusion should result, and merely write ψ .

Output: Best $\mathcal{L}$ -properties. The goal of our method is to synthesize a set of incomparable *sound and most precise* $\mathcal{L}$ -properties that are consequences of query Ψ . Ideally, the best $\mathcal{L}$ -property would be one that exactly describes ψ , but in general the language $\mathcal{L}$ might not be expressive enough to do so. We argue that this feature is actually a desirable one!³ The customizability of our approach via a DSL is what allows our work to focus on identifying small and readable properties (rather than complex first-order formulas), and to apply our method to different use cases (see §4.5).

Because in general there might not be an $\mathcal{L}$ -property that is equivalent to ψ , the goal becomes instead to find $\mathcal{L}$ -properties that tightly approximate ψ .

Definition 4.1.1 (A best $\mathcal{L}$ -property). *An $\mathcal{L}$ -property ϕ is a best $\mathcal{L}$ -property for a query Ψ if and only if (i) ϕ is sound with respect to Ψ : $\llbracket \psi \rrbracket \subseteq \llbracket \phi \rrbracket$. (ii) ϕ is precise with respect to Ψ and $\mathcal{L}$: $\neg \exists \phi' \in \mathcal{L}. \llbracket \psi \rrbracket \subseteq \llbracket \phi' \rrbracket \subset \llbracket \phi \rrbracket$. We use $\mathcal{P}(\Psi)$ to denote the set of all best $\mathcal{L}$ -properties for Ψ .*

When we refer to “a sound $\mathcal{L}$ -property,” soundness is always relative to some ψ . Strictly speaking, we should say “a ψ -sound $\mathcal{L}$ -property,” but ψ should always be clear from context.

A best $\mathcal{L}$ -property is a *strongest consequence* of ψ that is expressible in $\mathcal{L}$. Because $\mathcal{L}$ is constrained, there may be multiple, incomparable $\mathcal{L}$ -properties that are all strongest consequences of ψ —thus, we speak of *a* best $\mathcal{L}$ -property and not *the* best $\mathcal{L}$ -property. In our

³The idea of imposing a limit on the expressibility of the language in which properties can be stated is related to the concept of *inductive bias* in machine learning [107, §2.7]: When there is no inductive bias, “[a] concept learning algorithm is ... completely unable to generalize beyond the observed examples.”

running example, $\text{len}(l_1) = \text{len}(ol_1)$ is a best $\mathcal{L}$ -property and so is $\neg \text{eq}(ol_1, l_2) \vee \text{eq}(l_1, ol_2)$. (Stated as an implication: $\text{eq}(ol_1, l_2) \Rightarrow \text{eq}(l_1, ol_2)$.) The former states that the sizes of the input and output of reverse are the same, while the latter states that applying reverse twice to a list yields the same list.

The goal of this work is to find semantically minimal sets of incomparable best $\mathcal{L}$ -properties.

Definition 4.1.2 (Best $\mathcal{L}$ -conjunction). *A potentially infinite set of $\mathcal{L}$ -properties $\Pi = \{\varphi_i\}$ forms a best $\mathcal{L}$ -conjunction $\varphi_\wedge = \bigwedge_i \varphi_i$ for query Ψ if and only if (i) every $\varphi \in \Pi$ is a best $\mathcal{L}$ -property for Ψ ; (ii) every two distinct $\varphi_i, \varphi_j \in \Pi$ are incomparable—i.e., $\llbracket \varphi_i \rrbracket \setminus \llbracket \varphi_j \rrbracket \neq \emptyset$ and $\llbracket \varphi_j \rrbracket \setminus \llbracket \varphi_i \rrbracket \neq \emptyset$; (iii) the set is semantically minimal—i.e., for every best $\mathcal{L}$ -property $\varphi \in \mathcal{P}(\Psi)$ we have $\llbracket \varphi_\wedge \rrbracket \subseteq \llbracket \varphi \rrbracket$.*

While there can be multiple best $\mathcal{L}$ -conjunctions, they are all logically equivalent and they are all equivalent to a strongest $\mathcal{L}$ -conjunction. However, note that a strongest $\mathcal{L}$ -conjunction is not necessarily a best $\mathcal{L}$ -conjunction; it could contain $\mathcal{L}$ -properties that are not best and could potentially have repeated or redundant $\mathcal{L}$ -properties.

Theorem 4.1.3. *If $\varphi_\wedge$ is a best $\mathcal{L}$ -conjunction, then its interpretation coincides with the conjunction of all possible best properties: $\llbracket \varphi_\wedge \rrbracket = \llbracket \bigwedge_{\varphi \in \mathcal{P}(\Psi)} \varphi \rrbracket$.*

Proof. We first show the $\subseteq$ direction. The interpretation of the conjunction of all possible best properties must include $\varphi_\wedge$ because $\varphi_\wedge$ is semantically minimal—i.e., for every best $\mathcal{L}$ -property $\varphi \in \mathcal{P}(\Psi)$ we have $\llbracket \varphi_\wedge \rrbracket \subseteq \llbracket \varphi \rrbracket$. The $\supseteq$ direction holds because each conjunct of $\varphi_\wedge$ is a best $\mathcal{L}$ -property. $\square$

We are now ready to state our problem definition:

Definition 4.1.4 (Problem definition). *Given query Ψ , the concrete semantics ψ for the function symbols in Ψ , and a domain-specific language $\mathcal{L}$ with its corresponding semantic definition, synthesize a best $\mathcal{L}$ -conjunction for Ψ .*

As illustrated in Section 4.2, given query (4.1), the DSL in Eq. (4.2), and the semantic definitions of `reverse`, `isEmpty`, `len`, etc., our tool SPYRO synthesizes the set of $\mathcal{L}$ -properties shown below in Eq. (4.4), and establishes that the conjunction of these properties is a best $\mathcal{L}$ -conjunction. (For clarity, we write properties of the form $\neg a \vee b$ as $a \Rightarrow b$.)

$$\begin{array}{lll}
\text{len}(l_1) = \text{len}(ol_1) & \text{len}(l_2) = \text{len}(ol_2) & \text{eq}(l_2, ol_2) \vee \text{len}(l_2) > 1 \\
\text{len}(l_1) > 1 \vee \text{eq}(l_1, ol_1) & \text{eq}(ol_2, l_1) \Rightarrow \text{eq}(l_2, ol_1) & \text{eq}(ol_1, ol_2) \Rightarrow \text{eq}(l_1, l_2) \\
\text{eq}(ol_1, l_2) \Rightarrow \text{eq}(l_1, ol_2) & \text{eq}(l_1, l_2) \Rightarrow \text{eq}(ol_1, ol_2) &
\end{array} \tag{4.4}$$

Even though `reverse` is a simple function, its corresponding best $\mathcal{L}$ -conjunction (w.r.t. the DSL $\mathcal{L}$) for query (4.1) is non-trivial. For example, our approach can discover properties involving single function calls (e.g., `reverse` behaves like the identity function on a list of length 0 or 1), but also hyperproperties, i.e., properties involving multiple calls to the same function. For example, the property $\text{eq}(ol_1, l_2) \Rightarrow \text{eq}(l_1, ol_2)$ states that applying the `reverse` function twice to an input returns the same input, while the property $\text{eq}(ol_1, ol_2) \Rightarrow \text{eq}(l_1, l_2)$ shows that `reverse` is injective!

Moreover, because the user has control over the DSL $\mathcal{L}$, they can change the language in which properties are to be expressed. In particular, if the formulas returned by SPYRO are too complicated for the user's taste, they can modify $\mathcal{L}$ and reinvoke SPYRO until they are satisfied with the results.

Depending on the DSL, a best $\mathcal{L}$ -conjunction may need to be an infinite formula.

Example 3 (Infinite $\mathcal{L}$ -conjunction). *Consider again the running example, and assume we change the DSL to the one defined by the following grammar $\mathcal{L}_{\text{inf}}$:*

$$\text{Root} := l_1 = CL \wedge l_2 = ol_1 \Rightarrow ol_2 = CL \quad CL := [] \mid 1 :: CL \mid 2 :: CL$$

where “ $::$ ” denotes the infix `cons` operator. There exists only one best $\mathcal{L}_{\text{inf}}$ -conjunction, which

has an infinite number of conjuncts.

$$\begin{aligned}
l_1 &= [] \wedge l_2 = ol_1 \Rightarrow ol_2 = [] \\
l_1 &= 1 :: [] \wedge l_2 = ol_1 \Rightarrow ol_2 = 1 :: [] \\
l_1 &= 2 :: [] \wedge l_2 = ol_1 \Rightarrow ol_2 = 2 :: [] \\
l_1 &= 1 :: 1 :: [] \wedge l_2 = ol_1 \Rightarrow ol_2 = 1 :: 1 :: [] \\
l_1 &= 1 :: 2 :: [] \wedge l_2 = ol_1 \Rightarrow ol_2 = 1 :: 2 :: [] \\
&\dots
\end{aligned}$$

Our implementation focuses on DSLs for which this problem does not arise. Assumptions on DSLs are formally discussed in Section 4.2.

All the inputs to the framework are reusable. To synthesize a best $\mathcal{L}$ -conjunction for a different query Ψ that still operates over lists, one only needs to supply the semantic definition of the functions in Ψ , and (if needed) modify the variable names generated by nonterminal L of Eq. (4.2).

For example, for the function that takes a list and duplicates its entries $l_{out} = \text{stutter}(l)$ —e.g., $\text{stutter}([1, 2]) = [1, 1, 2, 2]$ —SPYRO synthesizes the following $\mathcal{L}$ -conjunction using the DSL $\mathcal{L}_{list}$.

$$\begin{aligned}
&\text{len}(l_{out}) = \text{len}(l) + 1 \vee \text{len}(l) > 1 \vee \text{isEmpty}(l_{out}) \\
&\text{len}(l_{out}) \leq 0 \vee \text{len}(l) = 1 \vee \text{len}(l_{out}) > \text{len}(l) + 1 \\
&\text{len}(l_{out}) \leq 0 \vee \text{len}(l_{out}) > \text{len}(l)
\end{aligned} \tag{4.5}$$

Because our DSL does not contain multiplication by 2, SPYRO could not synthesize the property that states that the length of the output list is twice the length of the input list. If we modify the DSL $\mathcal{L}_{list}$ to contain multiplication by 2 and the ability to describe when an element appears both in the input and output list, SPYRO successfully synthesizes the following new best

$\mathcal{L}$ -properties:

$$\begin{aligned}
& \text{len}(l_{out}) = 2 \cdot \text{len}(l) \\
& \forall v. (\exists x \in l. x = v) \Rightarrow (\exists x \in l_{out}. x = v) \quad \forall v. (\exists x \in l_{out}. x = v) \Rightarrow (\exists x \in l. x = v) \\
& \forall v. (\forall x \in l. x \leq v) \Rightarrow (\forall x \in l_{out}. x \leq v) \quad \forall v. (\forall x \in l_{out}. x \leq v) \Rightarrow (\forall x \in l. x \leq v) \\
& \forall v. (\forall x \in l. x \geq v) \Rightarrow (\forall x \in l_{out}. x \geq v) \quad \forall v. (\forall x \in l_{out}. x \geq v) \Rightarrow (\forall x \in l. x \geq v)
\end{aligned} \tag{4.6}$$

The ability to modify the DSL empowers the user of SPYRO with ways to customize the type of properties they are interested in synthesizing. As we will show in Section 4.5, customizing the DSL also allows us to use SPYRO for different applications and case studies—e.g., synthesizing abstract transformers and algebraic properties of programs.

4.2 An Algorithm for Synthesizing Best $\mathcal{L}$ -conjunctions

In this section, we present the main contribution of the work: an algorithm for synthesizing a best $\mathcal{L}$ -conjunction. The algorithm synthesizes one most-precise $\mathcal{L}$ -property at a time. It keeps track of the $\mathcal{L}$ -properties it has synthesized and uses this information to synthesize a new most-precise $\mathcal{L}$ -property that is incomparable to all the ones synthesized so far.

4.2.1 Positive and Negative Examples

Given query Ψ , the concrete semantics ψ for the function symbols in Ψ , and a domain-specific language $\mathcal{L}$ with its corresponding semantic definition, our algorithm synthesizes best $\mathcal{L}$ -properties and a best $\mathcal{L}$ -conjunction for Ψ using an example-guided approach.

Definition 4.2.1 (Examples). *Given a model e , we say that e is a positive example if $e \in \llbracket \psi \rrbracket$ and a negative example otherwise.*

Given a formula ϕ and an example e , we write $\phi(e)$ to denote $e \in \llbracket \phi \rrbracket$ and $\neg\phi(e)$ to denote $e \notin \llbracket \phi \rrbracket$. Given a set of examples E , we write $\phi(E)$ to denote $\bigwedge_{e \in E} \phi(e)$ and $\neg\phi(E)$ to denote $\bigwedge_{e \in E} \neg\phi(e)$.

Example 4. Given the query $ol_1 = \text{reverse}(l_1)$, the model that assigns l_1 to the list $[1, 2]$ and ol_1 to the list $[2, 1]$ is a positive example. For brevity, we use the notation $([1, 2], [2, 1])$ to denote such an example. The following examples are negative ones: $([1], [2])$, $([1, 2], [1, 3])$, $([1, 2], [1])$, $([], [1])$.

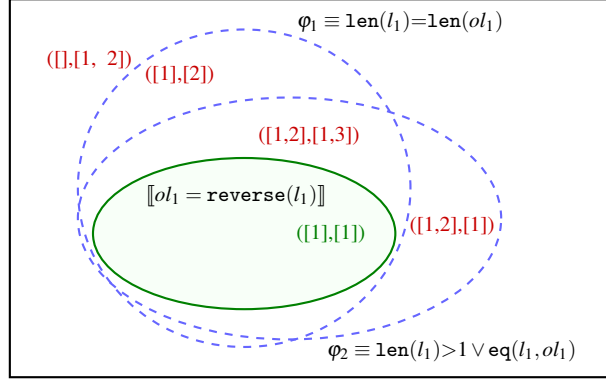
When considering the query from Eq. (4.1), which contains two calls on *reverse*, $([1, 2], [2, 1], [1], [1])$ is a positive example (where the values denote l_1 , ol_1 , l_2 , and ol_2 , respectively), while $([1, 2], [1, 3], [1], [1])$ and $([1, 2], [1, 3], [1], [0])$ are negative examples.

Intuitively, a best $\mathcal{L}$ -property must accept all positive examples while also excluding as many negative examples as possible.

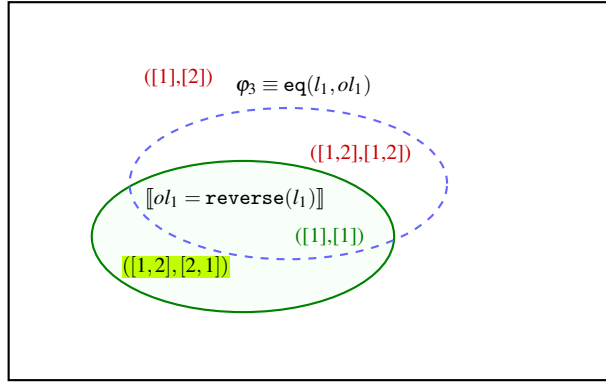
Positive examples can be treated as ground truth—i.e., a best $\mathcal{L}$ -property should always accept all positive examples—but negative examples are more subtle. First, there can be multiple, incomparable, best $\mathcal{L}$ -properties, each of which rejects a different set of negative examples. Second, there may be negative examples that no best $\mathcal{L}$ -property can reject—they are accepted by every best $\mathcal{L}$ -property.

Example 5. Consider again the query $ol_1 = \text{reverse}(l_1)$ and the diagram shown in Figure 4.1a. The $\mathcal{L}$ -properties $\varphi_1 \equiv \text{len}(l_1) = \text{len}(ol_1)$ and $\varphi_2 \equiv \text{len}(l_1) > 1 \vee \text{eq}(l_1, ol_1)$ are both best $\mathcal{L}$ -properties. While they both accept the positive example $([1], [1])$ and reject the negative example $([], [1, 2])$, we can see that φ_1 rejects the negative example $([1, 2], [1])$ whereas φ_2 accepts it. Similarly φ_2 rejects the negative example $([1], [2])$ whereas φ_1 accepts it. Finally, neither property rejects the negative example $([1, 2], [1, 3])$. In fact, no best $\mathcal{L}$ -property in the given DSL can reject this example, and thus any best $\mathcal{L}$ -conjunction will accept this negative example.

In most cases, we use common datatypes—integers or lists—as the domain of our examples; however, more complicated definitions are sometimes required. If we are interested in queries involving binary search trees (BSTs), a tree datatype can describe the syntactic structure of the examples, but cannot capture BST invariants, e.g., for every node n , all values in the left (resp. right) subtree of n must be $\leq$ (resp. $\geq$) n 's value. In our implementation, the set of



(a) Positive and Negative examples. Different negative examples are rejected by different best $\mathcal{L}$ -properties φ_1 and φ_2



(b) A possible soundness counterexample $([1,2], [2,1])$ produced when performing $\text{CHECKSOUNDNESS}(\varphi_3, \psi)$

Figure 4.1. The role of examples and CHECKSOUNDNESS .

valid BSTs is defined using the following **SKETCH** program—called a generator—that uses BST insertion operations to generate valid binary search trees:

```
generate_BST() := if (??) then emptyBST() else insert(generate_BST(), ??)
```

The code is then transformed into a bounded formula similar to the one in Eq. 4.3, but where the holes (i.e., $??$) at each recursive call are replaced with unknown variables. In this case, different values for the holes result in different BSTs. In the rest of the chapter, we assume that examples are only drawn from their valid domains regardless of how this domain is expressed.

4.2.2 Soundness and Precision

Now that we have established how examples relate to best $\mathcal{L}$ -properties, we can introduce the two key operations that make our algorithm work: `CHECKSOUNDNESS` and `CHECKPRECISION`. These two operations are similar to the ones used by Kalita et al. [81] to synthesize abstract transformers, and are used by the inductive-synthesis algorithm to determine whether an $\mathcal{L}$ -property is sound (i.e., a valid $\mathcal{L}$ -property) and precise (i.e., a best $\mathcal{L}$ -property). We modify the definition of `CHECKPRECISION` proposed by Kalita et al. [81] to account for already synthesized best $\mathcal{L}$ -properties and thus facilitate the synthesis of distinct best $\mathcal{L}$ -properties.

Checking Soundness

Given an $\mathcal{L}$ -property φ , `CHECKSOUNDNESS`(φ, ψ) checks whether φ is an overapproximation of ψ . In other words, `CHECKSOUNDNESS` checks if there exists a positive example $e^+ \in \llbracket \psi \rrbracket$ that is not accepted by φ ; it returns that example if it exists, and $\perp$ otherwise. The soundness check can be expressed as $\exists e^+. \neg \varphi(e^+) \wedge \psi(e^+)$.

Example 6 (`CHECKSOUNDNESS`). Consider again the query $ol_1 = \text{reverse}(l_1)$. We describe how `CHECKSOUNDNESS` operates using the example depicted in Figure 4.1b. The property $\varphi_3 \equiv \text{eq}(l_1, ol_1)$ is unsound because it does not overapproximate ψ . `CHECKSOUNDNESS`(φ_3, ψ) would return a positive example that is incorrectly rejected by φ_3 —e.g., $([1, 2], [2, 1])$. In contrast, `CHECKSOUNDNESS`(φ_1, ψ) on the property $\varphi_1 \equiv \text{len}(l_1) = \text{len}(ol_1)$ would instead return $\perp$ because the property φ_1 is sound (see Figure 4.1a).

Checking Precision

Given an $\mathcal{L}$ -property φ , a set of positive examples E^+ accepted by φ , a set of negative examples E^- rejected by φ , and a Boolean formula ψ denoting the set from which examples can be drawn, `CHECKPRECISION`(φ, ψ, E^+, E^-), checks whether there exist an $\mathcal{L}$ -property φ' and a negative example e^- such that: (i) φ' accepts all the positive examples in E^+ and rejects all the

negative examples in E^- ; (ii) $\psi(e^-)$ and φ' rejects e^- , whereas φ accepts e^- . Formally,

$$\exists \varphi', e^-. \psi(e^-) \wedge \varphi(e^-) \wedge \neg \varphi'(e^-) \wedge \varphi'(E^+) \wedge \neg \varphi'(E^-)$$

CHECKPRECISION can be thought of as a primitive that synthesizes a negative example and a formula that can reject such an example at the same time (or proves whether the synthesis problem does not admit a solution). The formula φ' is a witness that the negative example e^- can be rejected by some $\mathcal{L}$ -property. In our algorithm, the set ψ is used to ensure that the negative example produced by CHECKPRECISION is not already rejected by best $\mathcal{L}$ -properties we already synthesized.

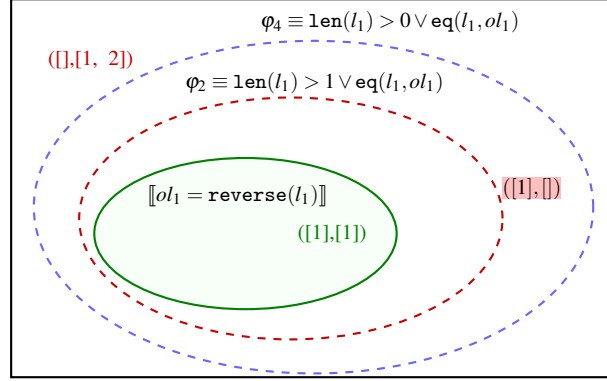
Example 7 (CHECKPRECISION). *Consider again the query $ol_1 = \text{reverse}(l_1)$. We describe how CHECKPRECISION operates using the example depicted in Figure 4.2. The property $\varphi_4 \equiv \text{len}(l_1) > 0 \vee \text{eq}(l_1, ol_1)$ is sound but imprecise.*

Figure 4.2a shows how running $\text{CHECKPRECISION}(\varphi_4, \neg\psi, \{([1], [1])\}, \{([], [1, 2])\})$ could for example return $\varphi_2 \equiv \text{len}(l_1) > 1 \vee \text{eq}(l_1, ol_1)$ and the negative example $e^- = ([1], [])$.

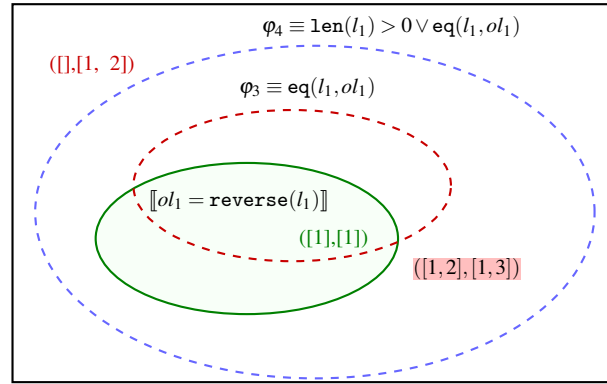
Figure 4.2b shows how running $\text{CHECKPRECISION}(\varphi_4, \neg\psi, \{([1], [1])\}, \{([], [1, 2])\})$ could alternatively return $\varphi_3 \equiv \text{eq}(l_1, ol_1)$ and the negative example $e^- = ([1, 2], [1, 3])$. While φ_3 satisfies all the requirements of CHECKPRECISION—i.e., it correctly classifies the current positive and negative examples—the formula φ_3 is unsound because it incorrectly rejects, among others, the positive example $([1, 2], [2, 1])$. Moreover, in this case CHECKPRECISION has returned the negative example $([1, 2], [1, 3])$, which—as observed in Example 5—is not rejected by any sound $\mathcal{L}$ -property!

Monotonicity of Sound $\mathcal{L}$ -properties

The fact that CHECKPRECISION can return (i) an unsound $\mathcal{L}$ -property, and (ii) a negative example that cannot be rejected by any sound $\mathcal{L}$ -property, makes designing a synthesis algorithm that can solve our problem challenging.



(a) Possible result $(\varphi_2, ([1], []))$ obtained when calling CHECKPRECISION on φ_4 and the current examples. φ_2 is *sound*.



(b) Possible result $(\varphi_3, ([1, 2], [1, 3]))$ obtained when calling CHECKPRECISION on φ_4 and the current examples. φ_3 is *unsound*.

Figure 4.2. Possible results produced by CHECKPRECISION

A key property that we exploit in our algorithm is that, under some assumptions on the language $\mathcal{L}$, once a sound $\mathcal{L}$ -property is found, there must exist a best $\mathcal{L}$ -property that implies it. This property allows searching for sound $\mathcal{L}$ -properties in a monotonic manner. Once a sound $\mathcal{L}$ -property is found, the search space can be narrowed down to a smaller set that includes all $\mathcal{L}$ -properties that are more precise than the one already found. We will show in Theorem 4.2.5 that this narrowing is guaranteed to be finite under some assumptions about the language $\mathcal{L}$.

We say that a relation $\preceq$ on a set X is a *well-quasi order* if $\preceq$ is a reflexive and transitive relation such that any infinite sequence of elements $x_0, x_1, x_2, \dots$ from X contains an increasing pair $x_i \preceq x_j$ with $i < j$. If the consequence relation $\Rightarrow$ for language $\mathcal{L}$ is a well-quasi order, we have that any descending sequence of sound $\mathcal{L}$ -properties cannot be infinite—i.e., for any

$\mathcal{L}$ -property φ , there exist only finitely many $\mathcal{L}$ -properties that imply φ . Moreover, a well-quasi order has no infinite anti-chains (i.e., there are no infinite sequences of pairwise incomparable elements). Clearly, if $\mathcal{L}$ is *finite*, then $\Rightarrow$ is a well-quasi order, but finiteness is not a necessary condition. For example, consider the absolute-value function $o = \text{abs}(x)$, and a grammar $\mathcal{L}_{inf}$ that defines properties of the form $-20 \leq x \leq 10 \Rightarrow o \leq N$ (for any natural number N). The set of properties is infinite, but $\Rightarrow$ is a well-quasi order on the set of sound $\mathcal{L}_{inf}$ -properties—i.e., for any concrete value of N , it is only possible to decrease the value of N and strengthen the property a finite number of times.

Lemma 4.2.2 (Monotonicity of Sound $\mathcal{L}$ -properties). *If $\Rightarrow$ is a well-quasi order on the set of $\mathcal{L}$ -properties, for every sound $\mathcal{L}$ -property φ , there exists a best $\mathcal{L}$ -property φ' such that $\varphi' \Rightarrow \varphi$.*

Proof. For any $\varphi \in \mathcal{L}$, because $\Rightarrow$ is a well-quasi order on the set of $\mathcal{L}$ -properties, the set of distinct properties $\Pi = \{\varphi' \mid \varphi' \Rightarrow \varphi \wedge \varphi \neq \varphi'\}$ is finite. Then, either φ is a best $\mathcal{L}$ -property or at least one of the properties in Π is. $\square$

Consequently, if φ is a sound $\mathcal{L}$ -property that rejects a set of negative examples E^- , there must exist a best $\mathcal{L}$ -property that also rejects the examples in E^- . This property lets us infer when a set of negative examples *must* be rejected by a best $\mathcal{L}$ -property (i.e., after a sound property is found).

4.2.3 Synthesizing One Most Precise $\mathcal{L}$ -property

We are now ready to describe the method used to synthesize an individual best $\mathcal{L}$ -property (Algorithm 8). The procedure SYNTHPROPERTY takes as input

- ψ : a formula describing the behavior of the program,
- ψ : a formula describing the domain from which examples can be drawn,
- φ_{init} : an initial sound $\mathcal{L}$ -property that rejects all of the examples in E_{must}^- ,

- E^+ : a set of positive examples that *must* be accepted by the returned $\mathcal{L}$ -property,
- E_{must}^- : a set of negative examples that *must* be rejected by the returned $\mathcal{L}$ -property,

and produces as output

- a sound $\mathcal{L}$ -property φ that accepts all the examples in E^+ , rejects all the examples in E_{must}^- ,
- an updated set of positive examples E^+ that the synthesized $\mathcal{L}$ -property accepts,
- an updated set of negative examples E_{must}^- that the synthesized $\mathcal{L}$ -property rejects.

We next discuss the procedure SYNTHESIZE, which SYNTHPROPERTY uses to identify $\mathcal{L}$ -properties that behave correctly on a finite set of examples.

Synthesis from examples

Besides CHECKSOUNDNESS and CHECKPRECISION, which we have already discussed, SYNTHPROPERTY uses the procedure $\text{SYNTHESIZE}(E^+, E^-)$, which returns—when one exists—an $\mathcal{L}$ -property φ that accepts all of the examples in E^+ (i.e., $\varphi(E^+)$) and rejects all of the examples in E^- (i.e., $\neg\varphi(E^-)$). If no such property exists—i.e., the synthesis problem is *unrealizable*— $\text{SYNTHESIZE}(E^+, E^-) = \perp$.

Example 8 (Example-based Synthesis). *Figure 4.1a showed that there may be negative examples that no sound $\mathcal{L}$ -property can reject. Our algorithm uses E_{may}^- to handle such examples and make sure that they never end up in E_{must}^- . With $\mathcal{L}_{list}$ from Eq. (4.2), if $E^+ = \{([1], [1])\}$, $E_{must}^- = \{\}$, and $E_{may}^- = \{([1, 2], [1, 3])\}$, then $\text{SYNTHESIZE}(E^+, E_{must}^- \cup E_{may}^-)$ can return the $\mathcal{L}$ -property $len(l_1) = 1$, which is clearly unsound. In this case, CHECKSOUNDNESS will repeatedly add positive E^+ (without changing E_{must}^- and E_{may}^-) until $\text{SYNTHESIZE}(E^+, E_{must}^- \cup E_{may}^-)$ returns $\perp$ —e.g., when $E^+ = \{([1], [1]), ([1, 2], [2, 1])\}$. Even when each member of a set of negative examples can be rejected (individually) by a sound $\mathcal{L}$ -property, there may not exist a single sound $\mathcal{L}$ -property that rejects all members of the set. For example, the negative*

Algorithm 8: SYNTHPROPERTY ($\psi, \psi, \phi_{init}, E^+, E_{must}^-$)

```
1  $\phi, \phi_{last} \leftarrow \phi_{init}; E_{may}^- \leftarrow \emptyset$  // initial sound  $\mathcal{L}$ -property that rejects  $E_{must}^-$ 
2 while true do
3    $e^+ \leftarrow \text{CHECKSOUNDNESS}(\phi, \psi)$  // check soundness first
4   if  $e^+ \neq \perp$  then
5      $E^+ \leftarrow E^+ \cup \{e^+\}$  // unsound, update positive examples
6      $\phi' \leftarrow \text{SYNTHESIZE}(E^+, E_{must}^- \cup E_{may}^-)$  // learn a new  $\mathcal{L}$ -property
7     if  $\phi' \neq \perp$  then
8        $\phi \leftarrow \phi'$  // new candidate  $\mathcal{L}$ -property
9     else
10       $\phi \leftarrow \phi_{last}; E_{may}^- \leftarrow \emptyset$  // no sound  $\mathcal{L}$ -property rejects  $E_{may}^-$ , revert
11      to  $\phi_{last}$ 
12   else
13      $E_{must}^- \leftarrow E_{must}^- \cup E_{may}^-; E_{may}^- \leftarrow \emptyset$  // sound, so  $E_{may}^-$  example is added to  $E_{must}^-$ 
14      $\phi_{last} \leftarrow \phi$  // remember sound  $\mathcal{L}$ -property
15      $e^-, \phi' \leftarrow \text{CHECKPRECISION}(\phi, \neg\psi \wedge \psi, E^+, E_{must}^-)$  // sound, check precision
16     if  $e^- \neq \perp$  then
17        $E_{may}^- \leftarrow E_{may}^- \cup \{e^-\}$  // update negative examples
18        $\phi \leftarrow \phi'$  // new candidate formula
19     else
20       return  $\phi, E^+, E_{must}^- \cup E_{may}^-$  // sound and precise
```

examples $\{([1], [2]), ([1, 2], [1])\}$ in Figure 4.1a cannot both be rejected by a single best $\mathcal{L}$ -property. If $E^+ = \{([1, 2], [2, 1])\}$, $E_{must}^- = \{([1], []), ([1], [2])\}$, and $E_{may}^- = \{([1, 2], [1])\}$, then $\text{SYNTHESIZE}(E^+, E_{must}^- \cup E_{may}^-)$ will return $\perp$.

Preserved Invariants

We describe SYNTHPROPERTY and the invariants it maintains.

Invariant 1: At the beginning of each loop iteration, the $\mathcal{L}$ -property ϕ accepts all the examples in the current set E^+ and rejects all the examples in the current set $E_{must}^- \cup E_{may}^-$.

In each iteration, SYNTHPROPERTY checks if the current property is sound using CHECKSOUNDNESS (line 3). If the property is sound, it is then checked for precision using CHECK-

PRECISION (line 14). The algorithm terminates once the property is sound and precise (line 19).

After a new candidate is found by CHECKPRECISION, the set E_{may}^- stores one negative example that might be rejected by a sound $\mathcal{L}$ -property. If CHECKSOUNDNESS determines that φ is sound, the example in E_{may}^- is added to E_{must}^- (line 12) in accordance with Lemma 4.2.2. If a new negative example is returned by CHECKPRECISION, together with a new property φ' , the set E_{may}^- is reinitialized, and φ' becomes the current property to check in the next iteration (lines 16 and 17).

The next invariant is guaranteed by Lemma 4.2.2 and by the fact that negative examples are added to E_{must}^- only when a sound property is found (line 12).

Invariant 2: There exists a sound $\mathcal{L}$ -property that rejects all the negative examples in E_{must}^- .

When E_{must}^- is augmented with E_{may}^- in line 12, SYNTHPROPERTY stores in φ_{last} the current sound $\mathcal{L}$ -property that rejects all of the negative examples in E_{must}^- . In the next iteration, if a new positive example is returned by CHECKSOUNDNESS, the positive examples are updated, and a new property is synthesized using SYNTHESIZE (line 6). If SYNTHESIZE cannot synthesize a property that is consistent with the new set of examples, SPYRO discards the conflicting negative example in E_{may}^- and reverts to the last sound property it found (line 10).

Invariant 3: At the beginning of each loop iteration, the $\mathcal{L}$ -property φ_{last} accepts all of the examples in the current set E^+ and rejects all of the examples in the current set E_{must}^- .

Monotonically increasing the set of negative examples E_{must}^- when a sound $\mathcal{L}$ -property is found is one of the contributions of our algorithm. While the algorithm is sound even without line 12, this step prevents the algorithm from often oscillating between multiple best $\mathcal{L}$ -properties throughout its execution. We found that this optimization gives a 3.06% speedup to the algorithm (see §4.5.5).

Example 9 (Algorithm 8 Run). Consider again the query $ol_1 = \text{reverse}(l_1)$, and a call to SYNTHPROPERTY ($\psi, \top, \top, \emptyset, \emptyset$)—i.e., $\varphi_{init} = \psi = \top$ and $E^+ = E_{must}^- = \emptyset$.

Iteration 1. The run starts with $\text{CHECKSOUNDNESS}(\top, \psi)$ (line 3), which returns $\perp$ because the property $\top$ is sound. Then all negative examples in E_{may}^- are added to E_{must}^- (line 12), but because $E_{may}^- = E_{must}^- = \emptyset$, both sets remain empty. $\text{CHECKPRECISION}(\top, \neg\psi, \emptyset, \emptyset)$ (line 14) returns a new candidate $\varphi_1 \equiv eq(l_1, ol_1)$ with a negative example $([], [1, 2])$ (line 14). This negative example is added to E_{may}^- (line 16), and the code goes back to line 3.

Iteration 2. $\text{CHECKSOUNDNESS}(\varphi_1, \psi)$ returns a positive example $([1, 2], [2, 1])$, E^+ is updated to $\{([1, 2], [2, 1])\}$ (line 5), and $\text{SYNTHESIZE}(\{([1, 2], [2, 1])\}, \{([], [1, 2])\})$ (line 6) returns a new candidate $\varphi_2 \equiv len(l_1) \neq 0$.

Iteration 3. $\text{CHECKSOUNDNESS}(\varphi_2, \psi)$ returns a positive example $([], [])$, E^+ is updated to $\{([1, 2], [2, 1]), ([], [])\}$ (line 5). $\text{SYNTHESIZE}(E^+, \{([], [1, 2])\})$ (line 6) returns a new candidate $\varphi_3 \equiv len(l_1) > 1 \vee eq(l_1, ol_1)$. E_{may}^- is unchanged, and the code goes to line 3.

Iteration 4. $\text{CHECKSOUNDNESS}(\varphi_3, \psi)$ returns $\perp$ because φ_3 is sound. Because the synthesizer found a sound property, the negative example in E_{may}^- is added to E_{must}^- (line 12)—i.e., $E_{must}^- = \{([], [1, 2])\}$. Although φ_3 is a best $\mathcal{L}$ -property, $\text{CHECKPRECISION}(\varphi_3, \neg\psi, E^+, \{([], [1, 2])\})$ (line 14) returns $\varphi_4 \equiv len(l_1) = len(ol_1)$ with a negative example $([1, 2], [1])$. E_{may}^- is set to $\{([1, 2], [1])\}$, and the code goes to line 3. The current set of examples is not enough for CHECKPRECISION to prove that φ_3 is indeed a best $\mathcal{L}$ -property—i.e., CHECKPRECISION was able to satisfy all requirements from §4.2.2 by finding a different, incomparable $\mathcal{L}$ -property φ_4 and an additional negative example.

Iteration 5. $\text{CHECKSOUNDNESS}(\varphi_4, \psi)$ returns $\perp$, and the negative example $([1, 2], [1])$ in E_{may}^- is added to E_{must}^- (line 12). $\text{CHECKPRECISION}(\varphi_4, \neg\psi, E^+, E_{must}^-)$ (line 14) returns $\perp$, which means that we found a sound and precise $\mathcal{L}$ -property. SYNTHPROPERTY terminates with $E_{\delta}^- = \emptyset$, $\varphi \equiv \varphi_4 \equiv len(l_1) = len(ol_1)$, $E^+ = \{([1, 2], [2, 1]), ([], [])\}$, and $E_{must}^- = \{([], [1, 2]), ([1, 2], [1])\}$.

The last invariant merely states that all the examples in E_{may}^- are elements of ψ .

Invariant 4: For every example $e \in E_{may}^-$, we have $\psi(e)$.

We say that a sound $\mathcal{L}$ -property ϕ is *precise for ψ with respect to ψ* if there does not exist a negative example $e^- \in \llbracket \psi \rrbracket$ and $\mathcal{L}$ -property ϕ' such that $\psi \Rightarrow \phi' \Rightarrow \phi$ and ϕ' rejects e^- , whereas ϕ accepts e^- . The following lemma characterizes the behavior of SYNTHPROPERTY.

Lemma 4.2.3 (Soundness and Relative Precision of SYNTHPROPERTY). *If SYNTHPROPERTY terminates, it returns a sound $\mathcal{L}$ -property ϕ that accepts all the examples in E^+ , rejects all the examples in $E_{must}^- \cup E_{may}^-$, and is precise for ψ with respect to ψ .*

Proof. The fact that ϕ accepts all the examples in E^+ and rejects all the examples in $E_{may}^- \cup E_{must}^-$ follows from Invariant 1. The condition for ϕ_{init} guarantees that Invariant 1 holds on the first iteration, and the definitions of SYNTHESIZE and CHECKPRECISION guarantee that Invariant 1 holds whenever ϕ is updated. The $\mathcal{L}$ -property ϕ at line 19 must be sound and precise with respect to ψ because both CHECKSOUNDNESS and CHECKPRECISION have returned $\perp$. Returning $\perp$ from CHECKSOUNDNESS(ϕ, ψ) (line 3) guarantees that ϕ is sound. Returning $\perp$ from CHECKPRECISION($\phi, \psi, E^+, E_{must}^-$) (line 14) guarantees that ϕ is precise for ψ with respect to ψ . $\square$

4.2.4 Synthesizing a Most-Precise $\mathcal{L}$ -conjunction

In this section, we present SYNTHSTRONGESTCONJUNCTION (Algorithm 9), which uses SYNTHPROPERTY to synthesize a best $\mathcal{L}$ -conjunction of $\mathcal{L}$ -properties.

On each iteration, SYNTHSTRONGESTCONJUNCTION maintains a conjunction of best $\mathcal{L}$ -properties $\phi_\wedge$, and uses SYNTHPROPERTY to synthesize a best $\mathcal{L}$ -property that rejects some negative examples that are still accepted by $\phi_\wedge$ (i.e., negative examples in $\phi_\wedge \wedge \neg\psi$). It also maintains the set of positive examples E^+ that have been observed so far.

Each iteration performs three steps: First, it uses SYNTHPROPERTY to try to find an $\mathcal{L}$ -property ϕ that rejects new negative examples E_{must}^- that no $\mathcal{L}$ -property synthesized so far could reject—i.e., by calling SYNTHPROPERTY with $\psi = \phi_\wedge \wedge \neg\psi$ (line 5).

Algorithm 9: SYNTHSTRONGESTCONJUNCTION(ψ)

```
1  $\varphi_{\wedge} \leftarrow \top$  // conjunction of best  $\mathcal{L}$ -properties
2  $\Pi \leftarrow \emptyset$  // set of best  $\mathcal{L}$ -properties
3  $E^+, E_{may}^- \leftarrow \emptyset$  // initialize examples
4 while  $\top$  do
    // find sound  $\varphi$  that rejects examples in  $E_{must}^-$  that are still in  $\varphi_{\wedge}$ 
5     $\varphi, E^+, E_{must}^- \leftarrow \text{SYNTHPROPERTY}(\psi, \varphi_{\wedge}, \top, E^+, \emptyset)$ 
6    if  $E_{must}^- = \emptyset$  then
7         $e^- \leftarrow \text{ISSAT}(\varphi_{\wedge} \wedge \neg \varphi)$  // check if  $\varphi$  improves  $\varphi_{\wedge}$ 
8        if  $e^- \neq \perp$  then
9             $E_{must}^- \leftarrow \{e^-\}$ 
10        else
11            return  $\Pi$  // return best  $\mathcal{L}$ -conjunction
    // refine  $\varphi$  to reject more examples outside  $\varphi_{\wedge}$  if possible
12     $\varphi, E^+, - \leftarrow \text{SYNTHPROPERTY}(\psi, \top, \varphi, E^+, E_{must}^-)$ 
13     $\Pi \leftarrow \Pi \cup \{\varphi\}$  //  $\varphi$  is a best  $\mathcal{L}$ -property
14     $\varphi_{\wedge} \leftarrow \varphi_{\wedge} \wedge \varphi$  // Update conjunction
```

Second, it checks whether φ rejects some example that was not rejected by $\varphi_{\wedge}$ (line 7). If it does not, the algorithm terminates, and returns the $\mathcal{L}$ -properties in Π synthesized so far. They are all best $\mathcal{L}$ -properties and their conjunction is a best $\mathcal{L}$ -conjunction.

Finally, if we reach line 12, we know that φ rejects negative examples in E_{must}^- that $\varphi_{\wedge}$ did not reject. Furthermore, because of the guarantees of SYNTHPROPERTY, φ is precise with respect to $\varphi_{\wedge}$ —i.e., no sound $\mathcal{L}$ -property φ' exists that can reject more negative examples in $\varphi_{\wedge}$ than φ could reject. However, there may be a more precise $\mathcal{L}$ -property that rejects more negative examples outside of $\varphi_{\wedge}$ that φ does not reject, while still rejecting all the negative examples in E_{must}^- . The call to SYNTHPROPERTY in line 12 addresses this issue; it computes a best $\mathcal{L}$ -property starting from φ and makes sure that the $\mathcal{L}$ -property obtained rejects everything in E_{must}^- while allowing negative examples to be computed anywhere—i.e., $\psi = \top$. (Compare this call with the one on line 5, which only allows negative examples to be drawn from $\varphi_{\wedge}$.) Because precision with respect to $\top$ implies actual precision, we have that when $\psi = \top$, if SYNTHPROPERTY terminates, it returns the best $\mathcal{L}$ -property φ for ψ by Lemma 4.2.3.

Example 10 (Algorithm 9 Run). *Revisiting the query $ol_1 = \text{reverse}(l_1)$, assume that $\varphi_\wedge$ has been updated with the recently synthesized property $\text{len}(l_1) = \text{len}(ol_1)$. We now describe the execution of $\text{SYNTHPROPERTY}(\psi, \varphi_\wedge, \top, \{([1,2], [2,1]), ([], [])\}, \emptyset)$ —i.e., $\psi = \varphi_\wedge$, $\varphi_{\text{init}} = \top$, $E^+ = \{([1,2], [2,1]), ([], [])\}$ and $E_{\text{must}}^- = \emptyset$.*

Iteration 1. *The run starts with $\text{CHECKSOUNDNESS}(\top, \psi)$ (line 3), which returns $\perp$ because the property $\top$ is sound. Then all negative examples in E_{may}^- are added to E_{must}^- (line 12), but because $E_{\text{may}}^- = E_{\text{must}}^- = \emptyset$, both sets remain empty. $\text{CHECKPRECISION}(\top, \varphi_\wedge \wedge \neg\psi, E^+, \emptyset)$ (line 14) returns a new candidate $\varphi_1 \equiv \text{len}(l_1) = 0 \vee \neg \text{eq}(l_1, ol_1)$ with a negative example $([1,2], [1,2])$ (line 14). Note that this negative example satisfies $\psi = \varphi_\wedge$. This negative example is added to E_{may}^- (line 16), and the code goes back to line 3.*

Iteration 2. $\text{CHECKSOUNDNESS}(\varphi_1, \psi)$ returns a positive example $([1], [1])$, E^+ is updated to $\{([1,2], [2,1]), ([], []), ([1], [1])\}$ (line 5), and $\text{SYNTHESIZE}(E^+, E_{\text{must}}^- \cup E_{\text{may}}^-)$ (line 6) returns a new candidate $\varphi_2 \equiv \text{len}(l_1) > 1 \vee \text{eq}(l_1, ol_1)$.

Iteration 3. $\text{CHECKSOUNDNESS}(\varphi_2, \psi)$ returns $\perp$, and the negative example E_{may}^- is added to E_{must}^- (line 12), but because E_{may}^- is empty, the set E_{must}^- remains the same. $\text{CHECKPRECISION}(\varphi_2, \varphi_\wedge \wedge \neg\psi, E^+, E_{\text{must}}^-)$ (line 14) returns $\perp$, which means that we found a sound and precise $\mathcal{L}$ -property. So SYNTHPROPERTY terminates with $\varphi \equiv \varphi_2 \equiv \text{len}(l_1) > 1 \vee \text{eq}(l_1, ol_1)$, $E^+ = \{([1,2], [2,1]), ([], []), ([1], [1])\}$, $E_{\text{must}}^- = \{([1,2], [1,2])\}$ and $E_\delta^- = \emptyset$.

Every iteration of the loop computes a best $\mathcal{L}$ -property, which is conjoined onto $\varphi_\wedge$ (line 14).

Invariant 5: Π is a set of incomparable best $\mathcal{L}$ -properties.

We are now ready to show that $\text{SYNTHSTRONGESTCONJUNCTION}$ is sound.

Theorem 4.2.4 (Soundness of $\text{SYNTHSTRONGESTCONJUNCTION}$). *If $\text{SYNTHSTRONGESTCONJUNCTION}$ terminates, it returns a best $\mathcal{L}$ -conjunction for ψ .*

Proof. Notice that Π is a best $\mathcal{L}$ -conjunction if Invariant 5 holds and Π is semantically minimal

at line 11. We can give an inductive argument to show that Invariant 5 always holds. At the beginning, Invariant 5 holds trivially just after line 3. For the inductive step, assuming that Invariant 5 holds at line 12 of SYNTHSTRONGESTCONJUNCTION, we show that it still holds after $\Pi \leftarrow \Pi \cup \{\varphi\}$ in line 13. First, Lemma 4.2.3 guarantees that each φ returned from SYNTHPROPERTY at line 12 of SYNTHSTRONGESTCONJUNCTION is a best $\mathcal{L}$ -property for Ψ . Also, E_{must}^- must be non-empty for the control to reach line 12 of SYNTHSTRONGESTCONJUNCTION. The examples in E_{must}^- must have been discovered by the call to SYNTHPROPERTY in line 5, where the domain from which examples can be drawn is $\varphi_\wedge$ (the second argument of the call). Thus, φ rejects all of the negative examples in E_{must}^- while $\varphi_\wedge$ accepts them, which establishes that φ is not weaker than each $\varphi' \in \Pi$. Consequently, Invariant 5 holds for $\Pi \cup \{\varphi\}$.

We now prove that Π is semantically minimal at line 11—i.e., when SYNTHSTRONGESTCONJUNCTION returns, there is no best (most importantly, no sound) $\mathcal{L}$ -property φ' such that $\llbracket \varphi_\wedge \wedge \varphi' \rrbracket \subset \llbracket \varphi_\wedge \rrbracket$. For SYNTHSTRONGESTCONJUNCTION to return, $\text{ISAT}(\varphi_\wedge \wedge \neg\varphi)$ at line 7 must return $\perp$, which would imply that $\llbracket \varphi_\wedge \rrbracket = \llbracket \varphi_\wedge \wedge \varphi \rrbracket$. If Π is not semantically minimal, then there would exist a sound $\mathcal{L}$ -property φ' such that $\llbracket \varphi_\wedge \wedge \varphi' \rrbracket \subset \llbracket \varphi_\wedge \rrbracket = \llbracket \varphi_\wedge \wedge \varphi \rrbracket$, but this is not possible because the property φ returned by SYNTHPROPERTY at line 5 is precise with respect to $\varphi_\wedge$ (Lemma 4.2.3). $\square$

Note:

In some settings, we might know a priori that certain $\mathcal{L}$ -properties hold, and we would not want to waste time synthesizing them. In such a situation, the formula $\varphi_\wedge$ in Algorithm 9 can be initialized to hold those properties, in which case Algorithm 9 would synthesize only best $\mathcal{L}$ -properties that are not subsumed by $\varphi_\wedge$. For example, consider synthesizing a specification for two calls of the list-reversal function, as shown in Section 4.1. We can initialize $\varphi_\wedge$ with trivial properties—such as $eq(l_1, l_2) \Rightarrow eq(ol_1, ol_2)$ —that are true of every function definition. Furthermore, after synthesizing a property like $eq(l_2, ol_1) \Rightarrow eq(ol_2, l_1)$, we can also include the symmetric property $eq(l_1, ol_2) \Rightarrow eq(ol_1, l_2)$ in the conjunction. This approach enables us to

effectively filter out redundant and trivial specifications during the synthesis process.

4.2.5 Completeness

We observe that in `SYNTHPROPERTY`, because positive examples in E^+ are never removed, any property stronger than a property that fails `CHECKSOUNDNESS` at line 3 is never considered again. Consequently, the sequence of unsound $\mathcal{L}$ -properties in an execution of `SYNTHPROPERTY` is non-strengthening. Thus, if a non-strengthening sequence of unsound $\mathcal{L}$ -properties can only be finite, `SYNTHPROPERTY` can only find finitely many unsound $\mathcal{L}$ -properties.

Another key observation about `SYNTHPROPERTY` is that at line 14 if `CHECKPRECISION` $(\varphi, \dots)$ returns a sound property φ' with a negative example e^- , `CHECKSOUNDNESS` will return $\perp$ in the next iteration, and the negative example e^- will be added to E_{must}^- (from E_{may}^- in line 12). Therefore, any property weaker than φ —i.e., φ on the current iteration— will never be considered during this execution of `SYNTHPROPERTY`. (Recall from the definition of `CHECKPRECISION` $(\varphi, \dots)$ that e^- satisfies φ .) Thus, if a non-weakening sequence of sound $\mathcal{L}$ -properties can only be finite, `SYNTHPROPERTY` can only find finitely many sound $\mathcal{L}$ -properties.

Based on the above two observations, Theorem 4.2.5 provides a sufficient condition for our algorithm to terminate when DSL $\mathcal{L}$ generates an infinite set of formulas.

Theorem 4.2.5 (Relative Completeness). *Suppose that $\Rightarrow$ is a well-quasi order on the set of sound $\mathcal{L}$ -properties. Let $\Leftarrow$ denote the inverse of $\Rightarrow$, and suppose that $\Leftarrow$ is a well-quasi order on the set of unsound $\mathcal{L}$ -properties. If `SYNTHESIZE`, `CHECKSOUNDNESS` and `CHECKPRECISION` are decidable on $\mathcal{L}$, then `SYNTHPROPERTY` and `SYNTHSTRONGESTCONJUNCTION` always terminate.*

Proof. Intuitively, the theorem states that no infinite sequence of sound (resp. unsound) properties exists, thus guaranteeing that in finitely many steps `SYNTHPROPERTY` will find a best $\mathcal{L}$ -property and `SYNTHSTRONGESTCONJUNCTION` will find a best $\mathcal{L}$ -conjunction. For instance, because

each conjunct of a best $\mathcal{L}$ -conjunction must be incomparable via $\Rightarrow$, conjuncts of an infinite best $\mathcal{L}$ -conjunction would have to form an infinite anti-chain of $\mathcal{L}$ -properties, which contradicts the assumption that $\Rightarrow$ is a well-quasi order on sound $\mathcal{L}$ -properties.

A key property of SYNTHPROPERTY is that any property stronger than a property that fails CHECKSOUNDNESS at line 3 is never considered again, because positive examples in E^+ are never removed. Consequently, the sequence of unsound $\mathcal{L}$ -properties in an execution of SYNTHPROPERTY is non-strengthening. Because $\Leftarrow$ is a well-quasi order on the set of unsound $\mathcal{L}$ -properties, any non-strengthening sequence of unsound $\mathcal{L}$ -properties can only be finite.

We can now move to the case where an $\mathcal{L}$ -property φ reaches CHECKPRECISION at line 14. In this case, we have two possibilities (illustrated in Fig. 4.2): (i) CHECKPRECISION returns an unsound property φ'' (and a negative example e_1^-); (ii) CHECKPRECISION returns a sound property φ'' (and a negative example e_2^-). Case (i) can only happen finitely many times because only finitely many unsound $\mathcal{L}$ -properties can be found during an execution of SYNTHPROPERTY. For case (ii), CHECKSOUNDNESS will return $\perp$ in the next iteration, and the negative example e_2^- will be added to E_{must}^- (from E_{may}^-); therefore, any property weaker than φ will never be considered during this execution of SYNTHPROPERTY. Therefore, the sequence $\varphi_1 = \varphi'', \varphi_2, \dots, \varphi_n$ of sound $\mathcal{L}$ -properties obtained by CHECKPRECISION is non-weakening. Because $\Rightarrow$ is a well-quasi order on the set of sound $\mathcal{L}$ -properties, any non-weakening sequence of sound $\mathcal{L}$ -properties must be finite.

The algorithm is also complete if we remove line 12, but in this case the proof is a little more involved. Again, we need to worry about case (ii). Because φ'' is sound, in the next iteration, SYNTHPROPERTY will call CHECKPRECISION again on φ'' . If CHECKPRECISION returns another property (i.e., φ'' is not precise), it may look like the algorithm can get stuck in a loop. However, that cannot happen for the following reason. Let $\varphi_1 = \varphi'', \varphi_2, \dots, \varphi_n$ be a sequence of $\mathcal{L}$ -properties obtained by consecutive calls to CHECKPRECISION, each of which has returned a precision example $e_1^-, \dots, e_n^-$. Note that for every set of examples $\{e_1^-, \dots, e_i^-\}$, every φ_j (for $j > i$) rejects those examples, while every φ_j (for $j \leq i$) does not. Because $\Rightarrow$ is a

well-quasi order on the set of sound $\mathcal{L}$ -properties, such a sequence of $\mathcal{L}$ -properties obtained by consecutive calls to `CHECKPRECISION` must be finite. The sequence can be terminated by either finding an unsound property (see point (i) for that case), or by `CHECKPRECISION` returning $\perp$, in which case the algorithm terminates.

Turning to `SYNTHSTRONGESTCONJUNCTION`, each best $\mathcal{L}$ -property is a sound $\mathcal{L}$ -property; thus, if there were an infinite sequence of best $\mathcal{L}$ -properties, there would be an infinite sequence of sound $\mathcal{L}$ -properties, violating the assumption that $\Rightarrow$ is a well-quasi order on the set of sound $\mathcal{L}$ -properties. Consequently, because Π in `SYNTHSTRONGESTCONJUNCTION` is a set of incomparable sound $\mathcal{L}$ -properties (Invariant 5), the loop on lines 4–14 cannot repeat indefinitely. Moreover, the fact that each call on `SYNTHPROPERTY` terminates guarantees that `SYNTHSTRONGESTCONJUNCTION` always terminates. $\square$

Above, our argument about the second observation involved line 12 of `SYNTHPROPERTY`. Nevertheless, Theorem 4.2.5 remains valid even if we eliminate line 12 from `SYNTHPROPERTY`—i.e., line 12 is an optimization.

During each iteration, `SYNTHPROPERTY` either adds a new positive example to E^+ or adds a new negative example to E_{must}^- . As a result, the number of iterations is also limited by the size of the example domain.

Corollary 4.2.6. *Suppose that either $\mathcal{L}$ contains finitely many formulas, or the example domain is finite. If `SYNTHESIZE`, `CHECKSOUNDNESS` and `CHECKPRECISION` are decidable on $\mathcal{L}$, then `SYNTHPROPERTY` and `SYNTHSTRONGESTCONJUNCTION` always terminate.*

4.3 Implementation

We implemented our framework in a tool called `SPYRO`. Following §4.1, `SPYRO` takes the following inputs: (i) A query Ψ for which `SPYRO` is to find a best $\mathcal{L}$ -conjunction. (ii) The context-free grammar of the DSL $\mathcal{L}$ in which properties are to be expressed. (iii) A specification, as a logical formula, of the concrete semantics of the function symbols in Ψ and $\mathcal{L}$. `SYNTHESIZE`

and CHECKPRECISION may be undecidable synthesis problems in general, but we show that these primitives can be implemented in practice using program-synthesis tools that are capable of both finding solutions to synthesis problems and establishing that a problem is unrealizable (i.e., it has no solution).

We implemented two versions of SPYRO: SPYRO[SMT] supports problems in which semantics are definable as SMT formulas, and SPYRO[SKETCH] supports arbitrary problems but relies on the bounded encoding of program semantics of the SKETCH language [145]. For the current implementations of SPYRO[SMT] and SPYRO[SKETCH], it is necessary to give the inputs in slightly different forms. In particular, input (iii) is provided to SPYRO[SMT] in SMT-Lib format, whereas it is provided to SPYRO[SKETCH] as a piece of code in the SKETCH programming language.

In SPYRO[SMT], CHECKSOUNDNESS is just an SMT query, and SYNTHESIZE and CHECKPRECISION can be expressed as SYGUS problems. For the latter two primitives, SPYRO runs two SYGUS solvers in parallel and returns the result of whichever terminates first: (i) CVC5 (v. commit b500e9d) [12], which is optimized for finding solutions to SYGUS synthesis queries, and (ii) a re-implementation of the constraint-based unrealizability-checking technique from [76] that is specialized for finding whether the output of SYNTHESIZE and CHECKPRECISION is $\perp$.

In SPYRO[SKETCH], SYNTHESIZE, CHECKSOUNDNESS, and CHECKPRECISION are all implemented by calling the SKETCH synthesizer (v. 1.7.6) [145]. We describe how each primitive is encoded in SKETCH in Section 4.4 and how SKETCH’s encoding affects soundness in Section 4.5.

Timeouts

We use a timeout threshold of 300 seconds for each call to SYNTHESIZE, CHECKSOUNDNESS, and CHECKPRECISION. If any such call times out, SYNTHSTRONGESTCONJUNCTION returns the current $\mathcal{L}$ -conjunction, together with an indication that it might not be a best $\mathcal{L}$ -conjunction. (However, each of the individual conjuncts in the returned $\mathcal{L}$ -conjunction is a best

$\mathcal{L}$ -property.)

Additional Tooling

In our evaluation, we used Dafny [93] to verify that the properties synthesized by SPYRO[SKETCH] were sound for inputs beyond the bounds considered by SKETCH. Furthermore, for the SYGUS benchmarks only, we invoked CVC5 to verify whether the properties obtained by SPYRO[SKETCH] *exactly* characterized the function (which is a sufficient condition for an answer to be a “best” answer).

4.4 The Encoding of our Primitives in SKETCH

We discuss below how SPYRO uses the features of SKETCH to implement the DSL $\mathcal{L}$ specified by the given grammar, and the operations SYNTHESIZE, CHECKSOUNDNESS, and CHECKPRECISION.

The grammar of the DSL $\mathcal{L}$ is compiled to a SKETCH generator. A generator is a function with holes that allows one to build complex programs via recursion. Intuitively, holes are used to allow the synthesizer to make choices about what to output. In our case, holes are used to select which productions are expanded at each point in a derivation tree, and recursion is used to move down the derivation tree. SKETCH allows one to provide a recursion bound to make synthesis tractable, and SPYRO’s grammar syntax allows one to provide such bounds for each of the nonterminals in the grammar. Fig. 4.3 shows how the generator `genAP` for the productions of nonterminal AP in Equation 4.2 can be specified.

SKETCH uses assertions (which are often written within so-called harnesses) to impose constraints on the programs produced by a generator. In our setting, assertions can be used to produce properties that accept all positive examples and reject all must negative examples.

SYNTHESIZE is implemented using a call to SKETCH that finds an $\mathcal{L}$ -property produced by the generator that satisfies all the assertions.

We also use generators to write data-structure constructors, which are functions that can

```

1 generator boolean genAP(list l1, list l2, list ol1, list ol2) {
2   int t = ??;
3   list var_L_1 = genL(l1, l2, ol1, ol2);
4   if (t = 0) return isEmpty(var_L_1);
5   if (t = 1) return !isEmpty(var_L_1);
6   ...
7   int var_S_1 = genS(l1, l2, ol1, ol2);
8   int var_S_2 = genS(l1, l2, ol1, ol2);
9   return { |var_S_1 (< | ≤ | > | ≥ | = | ≠) var_S_2 + ??(1) | };
10 }

```

Figure 4.3. SKETCH generator `genAP` for the productions of nonterminal AP in the DSL in Equation 4.2. Lines 3, 7, and 8 are calls to generators of other nonterminals, and lines 4, 5, and 9 mimic productions in the grammar. A production is selected based on the value synthesized by SKETCH for t at line 2. (SKETCH can synthesize a different value for each recursive call.)

```

1 harness void checkPrecision() {
2   // find a negative example in psi
3   list lnegex = list_constructor();
4   boolean out;
5   psi(lnegex, ..., out);
6   assert out;
7   // lnegex accepted by current property
8   phi(lnegex, ..., out);
9   assert out;
10  // lnegex not accepted by new property
11  phiprime(lnegex, ..., out);
12  assert !out;
13 }

```

```

1 harness void pos_example_1() {
2   ... // example description
3   boolean out;
4   phiprime(lposex1, ..., out);
5   assert out;
6 }

```

```

1 harness void neg_example_1() {
2   ... // example description
3   boolean out;
4   phiprime(lnegex1, ..., out);
5   assert !out;
6 }

```

Figure 4.4. SKETCH code for CHECKPRECISION with positive and negative examples. `lnegex` is the negative example e^- generated by the generator `list_constructor` that would witness imprecision, `phi` is a function describing the current property φ , `psi` is a function describing the predicate ψ , and `phiprime` is a function calling the generator `genD` to find an $\mathcal{L}$ -property φ' that together with `lnegex` would form a witness for imprecision.

generate data structures that satisfy desired invariants—e.g., `list_constructor()` can generate any valid list up to a certain length. We bound the size of the inputs in our experiments.

Given a logical specification of a concrete function, we use SKETCH as a satisfiability solver. For a given candidate $\mathcal{L}$ -property, CHECKSOUNDNESS uses SKETCH to attempt to synthesize a soundness counterexample.

CHECKPRECISION asks SKETCH to find a property that satisfies all the assertions, together with a new negative example that the newly synthesized property can reject and the previous property accepts (see Figure 4.4).

Term minimization: To avoid generating unnecessarily complex formulas, we use the minimization feature of SKETCH to look for terms of smallest size (both for SYNTHESIZE and CHECKPRECISION). Because we are only interested in the final formula, it is enough to apply minimization only in the second call to SYNTHPROPERTY in Alg. 9 (line 12).

4.5 Evaluation

We evaluated the effectiveness of SPYRO through four case studies: specification mining (§4.5.1), synthesis of algebraic specifications for modular synthesis (§4.5.2), automating sensitivity analysis (§4.5.3), and enabling new abstract domains (§4.5.4). For each case study, we describe how we collected the benchmarks, present a *quantitative analysis* of the running time and effectiveness of SPYRO, and a *qualitative analysis* of the synthesised $\mathcal{L}$ -conjunctions. In §4.5.5, we describe additional experiments to identify what parameters affect SPYRO’s algorithm.

We ran all experiments on an Apple M1 8-core CPU with 8GB RAM. All results in this section are for the median run, selected from the results of three runs ranked by their overall synthesis time.

4.5.1 Application 1: Specification Mining

We considered a total of 45 general *specification-mining* problems to evaluate SPYRO: 7 syntax-guided synthesis (SYGUS) problems from the SYGUS competition [4], where the

semantics of operations is expressed using SMT formulas; 24 type-directed synthesis problems from SYNQUID [123], where the semantics of operations is expressed using SKETCH; and 14 problems we designed to cover missing interesting types of properties (11 had their semantics expressed using SKETCH and 3 had semantics expressed using SMT formulas). Cumulatively, we have 10 benchmarks for which the semantics of operations is expressed using SMT formulas, and 35 benchmarks for which the semantics of operations is expressed using SKETCH. For the SYGUS and SYNQUID benchmarks, we “inverted” the roles from the original benchmarks: given the reference implementation, SPYRO synthesized a specification. Each input problem consists of a set of functions (1 to 14 functions per problem, and the size of each function ranges from 1 to 30 lines of code per function). The largest problem contains 14 functions (8 list functions and 6 queue functions) and the file contains a total of 140 lines of code. Most functions are recursive and can call each other—e.g., `dequeue` calls `reverse`, which calls `snoc`, which calls `cons`.

For each set of similar benchmarks, we designed a DSL that contained operations that could describe interesting properties for the given set of problems. The construction of each DSL depended on syntactic information from the code: the number, types, and names of input and output variables, constants, and function symbols used in the code. We included operations that are commonly used in each category, such as equality, size primitives, and emptiness checking, but avoided problem-specific information. For benchmarks involving data structures with structural invariants (e.g., stacks, queues, and binary search trees), we provided data-structure constructors that guaranteed that functions were only invoked with data-structure instances that satisfied the invariants. The exact grammars are described in Appendix A.1. A DSL designed for a specific problem domain was often reused by modifying what function symbols could appear in the DSL. Overall, we created 7 distinct grammars for 14 different SYGUS and arithmetic problems; 10 grammars for 72 SYNQUID problems; and 3 grammars for 7 Stack and Queue problems. Although all but one of the DSLs are finite, they are still large languages; our finite DSLs can express between 4 thousand and 14.8 trillion properties, thus making the problem of synthesizing specifications challenging.

Quantitative Analysis, Part 1: Performance

SPYRO[SMT] synthesized best $\mathcal{L}$ -conjunctions for 6/10 benchmarks for which the semantics was expressed using SMT formulas. It took less than 6 minutes each for it to solve the successful examples, and timed out on the remaining 4 benchmarks (`max4`, `arrSearch3`, `abs` with the grammar from Eq. (A.15), and hyperproperties of `diff`)—SPYRO[SMT] typically times out when the synthesis algorithm requires many examples.

Although we did not consider this option in our initial set of benchmarks, for the 4 benchmarks on which SPYRO[SMT] failed, we also encoded the semantics of the function symbols in Ψ and $\mathcal{L}$ using SKETCH. SPYRO[SKETCH] could synthesize properties for all 4 benchmarks, and guaranteed that 1/4 were best $\mathcal{L}$ -conjunctions (with respect to SKETCH’s bounded semantics), but for the other 3 benchmarks (`max4`, `arrSearch3` and `diff`) SPYRO[SKETCH] timed out on a call to CHECKPRECISION. However, the 3 $\mathcal{L}$ -conjunctions obtained by the time CHECKPRECISION timed out were indeed best $\mathcal{L}$ -conjunctions: although most-preciseness was not shown by SPYRO[SKETCH] within the timeout threshold, we found—using an SMT solver—that the $\mathcal{L}$ -conjunctions in hand on the synthesis round on which the timeout occurred defined the *exact* semantics of the functions of interest, which implies they were best $\mathcal{L}$ -conjunctions. The 1 problem for which SPYRO[SKETCH] established most-preciseness terminated within 5 minutes. For the other 3 problems, if we disregard the last iteration—the one on which most-preciseness of the $\mathcal{L}$ -conjunction was to be established—SPYRO[SKETCH] found a best $\mathcal{L}$ -conjunction within 10 minutes.

SPYRO[SKETCH] could synthesize properties for 35/35 benchmarks for which the semantics was expressed using SKETCH, and guaranteed that 34/35 were best $\mathcal{L}$ -conjunctions. It took less than 10 minutes to solve each List and Tree benchmark, except for the branch problem—SPYRO[SKETCH] took about 30 minutes to find the best $\mathcal{L}$ -conjunction, but failed to show most-preciseness. It took less than 15 minutes to solve each Stack, Queue, and Integer-Arithmetic benchmark. For `nonlinSum`, SPYRO[SKETCH] was able to synthesize in 900 seconds

a best $\mathcal{L}$ -conjunction from a grammar that contains ≈ 14.8 trillion properties (see Eq. A.16).

As a baseline, we compared the running time of SPYRO to an estimate of the running time of an algorithm that enumerates all sound properties in $\mathcal{L}$. For each benchmark, we estimated the cost of enumerating all terms in the grammar and checking for their soundness by multiplying the size $|\mathcal{L}|$ of the language generated by the grammar by the average running time of each call to CHECKSOUNDNESS observed when running SPYRO on the same benchmark. While SPYRO demonstrated a small estimated speedup for smaller problems like `emptyQueue` (i.e., $3.5\times$, with $|\mathcal{L}| = 64$), SPYRO was 2-5 orders of magnitude faster than the baseline for problems with large languages— $10^4 \leq |\mathcal{L}| \leq 6 \cdot 10^7$ —and 8-10 orders of magnitude faster than the baseline for the two problems with very large languages— $10^{10} \leq |\mathcal{L}| \leq 1.5 \cdot 10^{13}$.

Together, SPYRO[SKETCH] and SPYRO[SMT] synthesized properties for 41/45 benchmarks (45/45 if we consider the 4 benchmarks rewritten using a SKETCH semantics), and guaranteed that 40 were best $\mathcal{L}$ -conjunctions (44 if we consider the 4 benchmarks rewritten using a SKETCH semantics and our further analysis using an SMT solver).

Quantitative Analysis, Part 2: Soundness

To assess whether the properties synthesized by SPYRO[SKETCH] were indeed sound beyond the given input bound considered by SKETCH, we used an external verifier: Dafny [93] (a general purpose semi-automatic verifier). Dafny successfully verified that 23/35 $\mathcal{L}$ -conjunctions synthesized by SPYRO[SKETCH] on non-SYGUS benchmarks were sound without any manual input from us. We could increase this number to 33/35 by providing invariants or some logical axioms to Dafny—e.g., $(\forall l. \text{len}(l) \geq 0)$. Dafny failed to verify properties synthesized from `enqueue` and `reverse`, which require a more expressive $\mathcal{L}$ to describe the order of elements.

Qualitative Analysis

Fig. 4.5 shows the properties synthesized by SPYRO on one of our three runs. In the SYGUS benchmarks, “[SKETCH]” denotes cases in which SPYRO[SKETCH] terminated with a

SyGuS	List	Binary Tree
$o = \max2(x_1, x_2)$ $o = x_1 \vee o = x_2$ $x_2 = x_1 \vee o > x_1 \vee o > x_2$ $o = \max3(x_1, x_2, x_3)$ $(x_3 \geq x_1 \wedge x_3 \geq x_2) \Rightarrow o = x_3$ $x_2 = x_3 \vee o > x_2 \vee o > x_3$ $x_1 < x_2 \vee o = x_1 \vee o = x_3$ $x_2 = x_1 \vee o > x_2 \vee o > x_1$ $x_1 < x_3 \vee o = x_1 \vee o = x_2$ $o = \text{diff}(x, y)$ $x = y + o \vee y = x + o$ $x \geq y \Rightarrow x = o + y$ $x > y + y \vee y = x + o \vee 0 < x + o$ $x \leq y \Rightarrow y = o + x$ $o_1 = \text{diff}(x_1, y_1)$ [SKETCH] $o_2 = \text{diff}(x_2, y_2)$ $(y_2 = x_2 \wedge x_1 = y_1) \Rightarrow o_2 = o_1$ $(x_2 = x_1 \wedge y_1 = y_2) \Rightarrow o_1 = o_2$ $(x_1 = y_1 \wedge o_1 = o_2) \Rightarrow y_2 = x_2$ $(o_1 = o_2 \wedge x_2 = y_2) \Rightarrow y_1 = x_1$ $(x_2 = y_1 \wedge y_2 = x_1) \Rightarrow o_2 = o_1$ $o = \text{abs}(x)$ (Eq. A.15) $-x \geq x \Rightarrow -o = x \quad x \geq -x \Rightarrow o = x$ $o = \text{abs}(x)$ (Eq. A.16) [SKETCH] $-x + 2o = 0 \vee x - o + 2 = 0$ $\vee x + 2o - 1 > 0$ $-x + o = 0 \vee 2x + 2o = 0$	$l_{out} = \text{append}(l_1, l_2)$ $\text{len}(l_{out}) = \text{len}(l_1) + \text{len}(l_2)$ $\text{eq}(l_2, l_{out}) \vee \text{eq}(l_1, l_{out}) \vee \text{len}(l_{out}) > 1$ $\text{eq}(l_2, l_{out}) \vee \text{len}(l_{out}) = \text{len}(l_2) + 1 \vee \text{len}(l_1) > 1$ $\text{eq}(l_1, l_{out}) \vee \text{len}(l_{out}) > \text{len}(l_1) + 1 \vee \text{len}(l_2) = 1$ $l_{out} = \text{deleteFirst}(l, v)$ $\text{eq}(l, l_{out}) \vee \text{len}(l) = \text{len}(l_{out}) + 1$ $(\exists x \in l. x < v) \Rightarrow (\exists x \in l_{out}. x < v)$ $(\forall x \in l. x \neq v) \Rightarrow \text{eq}(l, l_{out})$ $(\forall x \in l. x \geq v) \Rightarrow (\forall x \in l_{out}. x \geq v)$ $(\exists x \in l. x < v) \vee \text{len}(l) = \text{len}(l_{out}) + 1$ $\vee (\forall x \in l_{out}. x \neq v)$ $(\exists x \in l. x > v) \Rightarrow (\exists x \in l_{out}. x > v)$ $(\exists x \in l. x = v) \Rightarrow \text{len}(l) = \text{len}(l_{out}) + 1$ $(\forall x \in l. x \leq v) \Rightarrow (\forall x \in l_{out}. x \leq v)$ $l_{out} = \text{delete}(l, v)$ $(\forall x \in l. x \geq v) \Rightarrow (\forall x \in l_{out}. x \geq v)$ $(\forall x \in l. x \neq v) \Rightarrow \text{eq}(l, l_{out})$ $(\exists x \in l. x > v) \Rightarrow (\exists x \in l_{out}. x > v)$ $\text{eq}(l, l_{out}) \vee \text{len}(l) \geq \text{len}(l_{out}) + 1$ $\forall x \in l_{out}. x \neq v$ $(\exists x \in l. x < v) \Rightarrow (\exists x \in l_{out}. x < v)$ $(\forall x \in l. x \leq v) \Rightarrow (\forall x \in l_{out}. x \leq v)$ $l_{out} = \text{drop}(l, v)$ $\text{len}(l_{out}) = n + \text{len}(l) \vee n > 1$ $\vee \text{len}(l) = \text{len}(l_{out}) + 1$ $\text{len}(l) = n + \text{len}(l_{out})$ $\text{eq}(l_{out}, l) \vee n = 1 \vee \text{len}(l) > \text{len}(l_{out}) + 1$ $b = \text{elem}(l, v)$ $b \Leftrightarrow (\exists x \in l. x = v)$ $idx = \text{elemIndex}(l, v)$ $idx < \text{len}(l)$ $idx = -1 \vee (\exists x \in l. x = v)$ $idx > -1 \vee (\forall x \in l. x \neq v)$ $(\exists x \in l. x \neq v) \vee 0 = idx + 1 \vee 0 = idx$ $o = \text{min}(l)$ $\exists x \in l. x = o \quad \forall x \in l. x \geq o$ $l_{out} = \text{replicate}(n, v)$ $\text{len}(l_{out}) = n \quad \forall x \in l_{out}. x = v$ $l_{out} = \text{reverse}(l)$ $\text{eq}(l, l_{out}) \vee \text{len}(l) > 1 \quad \text{len}(l) = \text{len}(l_{out})$ $ol_1 = \text{reverse}(l_1)$ $ol_2 = \text{reverse}(l_2)$ $\text{eq}(ol_1, l_2) \Rightarrow \text{eq}(ol_2, l_1)$ $\text{eq}(ol_2, l_1) \Rightarrow \text{eq}(ol_1, l_2)$ $\text{eq}(ol_2, ol_1) \Rightarrow \text{eq}(l_1, l_2)$ $\text{eq}(l_1, l_2) \Rightarrow \text{eq}(ol_1, ol_2)$ $v = \text{ith}(l, idx)$ $0 < idx + 1 \quad \text{len}(l) = idx + 1 \vee \text{len}(l) > 1$ $idx < \text{len}(l) \quad \exists x \in l. x = v$	$ot = \text{emptyTree}()$ $\text{isEmpty}(ot)$ $ot = \text{branch}(v, t_1, t_2)$ $\text{len}(ot) = \text{len}(t_1) + \text{len}(t_2) + 1$ $(\exists x \in t_1. x > v) \Rightarrow (\exists x \in ot. x > v)$ $\exists x \in ot. x = v$ $(\exists x \in t_1. x \neq v) \Rightarrow (\exists x \in ot. x \neq v)$ $(\forall x \in t_1. x \leq v) \wedge (\forall x \in t_2. x \leq v)$ $\Rightarrow (\forall x \in ot. x \leq v)$ $(\exists x \in t_1. x < v) \Rightarrow (\exists x \in ot. x < v)$ $(\exists x \in t_2. x \neq v) \Rightarrow (\exists x \in ot. x \neq v)$ $(\forall x \in t_1. x \geq v) \wedge (\forall x \in t_2. x \geq v)$ $\Rightarrow (\forall x \in ot. x \geq v)$ $(\exists x \in t_2. x < v) \Rightarrow (\exists x \in ot. x < v)$ $(\exists x \in t_2. x > v) \Rightarrow (\exists x \in ot. x > v)$ $b = \text{elem}(t, v)$ $b \Leftrightarrow (\exists x \in t. x = v)$ $ot_1 = \text{branch}(v, t_1, t_2)$ $ot_2 = \text{left}(t)$ $\text{eq}(t_2, t) \Rightarrow \neg \text{eq}(ot_1, ot_2)$ $\text{eq}(ot_1, t) \Rightarrow \text{eq}(t_1, ot_2)$ $\text{eq}(t_1, t) \Rightarrow \neg \text{eq}(ot_1, ot_2)$ $ot_1 = \text{branch}(v, t_1, t_2)$ $ov = \text{rootval}(t)$ $\text{eq}(ot_1, t) \Rightarrow \text{eq}(v, ov)$
Other: integer arithmetic		Binary Search Tree
$o = \text{linSum}(x)$ (Eq. A.15) $o = x \vee -o = 0$ $-x = x \vee 1 \leq o \vee o > x$ $o = \text{linSum}(x)$ (Eq. A.16) $4o = 0 \vee 3x - 3o = 0$ $-3x - o + 4 = 0 \vee -3x + 4o - 1 > 0$ $\vee 4x - o = 0$ $o = \text{nonlinSum}(x)$ (Eq. A.15) $x > 1 \vee o = 1 \vee o = 0$ $-o < -x \vee o = 1 \vee x = 0$ $-o = o \vee x = 1 \vee o > 0$ $o = \text{nonlinSum}(x)$ (Eq. A.16) $2o = 0 \vee -x + 2o - x^2 = 0$ $-4x + 2o - x^2 = 0 \vee 3x - 4o + x^2 \leq 0$ $-x + 2o > 4$ $3x + o > 4 \vee -3o = 0 \vee -x + 2o - 4x^2 = -3$		$ot = \text{emptyBST}()$ $\text{isEmpty}(ot)$ $ot = \text{insert}(t, v)$ $\text{len}(ot) = \text{len}(t) + 1 \vee \text{eq}(t, ot)$ $(\exists x \in ot. x = v)$ $(\forall x \in t. x \geq v) \Rightarrow (\forall x \in ot. x \geq v)$ $(\forall x \in t. x < v) \vee (\exists x \in ot. x > v) \vee \text{eq}(t, ot)$ $(\forall x \in t. x \leq v) \Rightarrow (\forall x \in ot. x \leq v)$ $(\forall x \in t. x > v) \vee (\exists x \in ot. x < v) \vee \text{eq}(t, ot)$ $b = \text{elem}(t, v)$ $b \Leftrightarrow (\exists x \in t. x = v)$ $(\exists x \in t. x = v) \Rightarrow b$
Queue		Stack
$oq = \text{emptyQueue}()$ $\text{isEmpty}(\text{toList}(oq))$ $oq = \text{enqueue}(q, x)$ $\text{eq}(\text{toList}(oq), \text{snoc}(\text{toList}(q), x))$ $(oq, x) = \text{dequeue}(q)$ $\text{eq}(\text{toList}(q), \text{cons}(x, \text{toList}(oq)))$		$os = \text{emptyStack}()$ $\text{isEmpty}(os)$ $os = \text{push}(s, x)$ $\text{len}(os) = \text{len}(s) + 1$ $(os, x) = \text{pop}(s)$ $\text{len}(s) = \text{len}(os) + 1$ $os_1 = \text{push}(s_1, x_1)$ $(os_2, x_2) = \text{pop}(s_2)$ $\text{eq}(os_1, s_2) \Rightarrow x_1 = x_2$ $\text{eq}(os_1, s_2) \Rightarrow \text{eq}(s_1, os_2)$ $\text{eq}(os_2, s_1) \wedge x_1 = x_2 \Rightarrow \text{eq}(os_1, s_2)$

Figure 4.5. $\mathcal{L}$ -properties synthesized by SPYRO. Some properties are rewritten as implications for readability.

semantics defined using SKETCH, but SPYRO[SMT] did not with a semantics defined using SMT formulas. Due to space constraints, we omit `max4` and `arrSearch3`. They are similar to `max3` and `arrSearch2`, respectively, but result in many properties.

SYGUS Benchmarks

The $\mathcal{L}$ -conjunctions synthesized by SPYRO[SMT] and SPYRO[SKETCH] are more precise or equivalent to the original specifications given in the SYGUS problems themselves. In fact, SPYRO found $\mathcal{L}$ -conjunctions that define the exact semantics of the given queries. Inspired by this equivalence, we attempted to use a SYGUS solver (CVC5) on the SYGUS benchmarks to synthesize an exact formula: we used a grammar of conjunctive properties (including the “and” operator, unlike the grammars used by SPYRO), and the specification was the semantics of the function. For 6/8 cases, CVC5 timed out, thus showing that our approach (of synthesizing one $\mathcal{L}$ -property at a time) is beneficial even in the artificial situation in which an oracle supplies the semantics of the best $\mathcal{L}$ -conjunction. Moreover, directly synthesizing an $\mathcal{L}$ -conjunction—as CVC5 attempts—can yield a set of conjuncts of which some are not most-precise $\mathcal{L}$ -properties.

Synquid Benchmarks

To evaluate the synthesized properties, we provided the synthesized $\mathcal{L}$ -conjunctions to SYNQUID and asked it to re-synthesize the reference implementation from which we extracted the properties. In 12/16 cases, SYNQUID could re-synthesize the reference implementation. In 4/16 cases—`elemIndex`, `ith`, `reverse`, and `stutter`—the synthesized properties were not precise enough to re-synthesize the reference implementation. For example, as stated at the end of §4.1, for the `stutter` benchmark our DSL did not contain multiplication by 2 and SPYRO could not synthesize a property stating that the length of the output list is twice the length of the input list. After modifying the DSL to contain multiplication by 2 and the ability to describe when an element appears in both the input and output lists, SPYRO successfully synthesized 7 properties in 154.71 seconds (see Eq. 4.6). From the augmented set of properties, SYNQUID could synthesize the reference implementation of `stutter`. This experiment shows how the

ability to modify the DSL empowers the user of SPYRO with ways to customize the type of properties they are interested in synthesizing.

Other Benchmarks

A core property of Stack is the principle of Last-In First-Out (LIFO). SPYRO was able to synthesize a simple formula that captures LIFO by looking at the relationship between `push` and `pop`: given the query $os_1 = \text{push}(s_1, x_1)$ and $(os_2, x_2) = \text{pop}(s_2)$, SPYRO synthesized the properties $\text{eq}(os_1, s_2) \Rightarrow x_1 = x_2$ and $\text{eq}(os_1, s_2) \Rightarrow \text{eq}(s_1, os_2)$.

A Queue is a data structure whose formal behavior is somewhat hard to describe. Unlike Stack, the behavior of a Queue is not expressible by a simple combination of input and output variables. SPYRO could synthesize formulas describing the behavior of each Queue operation by providing a conversion function from a Queue consisting of two Lists into a List. For the query $(oq, x) = \text{dequeue}(q)$, SPYRO synthesized the property $\text{eq}(\text{toList}(q), \text{cons}(x, \text{toList}(oq)))$.

Finding: SPYRO can synthesize sound best $\mathcal{L}$ -properties and mine specifications for most of the programs in our three categories. Furthermore, SPYRO synthesizes desirable properties that can be easily inspected by a user, who can then modify the DSL $\mathcal{L}$ to obtain other properties if desired.

4.5.2 Application 2: Synthesizing Algebraic Specifications for Modular Synthesis

In many applications of program synthesis, one has to synthesize a function that uses an existing implementation of certain external functions such as data-structure operations—i.e., synthesis is to be carried out in a modular fashion. Even if one has to synthesize a small function implementation, the synthesizer will need to reason about the large amount of code required to represent the external functions, which can hamper performance. Mariano et al. [101] recently proposed a new approach to modular synthesis—i.e., functions are arranged in modules—where instead of providing the synthesizer with an explicit implementation of the external functions, one provides an algebraic specification—i.e., one that does not reveal the internals of the module—of

how the functions in a module operate. For example, to describe the semantics of the functions `emptySet`, `add`, `remove`, `contains` and `size` in a `HashSet` module, one would provide the algebraic properties in Eq. 4.7, which describe appropriate data-structure invariants, such as handling of duplicate elements.

$$\begin{aligned}
& \text{contains}(\text{emptySet}, x) = \perp & x_1 = x_2 \Rightarrow \text{contains}(\text{add}(s, x_1), x_2) = \top \\
& x_1 \neq x_2 \Rightarrow \text{contains}(\text{add}(s, x_1), x_2) = \text{contains}(s, x_2) \\
& \text{remove}(\text{emptySet}, x) = \text{emptySet} & x_1 = x_2 \Rightarrow \text{remove}(\text{add}(s, x_1), x_2) = \text{remove}(s, x_2) \\
& x_1 \neq x_2 \Rightarrow \text{remove}(\text{add}(s, x_1), x_2) = \text{add}(\text{remove}(s, x_2), x_1)
\end{aligned} \tag{4.7}$$

While their approach has shown promise in terms of scalability, to use this idea in practice one has to provide the algebraic specifications to the synthesizer *manually*, a tricky task because these specifications typically define how multiple functions interact with each other.

In our case study, we used SPYRO to synthesize algebraic specifications for benchmarks used in the evaluation of JLIBSKETCH, an extension of the SKETCH tool that supports algebraic specifications [101]. We considered the 3 modules—`ArrayList`, `HashSet`, and `HashMap`—that provided algebraic specifications, did not use `string` operations (our current implementation does not support strings), and did not require auxiliary functions that were not present in the implementation to describe the algebraic properties. For each module, JLIBSKETCH contained both the algebraic specification of the module and its mock implementation—i.e., a simplified implementation that mimics the intended library’s behavior (e.g., `HashSet` is implemented using an array). Given the mock implementation of the module, we asked SPYRO to synthesize most-precise algebraic specifications.

For this case study, designing a grammar that accepted all possible algebraic specifications but avoided search-space explosion proved to be challenging. Instead, we opted to create multiple grammars for each module to target different parts of the algebraic specifications, and called SPYRO separately for each grammar. For example, if the JLIBSKETCH benchmark contained an

algebraic specification $\text{size}(\text{add}(s, x)) = \text{size}(s) + 1$, we considered the grammar to contain properties of the form $\text{guard} \Rightarrow \text{size}(\text{add}(s, x)) = \text{exp}$. All the DSLs designed for algebraic specifications synthesis were reused by modifying what function symbols could appear in the DSL. The detailed grammars are presented in Appendix A.3.

SPYRO terminated with a best $\mathcal{L}$ -conjunction for all the benchmarks (and grammars) in less than 800 seconds per benchmark. SPYRO was slower than the enumerative baseline presented in Section 4.5.1 for very small languages ($|\mathcal{L}| < 5$) but faster in all other cases. The speedups were not as prominent as for Application 1.

For all but one benchmark, the $\mathcal{L}$ -conjunctions synthesized by SPYRO were equivalent to the algebraic properties manually designed by the authors of JLIBSKETCH. For the implementation of HashMap provided in JLIBSKETCH, for one specific grammar, SPYRO synthesized an empty $\mathcal{L}$ -conjunction (i.e., the predicate *true*) instead of the algebraic specification provided by the authors of JLIBSKETCH—i.e., $k_1 = k_2 \Rightarrow \text{get}(\text{put}(m, k_1, v), k_2) = v$. Upon further inspection, we discovered that the implementation of HashMap used in JLIBSKETCH was incorrect and did not satisfy the specification the authors provided, due to an incorrect handling of hash collision! After fixing the bug in the implementation of HashMap, we were able to synthesize the algebraic specification. Because algebraic properties often involve multiple functions, we were not able to separately verify their correctness on all inputs using the Dafny verifier, but the fact that we obtained the same properties that the authors of JLIBSKETCH specified in their benchmarks is a strong signal that our properties are indeed sound.

Finding: SPYRO can help automate modular synthesis by synthesizing precise algebraic specifications that existing synthesis tools can use to speed up modular synthesis. Thanks to SPYRO’s provable guarantees, we were able to uncover a bug in one of JLIBSKETCH’s module implementations.

4.5.3 Application 3: Automating Sensitivity Analysis

Automatically reasoning about quantitative properties such as differential privacy from D’Antoni et al. [33] in programs requires one to analyze how changes to a program input affect the program output—e.g., differential privacy typically requires that bounded changes to a function’s input cause bounded changes to its output. A common and inexpensive approach to tackle this kind of problem is to use a compositional *sensitivity analysis* (either in the form of an abstract interpretation or of a type system from [33]) in which one tracks how sensitive each operation in a program is to changes in its input. For example, one can say that the function $f(x) = \text{abs}(2x)$, when given two inputs x_1 and x_2 that differ by k , produces two outputs that differ by at most $2k$.

While for the previous function, it was pretty easy to identify a precise sensitivity property, it is generally tricky to do so for functions involving data structures, such as lists, which are of interest in differential privacy when the list represents a database of individuals [156]. In this case study, we considered 9 list-manipulating functions (`append`, `cons`, `deleteFirst`, `delete`, `reverse`, `snoc`, `stutter`, `tail`, and `cons_delete`) and used SPYRO[SKETCH] to synthesize precise sensitivity properties describing how changes to the input lists affect the outputs. The `cons_delete` benchmark uses the query `deleteFirst(cons(x,l),x)` involving the composition of two functions.

For each function f , we used SPYRO to synthesize a property of the form

$$\text{guard}(x_1, x_2) \wedge \text{dist}(l_1, l_2) \leq d \Rightarrow \text{dist}(f(l_1, x_1), f(l_2, x_2)) \leq \text{exp}$$

where *guard* can be the predicate `true` or an equality/inequality between x_1 and x_2 (the grammars vary across benchmarks), *dist* is the function computing the distance between two lists (we run experiments using both edit and Hamming distance), and the expression *exp* (the part to synthesize) can be any linear combinations of `len(x)`, `len(y)`, d , and constants in the range

-1 to 2. When considering all combinations of functions, guards, and distances, we obtained 18 benchmarks. All the DSLs designed for algebraic specifications synthesis were reused by modifying what function symbols could appear in the DSL. The complete grammars are shown in Appendix A.4.

SPYRO terminated with a best $\mathcal{L}$ -conjunction for all the benchmarks (and grammars) in less than 250 seconds per benchmark. SPYRO outperformed the enumerative baseline presented in Section 4.5.1 for every problem ($3.14\times$ speedup for Hamming-distance sensitivity problems and $11.93\times$ speedup for edit-distance sensitivity problems—geometric mean).

We observed that even for simple functions, sensitivity properties are fairly complicated and hard to reason about manually. For example, SPYRO synthesizes the following sensitivity $\mathcal{L}$ -property for the function `deleteFirst` (D_e denotes the edit distance):

$$D_e(l_1, l_2) \leq d \Rightarrow D_e(\text{deleteFirst}(l_1, x_1), \text{deleteFirst}(l_2, x_2)) \leq d + 2$$

However, if we add a condition that the element removed from the two lists is the same, SPYRO can synthesize the following $\mathcal{L}$ -property that further bounds the edit distance on the output:

$$x_1 = x_2 \wedge D_e(l_1, l_2) \leq d \Rightarrow D_e(\text{deleteFirst}(l_1, x_1), \text{deleteFirst}(l_2, x_2)) \leq d + 1$$

When inspecting this property, we were initially confused because we had thought that the edit distance should not increase at all if identical elements are removed. However, that is false as illustrated by the following tricky counterexample $l_1 = [1; 2; 3]$, $l_2 = [3; 2; 3]$ and $x_1 = x_2 = 3$. Besides the $\mathcal{L}$ -property shown above with bound $d + 1$, SPYRO also synthesized (incomparable) best $\mathcal{L}$ -properties with bounds $\text{len}(l_1) - \text{len}(l_2) + 2d$ and $\text{len}(l_2) - \text{len}(l_1) + 2d$ for the same query. All combined, these $\mathcal{L}$ -properties imply that the edit distance should not increase when $d = 0$.

Because of the complexity added by the programs that compute the edit and Hamming

distances, by the use of unbounded data structures, and by the fact that sensitivity properties are hyperproperties, we were not able to separately verify the soundness of synthesized $\mathcal{L}$ -conjunctions on all inputs using the Dafny verifier. However, we believe that the synthesized properties are indeed sound given that they hold for lists up to length 7—i.e., the bound imposed by SKETCH.

Finding: SPYRO can synthesize precise sensitivity properties for functions involving lists; the synthesized function would be challenging for a human to handcraft.

4.5.4 Application 4: Enabling New Abstract Domains

One of the most powerful relational abstract domains is the domain of *convex polyhedra* [32, 10]. While programs typically operate over `int`-valued program variables for which arithmetic is performed modulo a power of 2, such as 2^{16} or 2^{32} , existing implementations of polyhedra are based on conjunctions of linear inequalities with rational coefficients over rational-valued variables. This disconnect prevents polyhedra from precisely modeling how values wrap around in `int`/bit-vector arithmetic when arithmetic operations overflow.

Heretofore, it has not been known how to create an analog of polyhedra that is appropriate for bit-vector arithmetic. Yao et al. [161] recently defined two domains (Version 1 and Version 2 below), but have only devised algorithms to support Version 2.

Version 1 (bit-vector-polyhedra domain): conjunctions of linear bit-vector inequalities

Version 2 (integral-polyhedra domain): conjunctions of linear integer inequalities

The case study described in this section shows that SPYRO provides a way to enable precise polyhedra operations for Version 1. The main reason why operations for bit-vector polyhedra have not been proposed previously is that it is challenging to work with relations over bit-vector-valued variables. For example, let x and y be 4-bit bit-vectors. Fig. 4.6(a) depicts the satisfying assignments of the inequality $x + y + 4 \leq 7$ interpreted over 4-bit unsigned modular

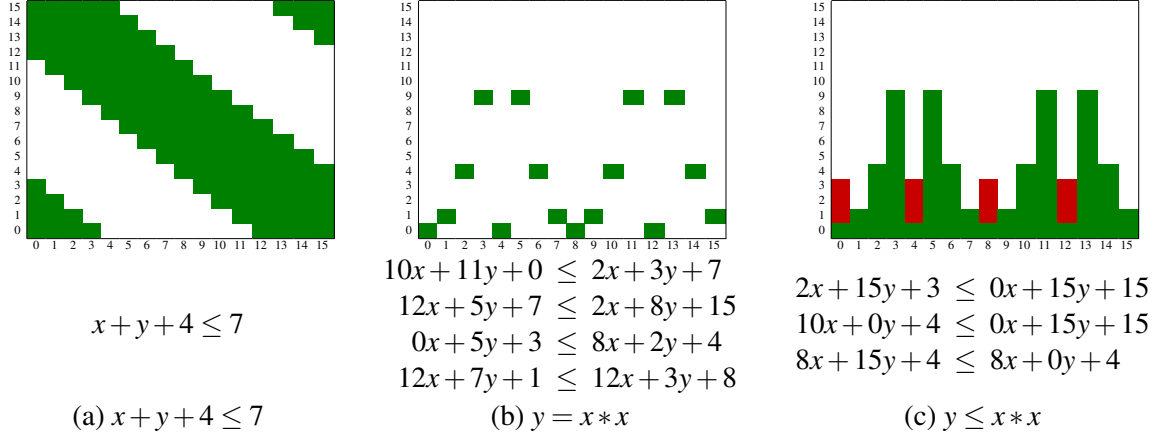


Figure 4.6. Each subfigure illustrates a bit-vector formula (in green) and the most precise bit-vector polyhedron computed by SPYRO (i.e., the inequalities above the formulas). Each colored cell in the plots represents a solution in 4-bit unsigned modular arithmetic of the conjunction of the inequalities found by SPYRO: green cells represent solutions to the original formula, whereas red cells are points that are solution to the inequalities, but do not satisfy the original formula. In (a) and (b), the conjunctive formula represents the original formula exactly (there are only green cells). In (c), the twelve occurrences of red cells are points that do not satisfy the original formula, but are needed for a conjunctive formula to over-approximate the original formula.

arithmetic. As seen in the plot, the set of points that satisfy a single bit-vector inequality can be a non-contiguous region.

In general, a bit-vector inequality over unsigned bit-vector variables $X = \{x_1, \dots, x_n\}$ has the form $\sum_{i=1}^n p_i x_i + q \leq \sum_{j=1}^n r_j x_j + s$, where $\{p_i\} \cup \{q\} \cup \{r_j\} \cup \{s\}$ are unsigned bit-vector constants, and $\leq$ is unsigned comparison. Conjunctions of inequalities are a fragment of quantifier-free bit-vector logic ($\mathcal{L}_{BV}$). Let $\llbracket \varphi \rrbracket_{BV}$ denote the set of assignments to X that satisfy formula $\varphi \in \mathcal{L}_{BV}$.

We instantiated SPYRO to take as input a formula $\varphi \in \mathcal{L}_{BV}$ and return a conjunction ψ of bit-vector inequalities—i.e., the *symbolic abstraction* [128, §5] of φ in the conjunctive fragment $\mathcal{L}_{BV(\wedge)}$. Because SPYRO computes best $\mathcal{L}$ -properties, in this setting it computes the most-precise symbolic abstraction—i.e., the formula $\hat{\alpha}$ computed by SPYRO is one representation of the *most-precise abstraction* of φ that is expressible as a conjunction of bit-vector inequalities.

As known from the literature ([129, 150, 151] and [128, §5]), operations needed for abstract interpretation, such as (i) the creation and/or application of abstract transformers, and (ii)

taking the join of two abstract-domain elements, can be performed via an algorithm for symbolic abstraction. For instance, if ψ_a and ψ_b are two formulas in $\mathcal{L}_{BV(\wedge)}$, we can perform the join $\psi_a \sqcup \psi_b$ by $\widehat{\alpha}(\psi_a \vee \psi_b)$. (Note that $\psi_a \vee \psi_b$ is not a formula of $\mathcal{L}_{BV}$.) For this reason, we say that SPYRO *enables* this new abstract domain.

In our experiments, we limited inequalities to two variables x and y on each side, and used 4-bit unsigned arithmetic. Our benchmarks were taken from an earlier study conducted by one of the authors, which on each example used brute force to consider all 16,762,320 non-tautologies of the 16,777,216 4-bit inequalities of the form $ax + by + c \leq dx + ey + f$. That study found that some example formulas had hundreds of thousands of inequalities as consequences. We selected 9 interesting-looking formulas to use as benchmarks, including linear/nonlinear operations, equalities and inequalities, Boolean combinations, and one pair of formulas on which to perform the join operation. (See §4.5.4 and Appendix A.5.)

Quantitative Analysis

SPYRO[SKETCH] computed a sound best $\mathcal{L}$ -conjunction for 9/9 formulas, and guaranteed that 8/9 were best $\mathcal{L}$ -conjunctions. For the query $y = x/2$, SPYRO[SKETCH] timed out on a call to CHECKPRECISION, but the obtained $\mathcal{L}$ -conjunction was indeed a best $\mathcal{L}$ -conjunction because it defined the exact semantics of the query. SPYRO computed an $\mathcal{L}$ -conjunction for all the 9 benchmarks in less than 400 seconds per benchmark, which is 2-5 orders of magnitude faster than the enumerative baseline presented in Section 4.5.1. Each output $\mathcal{L}$ -conjunction contained between 1 and 6 $\mathcal{L}$ -properties. For this domain, SKETCH—and hence SPYRO[SKETCH]—is sound and precise because we are working with bit-vector arithmetic of fixed bit-width.

SPYRO[SKETCH] could not terminate for most of our benchmarks when considering 8-bit arithmetic. Because our examples contain several multiplications, this limitation is not surprising because multiplication is one of the known weaknesses of SKETCH and its underlying SAT solver. Recently, there have been promising advances in SAT solving for multiplication circuits [82] that, if integrated with SKETCH, we believe would help SPYRO scale to larger bit-vectors.

Qualitative Analysis

Examples of results obtained by SPYRO are shown in Fig. 4.6. The result that the most-precise abstraction of these formulas could be expressed using only a small number of inequalities was surprising to the authors. In Fig. 4.6(c), the formula we are abstracting is $\varphi =_{df} y \leq x * x$. Fig. 4.6(c) shows that, in addition to the green points that satisfy the non-linear inequality $y \leq x * x$, the bit-vector-polyhedral abstraction found for φ includes twelve “extra” points, indicated by the red cells. Because SPYRO finds a *most-precise* sound bit-vector-polyhedral abstraction of φ , every sound bit-vector-polyhedral abstraction of φ must also include those twelve points. In an earlier study conducted by one of the authors, they used brute force to consider all 16,762,320 non-tautologies of the 16,777,216 4-bit inequalities of the form $ax + by + c \leq dx + ey + f$. That study found that the following numbers of inequalities over-approximated the original formula: 564 for Fig. 4.6(a), 109,008 for Fig. 4.6(b), and 456 for Fig. 4.6(c). Thus, SPYRO showed that a most-precise abstraction could be 2-4 orders of magnitude smaller than one obtained by brute force.

Finding: SPYRO can synthesize the most-precise sound bit-vector-polyhedral abstraction of a given bit-vector formula φ over 4-bit arithmetic. Furthermore, SPYRO surprised the authors by showing that the most-precise sound bit-vector-polyhedral abstraction for the presented examples could be precisely expressed with only a handful of bit-vector inequalities.

4.5.5 Further Analysis of SPYRO’s Performance

In the previous sections, we have shown that SPYRO can synthesize best $\mathcal{L}$ -conjunctions for a variety of case studies. In this section, we analyze what parameters affect SPYRO’s running time.

Q1: How do Different Primitives of the Algorithm Contribute to the Running Time?

On average, SPYRO spends 13.69 % of the time performing SYNTHESIZE, 26.78 % performing CHECKSOUNDNESS, and 42.33 % performing CHECKPRECISION (details in Table

2 in App. C of Park et al. [116]).

It usually takes longer for `CHECKSOUNDNESS` and `CHECKPRECISION` to show the nonexistence of an example—i.e., to return $\perp$ —than to find an example. `CHECKSOUNDNESS` is one of the simplest queries, but occupies a large portion of the running time because it is expected to return $\perp$ many times, whereas `CHECKPRECISION` needs to return $\perp$ only once for each call to `SYNTHPROPERTY`. The last call to `CHECKPRECISION` (i.e., the one that returns $\perp$) often takes a significant amount of time to complete (on average 19.61 % of the time spent on each run of `SYNTHPROPERTY`).

Finding: SPYRO spends most of the time checking soundness and precision.

Q2: What Parts of the Input Affect the Running Time?

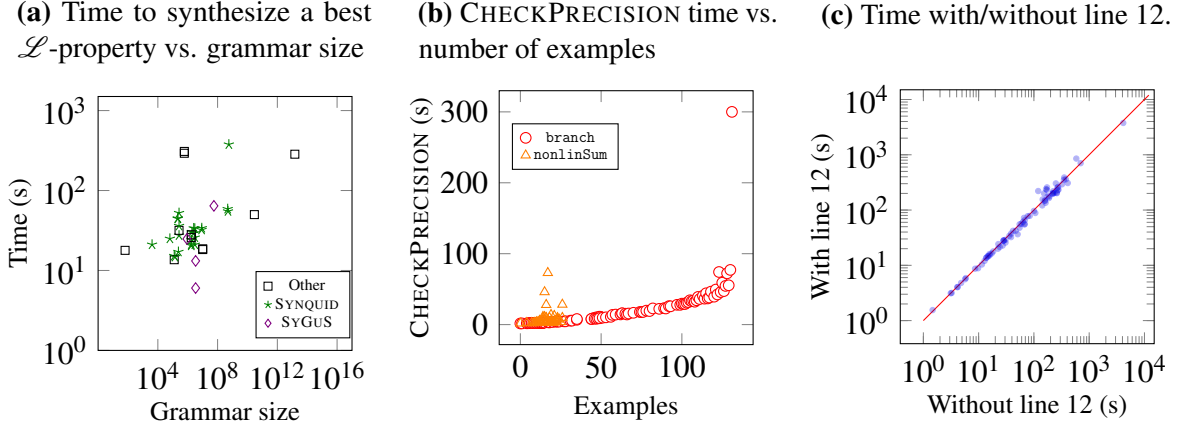
The number of $\mathcal{L}$ -properties in the language $\mathcal{L}$ has a large impact on the time taken by `SYNTHESIZE` (Fig. 4.7a) and `CHECKPRECISION`.

The complexity of the code defining the semantics of various operators has a large impact on how long `CHECKSOUNDNESS` takes. `insert`, `delete` of BST and edit distance have relatively complicated implementations, and `CHECKSOUNDNESS` takes longer for these problems.

The size and complexity of the example space also affect the running time. The biggest factor contributing to the size of the example space is the number of input and output variables used. The number of possible examples, i.e., variable assignments, increases exponentially with the number of variables. The size of the example space affects not only the number of total queries but also the time that each query takes. Specifically, the number of positive and negative examples affects the time taken by `SYNTHESIZE` or `CHECKPRECISION` (notice that `CHECKSOUNDNESS` does not take the examples as input), as shown in Figure 4.7b.

Finding: The running time of SPYRO is affected by the sizes of (i) the property search space, (ii) the programs that describe the semantics of the operators, and (iii) the example search space.

Figure 4.7. Evaluation of the running time of SPYRO for different input sizes and optimizations.



Q3: How effective is line 12 in Algorithm 8?

We compared the running times of SPYRO with and without line 12 (Figure 4.7c). When line 12 is present, SPYRO is 3.06% faster (geometric mean) than when line 12 is absent, but both versions can solve the same problems. The optimization is only effective when the language has many incomparable properties that do not imply each other and cause SYNTHESIZE to often return $\perp$, thus triggering Line 10 in Algorithm 8—e.g., in all SYGUS benchmarks the language $\mathcal{L}$ is such that the optimization is not used.

Finding: Freezing negative examples is slightly effective.

4.6 Related Work

Abstract-interpretation techniques

Many static program-analysis techniques are pitched as tools for checking safety properties, but behind the scenes they construct an artifact that abstracts the behavior of a program (in the sense of abstract interpretation [30]). One such kind of artifact is a procedure summary [32, 31, 136, 61], which abstracts a procedure’s transition relation with an abstract value from an abstract domain, the elements of which denote transition relations.

Our problem is an instance of the *strongest-consequence problem* [128]. Existing tech-

niques for solving this problem rely on properties of the language $\mathcal{L}$ that are typical of abstract interpretation. Some techniques work from “below,” identifying a chain of successively weaker implicants, until one is a consequence of φ [129]. Other techniques work from “above,” identifying a chain of successively stronger implicates, until no further strengthening is possible [150, 151]. Ozeri et al. [111] explored a different approach, which works from above by repeatedly applying a semantic-reduction operation [30]. (A semantic reduction operation finds a less-complicated description of a given set of states if one exists.) Our work differs from methods that use abstract interpretation in several aspects. First, our algorithm is the first to use both positive and negative examples to achieve precision. Second, while our work supports a variety of DSLs specified via a grammar, existing methods require that certain operations can be performed on concrete states and elements of the language $\mathcal{L}$ (e.g., joins [150]), thus limiting the language that can serve as the DSL.

Type inference

Liquid type inference [133, 67, 153] can infer a weakest precondition from a given postcondition or a strongest postcondition from a given precondition. To make the problem tractable, properties must be specified in a user-given restricted set of predicates that are closed under Boolean operations. Although our work shares some similarities—e.g., looking for properties over a restricted DSL—we tackled a fundamentally different problem because no pre- or post-condition is given as an input, and our algorithm instead looks for best $\mathcal{L}$ -properties. Furthermore, our work is not restricted to functional languages.

Invariant inference

Several data-driven, CEGIS-style algorithms can synthesize program invariants. These techniques look for any invariant that is satisfactory for a client verification problem, whereas we do not assume there is a client for whom the properties are synthesized. Without a client, “true” is a sound (and also weakest) but useless specification. The absence of a client requires synthesized specifications to be precise, therefore requiring our new `CHECKPRECISION` primitive.

A closely related system is Elrond [163], which synthesizes weakest library specifications that make verification possible in a client program. Elrond allows one to specify a set of target predicates of interest, and finds quantified Boolean formulas with equalities over the variables and the predicates. Because soundness (with respect to the library function) is only checked on a set of inputs, Elrond tries to synthesize weakest specifications using an iterative weakening approach. Their algorithm takes advantage of the structure of supported formulas (they can contain disjunctions), but has some limitations (they can only contain equalities).

The key differences between Elrond and our work are: (i) Our work supports a user-supplied DSL, which enables more generality, but prevents the use of techniques that rely on access to arbitrary Boolean operations. (ii) Our work uses a parametric DSL that can contain complex user-given functions, whereas Elrond only allows parametric atoms (i.e., user-defined Boolean function), equality over variables, and Boolean combinations of them. Our tool SPYRO can synthesize the $\mathcal{L}$ -property “ $2o = 0 \vee -x + 2o - x^2 = 0$,” whereas Elrond does not consider arithmetic predicates. (iii) The specifications generated by SPYRO are most precise with respect to the DSL $\mathcal{L}$, allowing for their reuse in multiple problem instances that use the same function. Such reuse is possible because SPYRO ensures soundness of the synthesized properties. In contrast, Elrond operates in a closed-box setting and uses a random sampler for soundness, intentionally weakening the synthesized specifications to enhance the likelihood of soundness. (iv) Our work can synthesize arithmetic properties efficiently (e.g., the ones considered in Sections 4.5.3 and 4.5.4) as well as complex algebraic properties (e.g., the ones in Section 4.5.2). In theory, Elrond can describe all the necessary components to express algebraic properties such as the property $(s, x) = \text{pop}(\text{push}(s, x))$, but one would need to provide a function that combines pop and push. This approach is feasible if one is interested in only a few possible ways of combining functions, but becomes infeasible once more combinations are possible (e.g., for an arithmetic formula). In short, the two approaches have different goals.

The presence of a user-supplied DSL and absence of a client distinguish our work from other prior work, e.g., abductive inference [41], ICE-learning [52], LoopInvGen [112],

Hanoi [106], and Data-Driven CHC Solving [164].

Dynamic techniques

Daikon [46, 47] is a system for identifying likely invariants by inspecting program traces. Invariants generally involve at most two program quantities and are checked at procedure entry and exit points (i.e., invariants form precondition/postcondition pairs). In Daikon, the default is to check 75 different forms of invariants, instantiated for the program variables of interest. The language of invariants can be extended by the user.

SPYRO differs from Daikon (and follow-up work, e.g., [13]) in two ways: (i) The language $\mathcal{L}$ is not limited to a set of predicates, and SPYRO scales to languages containing millions of properties; (ii) the properties that SPYRO synthesizes are sound and provably *best* $\mathcal{L}$ -properties. Furthermore, while Daikon’s dynamic approach can scale to large programs, we could not find a way to encode our case studies as instances that Daikon could receive as input.

A similar tool to Daikon is QuickSpec [143], which generates equational properties of Haskell programs from random tests. SPYRO differs from QuickSpec in two ways: (i) The language $\mathcal{L}$ is not limited to equational properties; (ii) the properties that SPYRO synthesizes are sound and provably *best* $\mathcal{L}$ -properties.

Astorga et al. [9] synthesize contracts that are sound with respect to positive examples generated by a test generator. We see two main differences between that work and ours: (i) They do not use negative examples, whereas we do. Negative examples are the key to synthesizing *best* $\mathcal{L}$ -properties. (ii) Their work does not allow a parametrized DSL and their notion of “tight” is with respect to a syntactic restriction on the logic in which the contract is to be specified.

Synthesis of best $\mathcal{L}$ -transformers

The paper that inspired our work synthesizes most-precise abstract transformers in a user-given DSL [81]. We realized that their basic insight—use both positive and negative examples; treat positive examples as hard constraints and negative examples as “maybe” constraints—had broader applicability than just creating abstract transformers.

Our work differs in two key ways. First, because our goal is to obtain a formula rather than a piece of code (their setting), we could take advantage of the structure of formulas—in particular, conjunctions—to decompose the problem into (i) an “inner search” to find a best $\mathcal{L}$ -property that is an individual conjunct (Alg. 8), and (ii) an “outer search” to accumulate best $\mathcal{L}$ -properties to form a best $\mathcal{L}$ -conjunction (Alg. 9). Second, Alg. 8 exploits monotonicity—i.e., once a sound $\mathcal{L}$ -property is found, there must exist a best $\mathcal{L}$ -property that implies it (Lemma 4.2.2). This observation allows us to use a simplified set of primitives: our algorithm uses a `SYNTHESIZE` primitive, whereas theirs requires a `MAXSYNTHESIZE` primitive—i.e., one that synthesizes a program that accept all the positive examples and rejects as many negative examples as possible. Our ideas could be back-ported to provide improvements in their setting as well: if the abstract domain supports `meet` ($\sqcap$), they could run their algorithm multiple times to create a kind of “conjunctive” transformer, which would run multiple, incomparable best $\mathcal{L}$ -transformers, and then take the `meet` of the results.

4.7 Conclusion

This chapter presents a formal framework for the problem of synthesizing a best $\mathcal{L}$ -conjunction—i.e., a conjunctive specification of a program with respect to a user-defined logic $\mathcal{L}$ —and an algorithm for automatically synthesizing a best $\mathcal{L}$ -conjunction. The innovations in the algorithm are three-fold: (i) it identifies individual conjuncts that are themselves strongest consequences; (ii) it balances negative examples that *must* be rejected by the $\mathcal{L}$ -property being synthesized and ones that *may* be rejected; and (iii) it guarantees progress via monotonic constraint hardening.

Our work opens up many avenues for further study. One is to harness other kinds of synthesis engines to implement `SYNTHESIZE`, `CHECKSOUNDNESS`, and `CHECKPRECISION`. Recent work on Semantics-Guided Synthesis (SEMGUS) [84] provides an expressive synthesis framework for expressing complex synthesis problems like the ones discussed in this chapter.

SEMGUS solvers, such as MESSY [83], are able to produce two-sided answers to a problem: either synthesizing a solution, or proving that the problem is unrealizable—i.e., has no solution—exactly what is needed for CHECKPRECISION. On the theoretical side, while we have used first-order logic, it would be interesting to try other logics, such as separation logic [131] or effectively propositional logic [79, 113]. On the practical side, our work could find applications in invariant generation [114] and code deobfuscation [19].

Acknowledgments

Chapter 4, in full, is based on the paper from Kanghee Park, Loris D’Antoni, and Thomas Reps “Synthesizing Specifications” published in Proceedings of the ACM on Programming Languages 7(OOPSLA2), pp. 1787-1816. 2023. The dissertation author is the primary investigator and author of this paper.

Chapter 5

Specifications for Nondeterministic Programs

Writing specifications is especially hard for programs involving nondeterminism, a construct necessary to model many practical applications such as concurrency, random execution, and games. Part of what makes writing such specifications hard is that specifications for programs involving nondeterminism might capture two types of properties: *demonic properties*, which hold for *every* nondeterministic execution, and *angelic properties*, which hold for *some* nondeterministic execution.

Consider a program that nondeterministically shuffles the elements of a list. A possible valid demonic property could state that the output list is always a permutation of the input list. However, de Vries and Koutavas [35] argued that this specification alone is “incomplete”; a sorting function would also meet this specification, even though it would not behave correctly as a shuffle function. A better specification would include an angelic property that states that the shuffle function can in fact produce all permutations of the input list—i.e., for any permutation of the input list, there exists *some* nondeterministic execution that can generate it.

The goal of this chapter is to devise a unified logical framework for synthesizing provably sound angelic and demonic specifications for nondeterministic programs. Most approaches that automatically generate specifications from code [46, 47, 143, 9] rely on a finite set of dynamically generated input executions and cannot guarantee soundness even when the programs

are deterministic. To our knowledge, SPYRO [116] is the only framework for synthesizing specifications that are provably sound, but it is fundamentally limited to deterministic programs. In this chapter, we redesign the SPYRO framework to support nondeterminism, and both angelic and demonic reasoning.

Limitations of SPYRO

In SPYRO, the problem of synthesizing a specification is phrased as follows: Given (i) a *quantifier-free* query Ψ posed about a set of function definitions, and (ii) a domain-specific language $\mathcal{L}$ in which each extracted property is to be expressed (we call properties in the language $\mathcal{L}$ -properties), the goal is to synthesize a conjunction $\bigwedge_i \phi_i$ of logically incomparable $\mathcal{L}$ -properties such that each ϕ_i is an *over-approximation* of Ψ and is a strongest in $\mathcal{L}$ —i.e., there is no other $\mathcal{L}$ -property that over-approximates Ψ and that strictly implies ϕ_i . For example, for a query $\Psi := (l_{out} = \text{reverse}(l))$ describing a list-reverse function, and a language $\mathcal{L}$ of arithmetic formulas over variables and their lengths, the property $\phi := \text{len}(l_{out}) \leq \text{len}(l)$ is an $\mathcal{L}$ -consequence of Ψ , but not a strongest one, because $\phi_1 := \text{len}(l_{out}) = \text{len}(l)$ is a stronger $\mathcal{L}$ -consequence of Ψ .

SPYRO is very expressive, but it does not support nondeterministic programs for two reasons. First, SPYRO’s synthesis algorithms are fundamentally limited to *quantifier-free queries*. Without existential quantifiers, SPYRO’s queries cannot model nondeterministic programs, concurrent programs, or uncertainty. Second, SPYRO is *limited to synthesizing over-approximations* of the program behavior. While over-approximations can capture what must happen for *every* (nondeterministic) execution (the demonic properties), reasoning about the actual behaviors that *some* (nondeterministic) execution can exhibit (the angelic properties) requires under-approximated specifications.

The LOUD Framework

This chapter addresses the two limitations of SPYRO and presents LOUD, a general framework for solving the following problem:

Given an **existentially quantified query** Ψ posed about a set of function definitions and a language $\mathcal{L}$, find (i) a **strongest conjunction** of $\mathcal{L}$ -consequences that is implied by Ψ , and (ii) a **weakest disjunction** of $\mathcal{L}$ -implicants that implies Ψ .

The LOUD framework is our key contribution. While LOUD superficially looks like a small modification of SPYRO, the minimal extension of including existential quantifications in queries allows LOUD to elegantly capture many new complex scenarios into a single unified logical framework. Specifically, existentially quantified queries allow LOUD to reason about both angelic and demonic properties of programs involving nondeterminism. For example, consider the dining-philosophers problem where n philosophers are arranged in a circle, each concurrently (and nondeterministically) acquiring and releasing contended resources placed on either of their sides. We can model what combinations of actions and scheduling lead to a deadlock using an existentially quantified query such as $\exists s. dl = \text{philo}(s, p_1, \dots, p_n)$, where $p_i \in \{L, R\}$ indicates which resource the philosopher p_i tries to take first and dl denotes that a deadlock has happened; s is the nondeterministic sequence of order in which threads are scheduled (detailed description in Section 5.7.3).

Supporting reasoning of both over- and under-approximate specifications (i.e., computing both $\mathcal{L}$ -consequences and $\mathcal{L}$ -implicants) enables new applications. Let's say we are interested in understanding what philosophers' actions *may* lead to a deadlock for *some* possible schedule. When given an appropriate language $\mathcal{L}$, a possible under-approximation (i.e., $\mathcal{L}$ -implicant) of the query $\exists s. dl = \text{philo}(s, p_1, \dots, p_n)$ is $dl \wedge p_1 = \dots = p_n$, which states that deadlock can happen when all the philosophers prefer the same direction. On the other hand, if we instead are interested in what philosophers' choices *must* prevent deadlocks for *any* possible schedule, we can resort to over-approximation. A possible over-approximation (i.e. $\mathcal{L}$ -consequence) in the LOUD framework is $p_1 \neq p_2 \Rightarrow \neg dl$, which states that deadlock will not happen when processes p_1 and p_2 disagree on their fork choice. The above two example properties show that, for nondeterministic programs, consequences hold for *every possible* nondeterministic choice

(the demonic perspective), whereas implicants hold for *at least one* nondeterministic choice (the angelic perspective).

Thanks to its generality, LOUD can also capture reasoning capabilities of Hoare logic [70] (e.g., computing weakest liberal precondition and strongest postcondition) and incorrectness logic [110, 35] (e.g., computing weakest possible precondition and weakest under-approximate postcondition).

New Synthesis Algorithms in LOUD

Existentially quantified queries and the ability to synthesize both over- and under-approximations make the LOUD framework more expressive than SPYRO, but also make synthesis more challenging, thus requiring new algorithmic insights.

Previous chapter presented a counterexample-guided synthesis (CEGIS) algorithm for solving problems in the SPYRO framework. The algorithm accumulates positive and negative examples of possible program behaviors with respect to the given query and synthesizes $\mathcal{L}$ -properties consistent with them. A primitive called CHECKSOUNDNESS checks if a candidate property is indeed sound and, if not, it produces a new positive example that the property fails to accept. To ensure $\mathcal{L}$ -consequences are strongest, a primitive CHECKPRECISION checks if the current $\mathcal{L}$ -consequence is strongest; if it is not, CHECKPRECISION returns a new $\mathcal{L}$ -property that accepts all positive examples, rejects all negative examples, and rejects one more negative example (which is also returned). By alternating calls to these primitives, the algorithm eventually finds a strongest $\mathcal{L}$ -consequence.

Key contributions and innovations in how we algorithmically solve LOUD problems include (i) generalized CHECKSOUNDNESS and CHECKPRECISION primitives so that each operation has a dual form that can be used to synthesize both $\mathcal{L}$ -consequences and $\mathcal{L}$ -implicants, and (ii) how we implement these primitives in the presence of existential quantifiers. Specifically, proving that an $\mathcal{L}$ -consequence is strongest and proving that an $\mathcal{L}$ -implicant is sound require solving a constraint with quantifier alternations of the form $\exists e. \forall h. \neg \psi(e, h) \wedge \varphi(e)$. To perform

this check, we integrate a counterexample-guided quantifier instantiation algorithm (CEGQI) that operates in tandem with the overall CEGIS algorithm. The CEGIS algorithm accumulates examples that approximate the behavior of the query, while the CEGQI algorithm accumulates instances of the quantified variable h that show if an example is positive or negative. To our knowledge, our algorithm is the first one to combine CEGIS and CEGQI to deal with multiple nested quantifiers.

We implement a tool, called ASPIRE, to solve the synthesis problems in the LOUD framework. ASPIRE can describe and identify sources of bugs in both deterministic and non-deterministic programs, extract properties from concurrent programs, and synthesize winning strategies in two-player games. Because ASPIRE is built on the top of the SKETCH program synthesizer [145], it is only sound for programs in which inputs, recursion, and loops are bounded. In the future, this limitation can be lifted by considering more general (though less efficient) program synthesizers [83].

Contributions

Our work makes the following contributions:

- A unified logical framework, LOUD, for the problem of synthesizing both strongest $\mathcal{L}$ -consequences and weakest $\mathcal{L}$ -implicants for existentially quantified queries (§5.2).
- Algorithms for solving LOUD problems using four simple well-defined primitives: SYNTHESIZE, CHECKIMPLICATION, CHECKSTRONGEST and CHECKWEAKEST (§5.4).
- An algorithm that combines CEGQI and CEGIS to efficiently implement the primitives CHECKIMPLICATION and CHECKSTRONGEST for existentially quantified queries (§5.5).
- A tool that implements our framework, called ASPIRE (§5.6).
- Multiple instantiations of LOUD, showing its capability across a wide range of applications, e.g., reasoning about nondeterministic/concurrent programs and synthesizing game strategies (§5.7).

<pre> Variables { int a, M, y; exist int x; } Query { y = modhash(a, M, x); } </pre> (a) Query Ψ	<pre> Language { D $\rightarrow \backslash/[AP, 0..6]$; AP $\rightarrow I \{ \leq < = \neq \} I$ isPrime(M) !isPrime(M) I $\rightarrow 0 \mid \mathbf{a} \mid \mathbf{y} \mid \mathbf{M} \mid \neg \mathbf{M}$ } </pre> (b) DSL $\mathcal{O}$	<pre> C1: $0 \leq \mathbf{y}$ C2: $\mathbf{y} < \mathbf{M}$ C3: $(\mathbf{a} = 0) \Rightarrow (\mathbf{y} = 0)$ C4: $(\mathbf{a} = \mathbf{M}) \Rightarrow (\mathbf{y} = 0)$ C5: $(\mathbf{a} = \neg \mathbf{M}) \Rightarrow (\mathbf{y} = 0)$ </pre> (c) $\mathcal{O}$-consequences
---	---	--

Figure 5.1. (a) A query Ψ for identifying properties of modhash that hold for any choice of input x . Users declare variables and label existentially quantified ones with the keyword `exist`. (b) A DSL $\mathcal{O}$ for over-approximations. $\backslash/[AP, 0..6]$ is a shorthand for the disjunction of 0 to 6 atomic propositions. (c) $\mathcal{O}$ -consequences by our tool ASPIRE when given the query Ψ and the DSL $\mathcal{O}$. We write $p \Rightarrow q$ instead of $\neg p \vee q$ for readability.

§5.8 discusses related work. §5.9 concludes.

5.1 Motivating Examples

In this section, we illustrate how the LOUD framework can be used to synthesize useful over-approximated (§5.1.1) and under-approximated (§5.1.2) properties of programs.

```
int modhash (int a, M, x) { return a * x mod M; }
```

Consider the parametric hash function above, where x is an integer input and a and $M > 0$ are possible parameters—i.e., a and M are fixed in a specific implementation of modhash. Intuitively, modhash can be viewed as a family of hash functions where the variable x is the input.

In LOUD, to reason about the behavior of a program, one provides a logical query they are interested in over- or under-approximating with properties in a given language. For example, one may provide the query Ψ in Figure 5.1a, which is equivalent to the existentially quantified formula $\exists x. y = ax \bmod M$. In this example, our goal is to identify which choices of parameters a and M will make modhash surjective onto $\mathbb{Z}_M = \{0, 1, \dots, M-1\}$. Specifically, we seek properties that capture the relationship between the output variable y and the parameters a and M (colored in red). To do so, we treat the input x as a nondeterministic parameter (existentially quantified and thus colored in blue).

5.1.1 Over-Approximate Reasoning

We start with properties that are consequences (over-approximations) of the query in Figure 5.1a. That is, we want formulae $\varphi(y, a, M)$ such that $\forall y, a, M. (\exists x. y = ax \bmod M) \Rightarrow \varphi(y, a, M)$.

As argued in chapter 4, different applications—e.g., generating type judgments or generating algebraic specifications—require formulae to adhere to a specific syntactic fragment. Thus in our framework, users are in charge of providing a DSL $\mathcal{L}$ as an input to express properties they are interested in; we call properties expressible in this DSL $\mathcal{L}$ -properties. We note that the DSL $\mathcal{L}$ might contain user-given functions for which an implementation should be provided. Furthermore, we say an $\mathcal{L}$ -property φ is an $\mathcal{L}$ -consequence if it is a consequence of the given query formula. One goal of our framework is to synthesize a set of incomparable *strongest* $\mathcal{L}$ -consequences.

In our example, the user provides the DSL $\mathcal{O}$ shown in Figure 5.1b, which includes the constant 0, all free variables and comparison operations between them, and also the user-defined function `isPrime` (together with its implementation) that is potentially related to the problem.

An incomparable set of strongest $\mathcal{O}$ -consequences for the query Ψ is shown in Figure 5.1c. Since conjoining two consequences of Ψ results in a stronger consequence of Ψ , we interpret the set of properties as their conjunction and thus call the set an $\mathcal{O}$ -conjunction. The properties in Figure 5.1c give us insights into the behavior of the function `modhash`—e.g., that setting the value of a to be equal to M or $-M$ is probably not a good idea as it would result in a function that always returns $y = 0$. For our discussion, we focus our attention on the first two properties, which imply that the output y falls within $0 \leq y < M$. A well-designed hash function with a set S as range should be surjective onto the set S , meaning that for every value v in S , there should be inputs that yield v as output. However, because the formulae in Figure 5.1c are over-approximations, we are not guaranteed that all the values in $0 \leq y < M$ are indeed possible outputs of `modhash`.

$$\begin{array}{ll}
\text{I1: } y = 0 & \text{I2: } 0 \leq a \wedge a < M \wedge a = y \\
\text{I3: } 0 \leq y \wedge y < M \wedge -M < a \wedge a < M \wedge a \neq 0 \wedge \text{isPrime}(M)
\end{array}$$

Figure 5.2. Synthesized $\mathcal{U}$ -implicants.

5.1.2 Under-Approximate Reasoning

Over-approximation alone cannot capture whether a specific program behavior *can* occur—i.e., is reachable. For a formula $\varphi(y, a, M)$ to define a reachability condition (i.e., a behavior that *must* happen) of y from some input x , the formula $\varphi(y, a, M)$ must be an implicant (under-approximation) of the query Ψ , which formally can be stated as follows:

$$\forall y, a, M. \varphi(y, a, M) \Rightarrow \exists x. y = ax \bmod M$$

We say an $\mathcal{L}$ -property φ is an $\mathcal{L}$ -implicant if it is an implicant of the given query formula. Another goal of our framework is to synthesize a set of incomparable *weakest* $\mathcal{L}$ -implicants.

In our example, the user provides the DSL $\mathcal{U}$ using the rules for AP and I as shown in Figure 5.1b, but replaces disjunction rules (nonterminal D) with the conjunction rules C $\rightarrow$ $\wedge$ [AP, 0..6]. Throughout the chapter we will use $\mathcal{O}$ to denote the language from Figure 5.1b in examples involving over-approximations, and $\mathcal{U}$ for the language described here in examples involving under-approximation. For instance, $a = 0 \wedge y = 0$ is a $\mathcal{U}$ -implicant for query Ψ , but not a weakest one, as it strictly implies a $\mathcal{U}$ -implicant $y = 0$.

A mutually incomparable set of weakest $\mathcal{U}$ -implicants for the query Ψ is shown in Figure 5.2. We interpret the set of properties as their disjunction and refer to the set as a $\mathcal{U}$ -disjunction. Each formula provides a sufficient condition for reachability of the output y —that is, if y , a and M satisfy any formula in Figure 5.2, then there exists an input x such that $y = ax \bmod M$. Crucially, the last formula provides a sufficient condition for modhash to be surjective onto $\mathbb{Z}_M = \{0, 1, \dots, M-1\}$ —i.e., for a prime value of M and non-zero value of a selected from the range $-M < a < M$, all values of y in $\mathbb{Z}_M$ are attainable from some choice of input x .

While our primary motivation is to model nondeterminism, the generality of our frame-

work enables a variety of other applications. In particular, existential quantifiers can be used to model both forward and backward reasoning in the style of Hoare and incorrectness program logics for both deterministic and nondeterministic programs. For example, given a program and a predicate over its output, one can synthesize preconditions on what input may produce (over-approximate) or must produce (under-approximate) an output that satisfies the given predicate.

A detailed discussion of various applications of our framework is provided in Section 5.7 and Appendix B.

5.2 Strongest Consequences and Weakest Implicants

In this section, we describe our framework, which extends the strongest $\mathcal{L}$ -consequences synthesis framework of chapter 4 in two key ways: (i) allowing existentially quantified variables, and (ii) enabling the synthesis of both the strongest $\mathcal{L}$ -consequences and weakest $\mathcal{L}$ -implicants.

We describe what inputs a user of the framework provides, and what they obtain as output.

Input 1: Query. The *query* Ψ is a first-order formula of the form $\exists h. \psi(v, h)$, where ψ is a quantifier-free formula ψ . The inclusion of the existentially quantified variables h in the query is a key novelty of this chapter: it enables many new applications such as reasoning about nondeterministic programs, and forward and backward reasoning in program logics (Section 5.7.2).

We use the symbol h (for hidden) to represent existentially quantified variables and the symbol v (for visible) to represent free variables. In practice, both h and v can be tuples and denote multiple variables. In our motivating examples, queries are given in Figure 5.1a.

Input 2: Grammar of $\mathcal{L}$ -properties. The grammar of the DSL $\mathcal{L}$ in which the synthesizer is to express properties for the query. Each formula ϕ in the DSL $\mathcal{L}$ is a predicate defined over the free variables v of the query $\exists h. \psi(v, h)$. An example of a DSL is the language $\mathcal{O}$ in Figure 5.1b.

Input 3: Semantics of the program and operators. Semantics of the function symbols that appear in query Ψ (e.g., `modhash`) and in the DSL $\mathcal{L}$ (e.g., `isPrime`). In our implementation, semantic definitions are given as a program in the SKETCH language [145], which automatically transforms them into first-order formulas. For example, one may provide a C-style function written in SKETCH that checks whether a number between 2 and $\sqrt{n}$ is a divisor of n as the semantic of `isPrime`. We discuss in Section 5.6 how SKETCH works and examine the limitations of this approach.

Output: Strongest $\mathcal{L}$ -consequences and weakest $\mathcal{L}$ -implicants. Our goal is to synthesize a set of incomparable *strongest* $\mathcal{L}$ -consequences and a set of incomparable *weakest* $\mathcal{L}$ -implicants of query Ψ . Ideally, both strongest $\mathcal{L}$ -consequence and weakest $\mathcal{L}$ -implicant would be the formula that is exactly equivalent to $\exists h. \psi(v, h)$, but in general, the DSL $\mathcal{L}$ might not be expressive enough to do so. As argued in chapter 4, this feature is actually a desired one as it allows for the application of our framework to various use cases, as demonstrated in Section 5.7. Because in general there might not be an $\mathcal{L}$ -consequence and an $\mathcal{L}$ -implicant that are equivalent to $\exists h. \psi(v, h)$, the goal becomes instead to find $\mathcal{L}$ -properties that tightly approximate $\exists h. \psi(v, h)$.

We denote the set of models (over the free variables of Ψ) of a formula φ as $\llbracket \varphi \rrbracket$. For example in Section 5.1.1, $\llbracket y \geq 0 \rrbracket$ represents the set of models $\{(y, a, M) \mid y \geq 0\}$. We say φ is stronger than φ' (or φ' is weaker than φ) when $\llbracket \varphi \rrbracket \subseteq \llbracket \varphi' \rrbracket$, and φ is strictly stronger than φ' when $\llbracket \varphi \rrbracket \subset \llbracket \varphi' \rrbracket$.

Definition 5.2.1 (A strongest $\mathcal{L}$ -consequence). *An $\mathcal{L}$ -property φ is a strongest $\mathcal{L}$ -consequence for a query Ψ if and only if*

(i) *φ is a consequence of the query Ψ :*

$$\text{ISCONS}_{\Psi}(\varphi) := \forall v. [\exists h. \psi(v, h) \Rightarrow \varphi(v)]$$

(ii) φ is strongest with respect to Ψ and $\mathcal{L}$:

$$\neg \exists \varphi' \in \mathcal{L}. \text{ISCONS}_{\Psi}(\varphi') \wedge \llbracket \varphi' \rrbracket \subset \llbracket \varphi \rrbracket$$

Definition 5.2.2 (A weakest $\mathcal{L}$ -implicant). *An $\mathcal{L}$ -property φ is a weakest $\mathcal{L}$ -implicant for a query Ψ if and only if*

(i) φ is an implicant of the query Ψ :

$$\text{ISIMPL}_{\Psi}(\varphi) := \forall v. [\varphi(v) \Rightarrow \exists h. \psi(v, h)]$$

(ii) φ is weakest with respect to Ψ and $\mathcal{L}$:

$$\neg \exists \varphi' \in \mathcal{L}. \text{ISIMPL}_{\Psi}(\varphi') \wedge \llbracket \varphi' \rrbracket \supset \llbracket \varphi \rrbracket$$

Throughout the chapter, we also use the term *most-precise* to mean strongest for $\mathcal{L}$ -consequences and weakest for $\mathcal{L}$ -implicants. We use $SCons_{\mathcal{L}}(\Psi)$ and $WImpl_{\mathcal{L}}(\Psi)$ to denote the set of all strongest $\mathcal{L}$ -consequences and the set of all best $\mathcal{L}$ -implicants for Ψ , respectively. Because $\mathcal{L}$ may not be closed under conjunction (and disjunction), strongest $\mathcal{L}$ -consequences (and weakest $\mathcal{L}$ -implicants) may not be semantically unique. In Section 5.1.1, both formulae $0 \leq y$ and $y < M$ are strongest $\mathcal{L}$ -consequences of query Ψ , and neither implies the other.

The goal of our framework is to find a semantically strongest conjunction of incomparable strongest $\mathcal{L}$ -consequences and a weakest disjunction of incomparable weakest $\mathcal{L}$ -implicants.

Definition 5.2.3 (Best $\mathcal{L}$ -conjunction). *A potentially infinite set of $\mathcal{L}$ -consequences $\Pi = \{\varphi_i\}$ forms a best $\mathcal{L}$ -conjunction $\varphi_{\wedge} = \bigwedge_i \varphi_i$ for query Ψ if and only if*

(i) each $\varphi_i \in \Pi$ is a strongest $\mathcal{L}$ -consequence of Ψ ;

(ii) every distinct $\varphi_i, \varphi_j \in \Pi$ are incomparable—i.e., $\llbracket \varphi_i \rrbracket \setminus \llbracket \varphi_j \rrbracket \neq \emptyset$ and $\llbracket \varphi_j \rrbracket \setminus \llbracket \varphi_i \rrbracket \neq \emptyset$;

(iii) the set is semantically minimal—i.e., $\llbracket \varphi_{\wedge} \rrbracket \subseteq \llbracket \varphi \rrbracket$ for every strongest $\mathcal{L}$ -consequence φ .

Definition 5.2.4 (Best $\mathcal{L}$ -disjunction). *A potentially infinite set of $\mathcal{L}$ -implicants $\Pi = \{\varphi_i\}$ forms a best $\mathcal{L}$ -disjunction $\varphi_\vee = \bigvee_i \varphi_i$ for query Ψ if and only if*

- (i) *each $\varphi_i \in \Pi$ is a weakest $\mathcal{L}$ -implicant of Ψ ;*
- (ii) *every distinct $\varphi_i, \varphi_j \in \Pi$ are incomparable—i.e., $\llbracket \varphi_i \rrbracket \setminus \llbracket \varphi_j \rrbracket \neq \emptyset$ and $\llbracket \varphi_j \rrbracket \setminus \llbracket \varphi_i \rrbracket \neq \emptyset$;*
- (iii) *the set is semantically maximal—i.e., $\llbracket \varphi_\vee \rrbracket \supseteq \llbracket \varphi \rrbracket$ for every weakest $\mathcal{L}$ -implicant φ .*

Best $\mathcal{L}$ -conjunctions and best $\mathcal{L}$ -disjunctions are not necessarily unique, but they are all logically equivalent. Specifically, a best $\mathcal{L}$ -conjunction is equivalent to the conjunction of all possible strongest $\mathcal{L}$ -consequences, and a best $\mathcal{L}$ -disjunction is equivalent to the disjunction of all possible weakest $\mathcal{L}$ -implicants. Note that *best* means more than semantic optimality because a strongest $\mathcal{L}$ -conjunction is not necessarily a best $\mathcal{L}$ -conjunction; predicates $\varphi_1(x) := x \geq 0$ and $\varphi_2(x) := x \geq 1$ could form a strongest $\mathcal{L}$ -conjunction, but it is not a best one because φ_1 is strictly stronger than φ_2 —i.e., φ_1 and φ_2 are comparable.

Theorem 5.2.5 (Semantic Optimality). *If $\varphi_\wedge$ is a best $\mathcal{L}$ -conjunction, then its interpretation coincides with the conjunction of all possible strongest $\mathcal{L}$ -consequences: $\llbracket \varphi_\wedge \rrbracket = \llbracket \bigwedge_{\varphi \in SCons_{\mathcal{L}}(\Psi)} \varphi \rrbracket$. If $\varphi_\vee$ is a best $\mathcal{L}$ -disjunction, then its interpretation coincides with the disjunction of all possible best $\mathcal{L}$ -implicants: $\llbracket \varphi_\vee \rrbracket = \llbracket \bigvee_{\varphi \in WImpl_{\mathcal{L}}(\Psi)} \varphi \rrbracket$.*

We are now ready to state our problem definition:

Definition 5.2.6 (Best $\mathcal{L}$ -conjunction and $\mathcal{L}$ -disjunction Synthesis). *Given query Ψ , the concrete semantics for the function symbols in Ψ , and a domain-specific language $\mathcal{L}$ with its corresponding semantic definition, synthesize a best $\mathcal{L}$ -conjunction and $\mathcal{L}$ -disjunction for Ψ .*

Practical remarks.

The algorithm presented in Section 5.4 computes finite $\mathcal{L}$ -conjunctions and finite $\mathcal{L}$ -disjunctions, but it is possible that a solution to a given LOUD problem instance requires an infinite number of conjuncts (or disjuncts). Even in this case, because the algorithm incrementally computes incomparable strongest $\mathcal{L}$ -consequences (or weakest $\mathcal{L}$ -implicants), one can stop it at

any point and all computed $\mathcal{L}$ -consequences (or $\mathcal{L}$ -implicants) will form a valid $\mathcal{L}$ -conjunction (or $\mathcal{L}$ -disjunction), just not a best one. The benchmarks considered in Section 5.7 do not encounter this problem since they consider finite (though very large) languages.

Note that Definition 5.2.3 and 5.2.4 only guarantee that each *pair* of synthesized properties is incomparable. However, ensuring that each φ_i is incomparable to the entire set of properties $\Pi \setminus \{\varphi_i\}$ only requires checking a single implication for each φ_i , which can be trivially done in a postprocessing phase. In practice, it is quite rare to encounter a redundant conjunct or disjunct.

5.3 Relation to Program Logics

In this section, we discuss how our problem formulation relates to the type of reasoning program logics like Hoare [70] and incorrectness logic [110] can do with respect precondition-/presumptions and postconditions/results.

5.3.1 Relation to Hoare Logic

We start by considering Hoare logic and its ability to reason about correctness properties of programs. A Hoare triple $\{P\}s\{Q\}$ consists of a precondition P , a statement s , and a postcondition Q , and it has the following meaning: if the precondition P holds before executing s and s terminates, then the postcondition Q holds upon termination. In other words, the postcondition *over-approximates* the set of possible behaviors the program can result in.

We assume the semantics of a program s is given by a relation $\llbracket s \rrbracket(\sigma, \sigma')$, which holds true if s on input state σ can terminate with an output state σ' . The meaning of the triple $\{P\}s\{Q\}$ can then be formalized as follows:

$$\forall \sigma, \sigma'. P(\sigma) \wedge \llbracket s \rrbracket(\sigma, \sigma') \Rightarrow Q(\sigma') \quad (5.1)$$

Backward Reasoning: Weakest Liberal Precondition

Weakest precondition operations can be formalized as predicate transformers that assign a unique (in a sense, most general) precondition P to each program s and postcondition Q . Given a program s and a postcondition Q , the *weakest liberal precondition* $wlp(s, Q)$ represents the weakest predicate P such that the triple $\{P\} s \{Q\}$ holds [38]. If we view $wlp(s, Q)$ as a backward predicate transformer, it reformulates the problem of verifying the triple $\{P\} s \{Q\}$ to the problem of checking a first-order formula $P \Rightarrow wlp(s, Q)$.¹

The problem of computing such a predicate transformer, and in particular the weakest one expressible as a *disjunction* of predicates in the DSL $\mathcal{L}$, can be phrased in our framework. The need to lift to *disjunction* of predicates arises because there may exist multiple incomparable predicates in $\mathcal{L}$ that satisfy the triple $\{P\} s \{Q\}$ and cannot be further weakened (while still satisfying $\{P\} s \{Q\}$). Thus, we define $wlp_{\mathcal{L}}(s, Q)$ as follows: if P is a weakest predicate in $\mathcal{L}$ such that $\{P\} s \{Q\}$ holds true, then it must also hold true that $P \Rightarrow wlp_{\mathcal{L}}(s, Q)$.

Definition 5.3.1 ($\mathcal{L}$ -weakest liberal precondition). *Given a program s , a postcondition Q and a DSL $\mathcal{L}$, the $\mathcal{L}$ -weakest liberal precondition $wlp_{\mathcal{L}}(s, Q)$ is the (possibly infinite) disjunction $\bigvee_i P_i$ of all predicates $P_i \in \mathcal{L}$ such that (i) $\{P_i\} s \{Q\}$ holds true, and (ii) no $P \in \mathcal{L}$ is strictly implied by P_i while $\{P\} s \{Q\}$ holds true.*

Note that if the DSL $\mathcal{L}$ is expressive enough, the $\mathcal{L}$ -weakest liberal precondition $wlp_{\mathcal{L}}(s, Q)$ will be equivalent to the weakest liberal precondition $wlp(s, Q)$ —i.e., the DSL would be what in Hoare logic is called an expressive enough assertion language.

Because $\mathcal{L}$ -weakest liberal precondition is a weakest formula one may be tempted to compute it via weakest $\mathcal{L}$ -implicants. However, this approach would not yield the desired result because an $\mathcal{L}$ -implicant only proves the existence of an execution satisfying the postcondition

¹Dijkstra’s original weakest precondition requires that whenever the precondition P holds before the execution of s , the execution of s is guaranteed to terminate [37]. While our over-approximation framework can elegantly capture the notion of weakest liberal precondition, reasoning about Dijkstra’s weakest precondition and total correctness is problematic because semantics encoding presented in this section describe *possible* end states σ' from an initial state σ , without addressing whether some executions may not terminate.

Q , but it does not prove whether *every* execution satisfies the postcondition Q .

For example, consider a simple DSL that can only express the properties true and false. Given a nondeterministic program $x = *$ (i.e., one that nondeterministically assigns an integer to x) and a postcondition $Q(x) := x = 0$, the property “true” would be an $\mathcal{L}$ -implicant of the query $\exists x. (x = * \wedge x = 0)$ because there always *exist* an execution that results in x being zero, while the formula “false” is a valid $\mathcal{L}$ -weakest liberal precondition for the query $\exists x. x = *$ and the postcondition $x = 0$ because there is no precondition that *ensures* x will be zero after the execution. As we will see in Section 5.3.2 $\mathcal{L}$ -implicants can be used to compute the *weakest possible precondition*, a different backward predicate transformer used in reverse Hoare logic and incorrectness logic.

We next show that by *negating* every predicate (precondition, postcondition, and language or properties), we can capture $\mathcal{L}$ -weakest liberal precondition via $\mathcal{L}$ -consequences. First, observe that by appropriately negating the pre- and postcondition, Equation (5.1) can be rewritten as follows:

$$\forall \sigma. [\exists \sigma'. \neg Q(\sigma') \wedge \llbracket s \rrbracket(\sigma, \sigma')] \Rightarrow \neg P(\sigma) \quad (5.2)$$

Intuitively, Equation (5.1) states that every state σ' that violates Q must come from a state σ that violates P . Thus, we can introduce a DSL for negated formulae $\overline{\mathcal{L}} = \{\neg \varphi \mid \varphi \in \mathcal{L}\}$ and have that any predicate P satisfies conditions (i) and (ii) of Definition 5.3.1 if and only if $\neg P$ is a strongest $\overline{\mathcal{L}}$ -consequence of the query:

$$\exists \sigma'. \neg Q(\sigma') \wedge \llbracket s \rrbracket(\sigma, \sigma') \quad (5.3)$$

By combining Theorem 5.2.5 and De Morgan’s laws with the above observation, we can prove the equivalence of $wlp_{\mathcal{L}}(s, Q)$ and a best $\overline{\mathcal{L}}$ -conjunction for the query (5.3).

Theorem 5.3.2. *The $\mathcal{L}$ -weakest liberal precondition $wlp_{\mathcal{L}}(s, Q)$ is semantically equivalent to the negation of a best $\overline{\mathcal{L}}$ -conjunction of query $\Psi := \exists \sigma'. \neg Q(\sigma') \wedge \llbracket s \rrbracket(\sigma, \sigma')$.*

For example, to compute $\mathcal{O}$ -weakest liberal precondition for the program $y = ax \bmod M$, a postcondition $y = 0$ and a DSL $\mathcal{O}$ defined in Section 5.1.1, we introduce a DSL for negated formulae $\overline{\mathcal{O}} = \{\neg\phi \mid \phi \in \mathcal{O}\}$. Then the problem of computing $wlp_{\mathcal{O}}(y = ax \bmod M, y = 0)$ is encoded as the problem of synthesizing a best $\overline{\mathcal{O}}$ -conjunction for the query:

$$\exists y. \neg(y = 0) \wedge y = ax \bmod M \quad (5.4)$$

The following set of $\overline{\mathcal{O}}$ -consequences forms a best $\overline{\mathcal{O}}$ -conjunction for the query (5.4):

$$a \neq 0 \quad x \neq 0 \quad a \neq M \quad x \neq M \quad (5.5)$$

By negating the formulas in Equation (5.5) we get the following disjunction—i.e., the $\mathcal{L}$ -weakest liberal precondition:

$$a = 0 \vee x = 0 \vee a = M \vee x = M$$

Forward Reasoning: Strongest Postcondition

Strongest postcondition predicate transformers can be thought of being the dual of the weakest precondition ones. Given a program s and a precondition P , the *strongest postcondition* $spo(s, P)$ represents the strongest predicate Q such that the triple $\{P\}s\{Q\}$ holds [39]. If we view $spo(s, Q)$ as a forward predicate transformer, we can reformulate the problem of verifying the triple $\{P\}s\{Q\}$ as the problem of checking whether a first-order formula $spo(s, P) \Rightarrow Q$ holds.

The problem of computing such a predicate transformer, and in particular the strongest one expressible as a *conjunction* of predicates in the DSL $\mathcal{L}$, can be phrased in our framework. Similar to the case of the weakest liberal precondition, we define $spo_{\mathcal{L}}(s, P)$ as follows: if Q is a strongest predicate in $\mathcal{L}$ such that $\{P\}s\{Q\}$ holds true, then it must also hold true that $spo_{\mathcal{L}}(s, P) \Rightarrow Q$ —i.e., the $\mathcal{L}$ -strongest postcondition $spo_{\mathcal{L}}(s, P)$ is stronger than any strongest postcondition Q in $\mathcal{L}$.

Definition 5.3.3 ($\mathcal{L}$ -strongest postcondition). *Given a program s , a precondition P and a DSL $\mathcal{L}$, the $\mathcal{L}$ -strongest postcondition $\text{spo}_{\mathcal{L}}(s, P)$ is the (possibly infinite) conjunction $\bigwedge_i Q_i$ of all predicates $Q_i \in \mathcal{L}$ such that (i) $\{P\}s\{Q_i\}$ holds true, and (ii) no $Q \in \mathcal{L}$ strictly implies Q_i while $\{P\}s\{Q\}$ holds true.*

The problem of obtaining the $\mathcal{L}$ -strongest postcondition $\text{spo}_{\mathcal{L}}(s, P)$ can also be encoded as synthesizing strongest $\mathcal{L}$ -consequences, this time without a need of negating formulas. Observe that relocating the quantifier $\forall \sigma$ into the implicant rewrites Equation (5.1) as follows:

$$\forall \sigma'. [\exists \sigma. P(\sigma) \wedge \llbracket s \rrbracket(\sigma, \sigma')] \Rightarrow Q(\sigma') \quad (5.6)$$

From Equation (5.6) we have that a predicate $Q \in \mathcal{L}$ satisfies conditions (i) and (ii) of Definition 5.3.3 if and only if Q is a strongest $\mathcal{L}$ -consequence of the following query:

$$\exists \sigma. P(\sigma) \wedge \llbracket s \rrbracket(\sigma, \sigma') \quad (5.7)$$

This observation yields the following theorem:

Theorem 5.3.4. *The $\mathcal{L}$ -strongest postcondition $\text{spo}_{\mathcal{L}}(s, P)$ is semantically equivalent to the best $\mathcal{L}$ -conjunction of the query $\Psi := \exists \sigma. P(\sigma) \wedge \llbracket s \rrbracket(\sigma, \sigma')$.*

Section 5.1.1 illustrated how to obtain a $\mathcal{O}$ -strongest postcondition $\text{spo}_{\mathcal{O}}(y = ax \bmod M, \top)$ using over-approximated reasoning in our framework.

5.3.2 Relation to Reverse Hoare Logic and Incorrectness Logic

Reverse Hoare logic [35] and incorrectness logic [110] can both be thought of being the dual of Hoare logic—they under-approximate (instead of overapproximate) the set of possible behaviors a program can result in. The under-approximating logics are semantically equivalent despite having been designed with different goals in mind: reverse Hoare logic was designed to reason about the correctness of nondeterministic programs, whereas incorrectness logic was

designed to identify the presence of bugs in programs. Our evaluation in Section 5.7 investigates both of these applications.

A *incorrectness triple* $[P]s[Q]$ consists of a presumption P , a statement s , and a result Q , and it has the following meaning: every final state satisfying Q is reachable by executing program s starting from *some* state that satisfies presumption P . In other words, the predicate Q *under-approximates* the set of possible behaviors the program s can result in when executed on inputs satisfying P :

$$\forall \sigma'. Q(\sigma') \Rightarrow \exists \sigma. [P(\sigma) \wedge \llbracket s \rrbracket(\sigma, \sigma')] \quad (5.8)$$

Forward Reasoning: Weakest Under-approximate Postcondition

Weakest postcondition operations can be formalized as predicate transformers that assign a unique precondition Q to each program s and precondition P . Given a program s and a presumption P , the *weakest under-approximate postcondition* $wupo(s, P)$ represents the weakest predicate Q such that the triple $[P]s[Q]$ holds. If we view $wupo(s, P)$ as a forward predicate transformer, we can reformulate the problem of verifying the triple $[P]s[Q]$ as the problem of checking whether a first-order formula $Q \Rightarrow wupo(s, P)$ holds.

The problem of computing such a predicate transformer, and in particular the weakest one expressible as a *disjunction* of predicates in the DSL $\mathcal{L}$, can be phrased in our framework. We define $wupo_{\mathcal{L}}(s, P)$ as follows: if Q is a weakest predicate in $\mathcal{L}$ such that $[P]s[Q]$ holds true, then it must also hold true that $Q \Rightarrow wupo_{\mathcal{L}}(s, P)$.

Definition 5.3.5 ($\mathcal{L}$ -weakest under-approximate postcondition). *Given a program s , a presumption P and a DSL $\mathcal{L}$, the $\mathcal{L}$ -weakest under-approximate postcondition $wupo_{\mathcal{L}}(s, P)$ is the (possibly infinite) disjunction $\bigvee Q_i$ of all predicates $Q_i \in \mathcal{L}$ such that (i) $[P]s[Q_i]$ holds true, and (ii) no $Q \in \mathcal{L}$ is strictly implied by Q_i while $[P]s[Q]$ holds true.*

Following Equation (5.8), the problem of obtaining the $\mathcal{L}$ -weakest under-approximate postcondition $wupo_{\mathcal{L}}(s, P)$ can be directly encoded as synthesizing weakest $\mathcal{L}$ -implicants. A predicate $Q \in \mathcal{L}$ satisfies conditions (i) and (ii) of Definition 5.3.5 if and only if Q is a weakest

$\mathcal{L}$ -implicant of the following query:

$$\exists \sigma. P(\sigma) \wedge \llbracket s \rrbracket(\sigma, \sigma') \quad (5.9)$$

This observation yields the following theorem:

Theorem 5.3.6. *The $\mathcal{L}$ -weakest under-approximate postcondition $wupo_{\mathcal{L}}(s, P)$ is semantically equivalent to the best $\mathcal{L}$ -disjunction of the query $\Psi := \exists \sigma. P(\sigma) \wedge \llbracket s \rrbracket(\sigma, \sigma')$.*

Backward Reasoning: Weakest Possible Precondition

While forward predicate transformers for incorrectness logic behave well—i.e., given a presumption P and a program s , one can always assign the weakest result Q such that $[P] s [Q]$ holds true—backward predicate transformers for incorrectness logic do not always exist! This problem arises because valid presumptions may not exist in incorrectness logic. For example, there is no predicate P making the triple $[P] y = ax \bmod M [y = -1]$ true because no values of a , M and x satisfies $ax \bmod M = -1$. To address this shortcoming and still take advantage of some form of backward reasoning in incorrectness logic, O’Hearn [110] suggests using the weakest possible precondition $wpp(s, Q)$, which is predicate transformer described by Hoare [71] for what he referred as “possible correctness”. Intuitively, $wpp(s, Q)$ captures the set of initial states from which it is *possible* to execute s and terminate in a state that satisfies Q .

O’Hearn then proposes to use a two phase approach to derive valid incorrectness triples as follows. Starting with a postcondition Q , one first computes the weakest possible precondition $P = wpp(s, Q)$ and then applies forward reasoning and computes the weakest under-approximate postcondition $Q' = wupo(s, P)$ to obtain a valid incorrectness triple $[P] s [Q']$. Since we already showed how to capture the weakest under-approximate postcondition in our framework, we now show how to capture the weakest possible precondition.

A predicate P is called a *possible precondition* of predicate Q for program s , if every input state satisfying P has a run of the program s that terminates to an end state satisfying Q .

That is, $wpp(s, Q)$ is the weakest P satisfying

$$\forall \sigma. P(\sigma) \Rightarrow [\exists \sigma'. Q(\sigma') \wedge \llbracket s \rrbracket(\sigma, \sigma')] \quad (5.10)$$

Note that $P = wpp(s, Q)$ forms neither a Hoare nor an incorrectness triple with the program s and the postcondition Q . The postcondition Q is not a valid over-approximation of possible final states because there could be other executions that do not satisfy Q —e.g., if the program is nondeterministic. The postcondition Q is not a valid under-approximation either because there might be states satisfying Q that are not reachable—e.g., if the postcondition Q is $\top$ and the program s is $y = ax \bmod M$, then any negative value of y satisfies the postcondition Q , but no values of a , M and x can yield a negative output. As proposed by O’Hearn, we can remedy this issue by computing a new postcondition $Q' = wupo(s, P)$ using the weakest under-approximate postcondition operator to obtain a valid incorrectness triple $[P] s [Q']$.

The problem of computing such a predicate transformer, and in particular the weakest one expressible as a *disjunction* of predicates in the DSL $\mathcal{L}$, can be phrased in our framework.

Definition 5.3.7 ($\mathcal{L}$ -weakest possible precondition). *Given a program s , a postcondition Q and a DSL $\mathcal{L}$, the $\mathcal{L}$ -weakest possible precondition $wpp_{\mathcal{L}}(s, Q)$ is the (possibly infinite) disjunction $\bigvee_i P_i$ of all predicates $P_i \in \mathcal{L}$ such that (i) Equation (5.10) holds true, and (ii) no $P \in \mathcal{L}$ is strictly implied by P_i while Equation (5.10) holds true.*

From Equation (5.10), the problem of obtaining $\mathcal{L}$ -weakest possible precondition $wpp_{\mathcal{L}}(s, Q)$ can be directly encoded as synthesizing weakest $\mathcal{L}$ -implicants. A predicate $P \in \mathcal{L}$ satisfies conditions (i) and (ii) of Definition 5.3.7 if and only if P is a weakest $\mathcal{L}$ -implicant of the following query:

$$\exists \sigma'. Q(\sigma') \wedge \llbracket s \rrbracket(\sigma, \sigma') \quad (5.11)$$

This observation yields the following theorem:

Theorem 5.3.8. *The $\mathcal{L}$ -weakest possible precondition $wpp_{\mathcal{L}}(s, Q)$ is semantically equivalent to the best $\mathcal{L}$ -disjunction of the query $\Psi := \exists \sigma'. Q(\sigma') \wedge \llbracket s \rrbracket(\sigma, \sigma')$.*

5.4 Counterexample-Guided Inductive Specification Synthesis

In this section, we present algorithms for synthesizing best $\mathcal{L}$ -conjunctions and best $\mathcal{L}$ -disjunctions. We follow the example-guided approach proposed in chapter 4 that synthesizes strongest $\mathcal{L}$ -consequences. In tandem, we present a dual algorithm to synthesize weakest $\mathcal{L}$ -implicants, and we extend both algorithms to allow existentially quantified query formulas.

We first present the primitives necessary to instantiate the synthesis algorithms (Section 5.4.1). Then, we present the algorithms for synthesizing a single $\mathcal{L}$ -consequence or $\mathcal{L}$ -implicant that is incomparable to all the ones synthesized so far (Section 5.4.2). Finally, we present the algorithms for iteratively synthesizing the properties forming an $\mathcal{L}$ -conjunction or an $\mathcal{L}$ -disjunction (Section 5.4.3).

5.4.1 Synthesis from Positive and Negative Examples

The algorithms for synthesizing strongest $\mathcal{L}$ -consequences and weakest $\mathcal{L}$ -implicants maintain two sets of examples: a set of positive examples E^+ , which should be accepted by the synthesized predicates, and negative examples E^- , which should be rejected by the synthesized predicates.

Definition 5.4.1 (Examples). *Given a query $\Psi := \exists h. \psi(v, h)$ and a model e over the free variable v of query Ψ , we say that e is a positive example if $\psi(e, h)$ holds true for some value of h (i.e., $e \in \llbracket \Psi \rrbracket$) and a negative example if $\psi(e, h)$ does not hold for all values of h (i.e., $e \notin \llbracket \Psi \rrbracket$).*

Example 11 (Positive and Negative Examples). *Given the query $\Psi := \exists x. y = ax \bmod M$, the model that assigns y to the integer 1, a to the integer 6, and M to the integer 5 is a positive example, because the choice of value $x = 1$ makes the equation $1 = 6 \bmod 5$ holds true. For brevity, we*

represent such example as $(1,6,5)$, where it denotes a valuation to the tuple (y,a,M) . The following examples are negative ones because no value of x satisfies $y = ax \bmod M$: $(-1,1,3)$, $(3,1,3)$, $(3,2,6)$.

Because the DSL $\mathcal{L}$ might not be expressive enough to capture the exact behavior of the query Ψ , in general there is no predicate capable of accepting all the positive examples and rejecting all the negative examples. Intuitively, a strongest $\mathcal{L}$ -consequence must accept all positive examples while also excluding as many negative examples as possible.

Example 12 (Examples and $\mathcal{L}$ -consequences). *Consider again the query $\Psi := \exists x. y = ax \bmod M$ and the set of strongest $\mathcal{O}$ -consequences $\{0 \leq y, y < M, a = 0 \Rightarrow y = 0, a = M \Rightarrow y = 0, a = -M \Rightarrow y = 0\}$ from Figure 5.1c. While a positive example $(y,a,M) = (1,6,5)$ is accepted by all $\mathcal{O}$ -consequences, the negative example $(3,2,6)$ is not rejected by any of them. In fact, the $\mathcal{O}$ -consequences in Figure 5.1c form a best $\mathcal{L}$ -conjunction, so $(3,2,6)$ cannot be rejected by any strongest $\mathcal{O}$ -consequence.*

As illustrated by Example 12 when attempting to synthesize $\mathcal{L}$ -consequences, we can consider positive examples as hard constraints but need to treat negative examples as soft constraints.

For $\mathcal{L}$ -implicants, the role of positive and negative examples is inverted. A weakest $\mathcal{L}$ -implicant must reject all negative examples while also accepting as many positive examples as possible.

Example 13 (Examples and $\mathcal{L}$ -implicants). *Consider again the query $\Psi := \exists x. y = ax \bmod M$ and the set of weakest $\mathcal{U}$ -implicants $\{y = 0, 0 \leq a < M \wedge a = y, 0 \leq y < M \wedge -M < a < M \wedge a \neq 0 \wedge \text{isPrime}(M)\}$ from Figure 5.2. While the negative example $(y,a,M) = (3,2,6)$ is rejected by all $\mathcal{U}$ -implicants, the positive example $(1,6,5)$ is not accepted by any of them. The $\mathcal{U}$ -implicants in Figure 5.2 form a best $\mathcal{L}$ -disjunction, so $(1,6,5)$ must be rejected by every weakest $\mathcal{U}$ -implicants.*

We are now ready to introduce the generalizations of the key operations used by Park et al. [116] to synthesize strongest $\mathcal{L}$ -consequences: **SYNTHESIZE** (Section 5.4.1), **CHECKIMPLICATION** (Section 5.4.1) and **CHECKSTRONGEST** (Section 5.4.1). Additionally, we introduce **CHECKWEAKEST** (Section 5.4.1), an operation used alongside **SYNTHESIZE** and **CHECKIMPLICATION** to synthesize weakest $\mathcal{L}$ -implicants.

Synthesis from Examples

While strongest $\mathcal{L}$ -consequences and weakest $\mathcal{L}$ -implicants can effectively treat some of the examples as soft constraints, as we will show in Section 5.4.2, our synthesis algorithm can find such properties by iteratively calling a synthesis primitive **SYNTHESIZE** that treats a carefully chosen set of examples as hard constraints. Avoiding soft constraints was one of the key innovations of chapter 4 with respect to prior work [81].

Given a set of positive examples E^+ and a set of negative examples E^- , the procedure **SYNTHESIZE**(E^+, E^-) returns an $\mathcal{L}$ -property ϕ that accepts all the positive examples in E^+ and rejects all the negative examples in E^- , if such an $\mathcal{L}$ -property ϕ exists. If no such $\mathcal{L}$ -property exists, then **SYNTHESIZE**(E^+, E^-) returns $\perp$. Given a set of examples E , we write $\phi(E)$ to denote the conjunction $\bigwedge_{e \in E} \phi(e)$ and $\neg\phi(E)$ to denote the conjunction $\bigwedge_{e \in E} \neg\phi(e)$. The operation **SYNTHESIZE**(E^+, E^-) can be expressed as the formula $\exists\phi. \phi(E^+) \wedge \neg\phi(E^-)$.

Example 14 (**SYNTHESIZE**). *Example 12 showed there can be a negative example that no $\mathcal{L}$ -consequences can reject. With the DSL $\mathcal{O}$ defined in Figure 5.1b, if $E^+ = \{(1, 6, 5)\}$ and $E^- = \{(3, 2, 6)\}$, then **SYNTHESIZE**(E^+, E^-) can return the formula $\phi(y, a, M) := y < a$, which is not a consequence of the query $\Psi := \exists x. y = ax \bmod M$. In this case, once more positive examples are added to E^+ (which is something our synthesis algorithm automatically takes care of), **SYNTHESIZE** will return $\perp$. For example, if E^+ is augmented to the set $\{(1, 6, 5), (1, 1, 5), (1, -4, 5), (6, 2, 8)\}$, then **SYNTHESIZE**(E^+, E^-) returns $\perp$ —i.e., the negative example $(3, 2, 6)$ cannot be rejected by any $\mathcal{O}$ -consequences and our synthesis algorithm will later remove it from E^- .*

Checking Implication

The CHECKIMPLICATION primitive described in this section allows us to check whether a formula is valid $\mathcal{L}$ -implicant or a valid $\mathcal{L}$ -consequence.

Given two predicates φ and φ' , the CHECKIMPLICATION (φ, φ') checks whether φ is an implicant of φ' (dually, whether φ' is consequence of φ). In logical terms, CHECKIMPLICATION (φ, φ') checks whether there does not exist an example e that is accepted by φ but rejected by φ' ; it returns $\top$ or an example if the check fails. This check can be expressed as $\neg\exists e. \neg\varphi'(e) \wedge \varphi(e)$.

CHECKIMPLICATION (Ψ, φ) returns $\top$ if a predicate φ is a consequence of Ψ . Similarly, CHECKIMPLICATION (φ, Ψ) returns $\top$ if a predicate φ is an implicant of Ψ .

Example 15 (CHECKIMPLICATION). *Consider again the query $\Psi := \exists x. y = ax \bmod M$. Because the formula $\varphi(y, a, M) := y < a$ is not a consequence of the query Ψ , the primitive CHECKIMPLICATION (Ψ, φ) would return a positive example that is rejected by φ , such as $(1, -4, 5)$. On the other hand, calling CHECKIMPLICATION (Ψ, φ') on the formula $\varphi'(y, a, M) := y \geq 0$ would instead return $\top$ because the formula φ' is indeed a consequence of Ψ .*

Similarly, for the formula $\varphi(y, a, M) := y < a$, which is also not a implicant of Ψ , running CHECKIMPLICATION (φ, Ψ) (where this time the query Ψ is the second parameter) would return a negative example that is accepted by φ , such as $(-1, 1, 3)$. On the other hand, running CHECKIMPLICATION (φ', Ψ) on the formula $\varphi'(y, a, M) := y = 0$ would instead return $\top$ because the formula φ' is an implicant of Ψ .

The CHECKIMPLICATION procedure can be implemented using a constraint solver. However, the presence of quantifiers in implicants can result in a constraint with alternating quantifiers, making the check computationally harder, and most importantly, outside the capabilities of solvers that do not support quantifiers. We discuss a practical procedure for performing this check in Section 5.5.

Checking Precision

Checking precision—i.e., whether an $\mathcal{L}$ -consequence is strongest or whether an $\mathcal{L}$ -implicant is weakest—requires more sophisticated queries than the one described above. Specifically, one cannot simply ask whether there exists a negative example that is accepted by φ to check whether φ is a strongest $\mathcal{L}$ -consequence, because, as shown in Example 12, there might be some negative example that must be accepted by every strongest $\mathcal{L}$ -consequence.

In theory, to prove or disprove that an $\mathcal{L}$ -consequence φ is strongest one needs to check whether there exists an $\mathcal{L}$ -consequence φ' that is (i) a consequence of the query Ψ (i.e., φ' accepts all the positive examples) and (ii) strictly stronger than φ (i.e., φ' rejects at least one more negative example while rejecting all the negative examples that were already rejected by φ). Such a check would be too expensive as it effectively asks one to synthesize a provably sound $\mathcal{L}$ -consequence.

Our algorithm does not require such a powerful primitive, and instead approximates (i) and (ii) using a set of positive examples E^+ accepted by φ and a set of negative examples E^- rejected by φ . By combining implication and precision checks in a counterexample-guided fashion, our algorithm improves the approximation over time and is thus sound.

Given an $\mathcal{L}$ -consequence φ , a set of positive examples E^+ accepted by φ , a set of negative examples E^- rejected by φ , a query Ψ , and the $\mathcal{L}$ -consequences $\varphi_\wedge$ we have already synthesized, $\text{CHECKSTRONGEST}(\varphi, \varphi_\wedge, \Psi, E^+, E^-)$ checks if there do not exist an $\mathcal{L}$ -consequence φ' and a negative example $e \notin \llbracket \Psi \rrbracket$ satisfying $\varphi_\wedge$ such that: (i) φ' accepts all the positive examples in E^+ ; (ii) φ' rejects e and all the negative examples in E^- , whereas φ accepts e . In our algorithm, the formula $\varphi_\wedge$ is used to ensure that the example produced by CHECKSTRONGEST is not already rejected by best $\mathcal{L}$ -consequences we have already synthesized. The above check

can be logically stated as follows:

$$\begin{aligned} \text{CHECKSTRONGEST}(\varphi, \varphi_{\wedge}, \Psi, E^+, E^-) = \\ \neg \exists \varphi', e. \neg \Psi(e) \wedge \varphi_{\wedge}(e) \wedge \varphi(e) \wedge \neg \varphi'(e) \wedge \varphi'(E^+) \wedge \neg \varphi'(E^-) \end{aligned} \quad (5.12)$$

The highlighted part of the formula is what changes when checking if the formula is weakest.

Example 16 (CHECKSTRONGEST). Consider again the query $\Psi := \exists x. y = ax \bmod M$, and an $\mathcal{O}$ -consequence $\varphi(y, a, M) := y \neq M$, which is not a strongest one.

$\text{CHECKSTRONGEST}(\varphi, \top, \Psi, \{(1, 6, 5)\}, \{(3, 1, 3)\})$ can return a strictly stronger $\mathcal{O}$ -consequence $\varphi'_1(y, a, M) := y < M$ with a negative example $e_1^- = (6, 1, 5)$.

However, because CHECKSTRONGEST only considers whether the formula φ is strongest with respect to the examples $\{(1, 6, 5)\}, \{(3, 1, 3)\}$, it can also alternatively return a property that is not an actual $\mathcal{O}$ -consequence—e.g., $\varphi'_2(y, a, M) := y < a$ with a negative example $e_2^- = (6, 1, 5)$. The formula φ'_2 is not a $\mathcal{O}$ -consequence of Ψ as it rejects the positive example $(1, -4, 5)$.

When checking if an $\mathcal{L}$ -implicant φ is a weakest one, we can perform a dual check and ask if there does not exist an $\mathcal{L}$ -consequence that can accept one more example than the current formula. That is, $\text{CHECKWEAKEST}(\varphi, \varphi_{\vee}, \Psi, E^+, E^-)$ checks whether there do not exist an $\mathcal{L}$ -consequence φ' and a positive example $e \in \llbracket \Psi \rrbracket$ satisfying $\varphi_{\vee}$ such that: (i) φ' accepts all the positive examples in E^+ ; (ii) φ' accepts e and all the positive examples in E^+ , whereas φ rejects e . This check can be logically stated as

$$\begin{aligned} \text{CHECKWEAKEST}(\varphi, \varphi_{\vee}, \Psi, E^+, E^-) = \\ \neg \exists \varphi', e. \Psi(e) \wedge \neg \varphi_{\vee}(e) \wedge \neg \varphi(e) \wedge \varphi'(e) \wedge \varphi'(E^+) \wedge \neg \varphi'(E^-) \end{aligned} \quad (5.13)$$

Example 17 (CHECKWEAKEST). Consider again the query $\Psi := \exists x. y = ax \bmod M$, and a $\mathcal{U}$ -implicant $\varphi(y, a, M) := y = 0 \wedge a = 0$, which is not a weakest $\mathcal{U}$ -implicant. CHECKWEAKEST

$(\varphi, \perp, \Psi, \{(0,0,5)\}, \{(3,2,6)\})$ can return a strictly weaker $\mathcal{U}$ -implicant $\varphi'_1(y, a, M) := y = 0$ with a positive example $e_1^+ = (0, 1, 5)$. However, for the same reasons outlined in Example 16, the returned property may not be a $\mathcal{U}$ -implicant.

The CHECKSTRONGEST and CHECKWEAKEST procedures are effectively solving a synthesis problem—i.e., they are looking for a formula—and implementing them requires a form of example-based synthesis. Again, the presence of quantifiers in the negation of the query $\neg\Psi$ for CHECKSTRONGEST can result in constraint (5.12) containing alternating quantifiers, thus bringing us outside of the capability of many program synthesizers. We discuss a practical procedure for sidestepping the quantifier-alternation problem and performing this check in Section 5.5.

5.4.2 Synthesizing One Strongest $\mathcal{L}$ -Consequence and One Weakest $\mathcal{L}$ -Implicant

Algorithm 10: SYNTHSTRONGESTCONSEQUENCE($\Psi, \varphi_\wedge, \varphi_{init}, E^+, E^-$)

```

1  $\varphi, \varphi_{last} \leftarrow \varphi_{init}; E_{may}^- \leftarrow \emptyset$ 
2 while true do
3    $e^+ \leftarrow \text{CHECKIMPLICATION}(\Psi, \varphi)$ 
4   if  $e^+ \neq \top$  then
5      $E^+ \leftarrow E^+ \cup \{e^+\}$ 
6      $\varphi' \leftarrow \text{SYNTHESIZE}(E^+, E^- \cup E_{may}^-)$ 
7     if  $\varphi' \neq \perp$  then
8        $\varphi \leftarrow \varphi'$ 
9     else
10       $\varphi \leftarrow \varphi_{last}; E_{may}^- \leftarrow \emptyset$ 
11   else
12      $E^- \leftarrow E^- \cup E_{may}^-; E_{may}^- \leftarrow \emptyset$ 
13      $\varphi_{last} \leftarrow \varphi$ 
14      $e^-, \varphi' \leftarrow \text{CHECKSTRONGEST}(\varphi, \varphi_\wedge, \Psi, E^+, E^-)$ 
15     if  $e^- \neq \top$  then
16        $E_{may}^- \leftarrow \{e^-\}$ 
17        $\varphi \leftarrow \varphi'$ 
18     else
19       return  $\varphi, E^+, E^-$ 

```

We are now ready to describe our main procedures: `SYNTHSTRONGESTCONSEQUENCE` (Algorithm 10) and `SYNTHWEAKESTIMPLICANT` (Algorithm 11). We first recall the description of `SYNTHPROPERTY` in chapter 4, the algorithm that synthesizes a strongest $\mathcal{L}$ -consequence that is incomparable with the $\mathcal{L}$ -consequences we already synthesized (the algorithm will be used in Section 5.4.3 to synthesize one $\mathcal{L}$ -consequence at a time). We then describe one of the contributions of this chapter, i.e., how the algorithm changes for its under-approximated dual `SYNTHWEAKESTIMPLICANT`.

Synthesizing One Strongest $\mathcal{L}$ -Consequence

Given a query formula Ψ and a conjunction of $\mathcal{L}$ -consequences we have already synthesized $\varphi_\wedge$, the procedure `SYNTHSTRONGESTCONSEQUENCE` synthesizes a strongest $\mathcal{L}$ -consequence φ for the query Ψ that is incomparable to the already synthesized formulas in $\varphi_\wedge$. We say an $\mathcal{L}$ -consequence φ for the query Ψ is strongest *with respect to* $\varphi_\wedge$ if there does not exist an $\mathcal{L}$ -consequence φ' for Ψ such that $\varphi' \wedge \varphi_\wedge$ is strictly stronger than $\varphi \wedge \varphi_\wedge$ —i.e., the $\mathcal{L}$ -consequence φ is incomparable to all the $\mathcal{L}$ -consequences in $\varphi_\wedge$.

In each iteration, `SYNTHSTRONGESTCONSEQUENCE` performs two steps. First, it uses `CHECKIMPLICATION` to check whether the current candidate φ is a consequence of Ψ (line 3). Second, if the candidate φ is a consequence of Ψ , it uses `CHECKSTRONGEST` to check whether φ is strongest with respect to $\varphi_\wedge$ (line 14). The algorithm terminates once a formula passes both checks (line 19).

If the current candidate φ is not a consequence of Ψ , `CHECKIMPLICATION` returns a positive example e^+ (line 3). The algorithm then adds e^+ to the set of positive examples E^+ and uses it to `SYNTHESIZE` a new candidate $\mathcal{L}$ -consequence (lines 5 and 6).

If the current candidate φ is a consequence of Ψ but there is an $\mathcal{L}$ -consequence φ' that aligns with the current set of positive and negative examples, E^+ and E^- , and can reject one more negative example e^- , `CHECKSTRONGEST` returns this property φ' along with the negative example e^- . The example e^- is then temporarily stored in E_{may}^- without immediately updating

Table 5.1. Algorithm 10 run table

Iter.	φ	E^+	E^-	E_{may}^-
$n-3$	$y \neq M$	$\{e_1^+, e_2^+, e_3^+\}$	$\{e_1^-\}$	$\emptyset$
$n-1$	$y \neq M$	$\{e_1^+, e_2^+, e_3^+, e_4^+\}$	$\{e_1^-\}$	$\emptyset$
n	$y < M$	$\{e_1^+, e_2^+, e_3^+, e_4^+\}$	$\{e_1^-\}$	$\{e_3^-\}$

E^- (line 16). Updating E^- is delayed because φ' may not be a consequence of the query Ψ (Example 16), and in the worst case, there might not exist an $\mathcal{L}$ -consequence that rejects e^- (Example 12). The example stored in E_{may}^- can be safely added to E^- when CHECKIMPLICATION verifies that $\mathcal{L}$ -consequence φ returned by CHECKSTRONGEST is indeed a consequence of the query Ψ (line 12); at this point we are certain that the example in E_{may}^- can be rejected by at least one $\mathcal{L}$ -consequence, as witnessed by φ .²

The candidate $\mathcal{L}$ -consequence returned by CHECKSTRONGEST in line 14 is only guaranteed to be consistent with the examples, and therefore must be checked again by CHECKIMPLICATION in line 3. If the candidate fails to pass CHECKIMPLICATION, the algorithm keeps adding more positive examples until either (i) it finds a $\mathcal{L}$ -consequence that rejects all the negative examples in $E^- \cup E_{may}^-$, or (ii) SYNTHESIZE in line 6 fails to find an $\mathcal{L}$ -consequence. In the latter case, SYNTHSTRONGESTCONSEQUENCE concludes that the example in E_{may}^- cannot be rejected by any $\mathcal{L}$ -consequence and thus restarts after discarding the example in E_{may}^- (line 10). For efficiency, whenever E^- is updated in line 12, the algorithm stores the current $\mathcal{L}$ -consequence φ that rejects all the negative examples in E^- in a variable φ_{last} (line 13). In this way, the algorithm can revert to φ_{last} when E_{may}^- is discarded (line 10).

Example 18 (Algorithm 10 Run). *Consider the query $\Psi := \exists x. y = ax \bmod M$. The Tab. 5.1 shows the last 4 iterations in a possible execution of SYNTHSTRONGESTCONSEQUENCE $(\Psi, \top, \top, \emptyset, \emptyset)$. Specifically, it shows the value of φ , E^+ , and E^- at the start of each iteration (Line 2). In the table, $e_1^+ = (y, a, M) = (1, 6, 5)$, $e_2^+ = (1, 1, 5)$, $e_3^+ = (1, -4, 5)$, $e_4^+ = (6, 2, 8)$,*

²Delaying the update of negative examples enables treating all negative examples in E^- as hard constraints. This is a key innovation by Park et al. [116] over prior work [81].

$e_1^- = (8, 1, 8)$, $e_2^- = (3, 2, 6)$, $e_3^- = (6, 2, 5)$.

Iteration $n - 3$. So far the algorithm has computed an $\mathcal{O}$ -consequence $y \neq M$ that is not a strongest one. Therefore, CHECKIMPLICATION passes but CHECKSTRONGEST fails, and the execution reaches Line 16. Then φ is set to a new candidate $isPrime(M)$, and a negative example e_2^- is added to E_{may}^- .

Iteration $n - 2$. The property $isPrime(M)$ is not an $\mathcal{O}$ -consequence, so CHECKIMPLICATION fails. The execution reaches Line 5, and a positive example e_4^+ is added to E^+ . As discussed in Example 14, e_2^- cannot be rejected by any $\mathcal{O}$ -consequence, so SYNTHESIZE fails, φ is reverted to $y \neq M$, and E_{may}^- is cleared.

Iteration $n - 1$. Similar to iteration $n - 3$ but with a new positive example e_4^+ added, CHECKSTRONGEST returns a new candidate $y < M$ and a new negative example e_3^- , which is added to E_{may}^- .

Iteration n . The property $y < M$ is indeed a strongest $\mathcal{O}$ -consequence, so it passes both the CHECKIMPLICATION and CHECKSTRONGEST and is finally returned.

Once SYNTHSTRONGESTCONSEQUENCE terminates, it always returns a strongest $\mathcal{L}$ -consequence for Ψ (with respect to $\varphi_\wedge$) that accepts all the examples in E^+ and rejects all the examples in E^- . Informally, if CHECKIMPLICATION returns $\top$, then the property is an $\mathcal{L}$ -consequence, and if CHECKSTRONGEST returns $\top$, then the property is a strongest property.

SYNTHSTRONGESTCONJUNCTION is also guaranteed to terminate for a finite DSL, when every call to primitives SYNTHESIZE, CHECKIMPLICATION and CHECKSTRONGEST terminates. Informally, CHECKIMPLICATION at Line 3 can only return a counterexample (i.e., φ is not a consequence) finitely many times, and CHECKSTRONGEST at Line 14 can only return a counterexample (i.e., φ is a consequence but not a strongest one) finitely many times between successive instances where CHECKIMPLICATION returns $\top$.

Algorithm 11: SYNTHWEAKESTIMPLICANT ($\Psi, \varphi_V, \varphi_{init}, E^+, E^-$)

```

1  $\varphi, \varphi_{last} \leftarrow \varphi_{init}; E_{may}^+ \leftarrow \emptyset$ 
2 while true do
3    $e^- \leftarrow \text{CHECKIMPLICATION}(\varphi, \Psi)$ 
4   if  $e^- \neq \top$  then
5      $E^- \leftarrow E^- \cup \{e^-\}$ 
6      $\varphi' \leftarrow \text{SYNTHESIZE}(E^+ \cup E_{may}^+, E^-)$ 
7     if  $\varphi' \neq \perp$  then
8        $\varphi \leftarrow \varphi'$ 
9     else
10       $\varphi \leftarrow \varphi_{last}; E_{may}^+ \leftarrow \emptyset$ 
11  else
12     $E^+ \leftarrow E^+ \cup E_{may}^+; E_{may}^+ \leftarrow \emptyset$ 
13     $\varphi_{last} \leftarrow \varphi$ 
14     $e^+, \varphi' \leftarrow \text{CHECKWEAKEST}(\varphi, \varphi_V, \Psi, E^+, E^-)$ 
15    if  $e^+ \neq \top$  then
16       $E_{may}^+ \leftarrow \{e^+\}$ 
17       $\varphi \leftarrow \varphi'$ 
18    else
19      return  $\varphi, E^+, E^-$ 

```

Synthesizing One Weakest $\mathcal{L}$ -Implicant

Given a query formula Ψ , a disjunction of $\mathcal{L}$ -implicants we already synthesized φ_V , the goal of the procedure SYNTHWEAKESTIMPLICANT is to synthesize a weakest $\mathcal{L}$ -implicant φ for the query Ψ that is incomparable to the already synthesized formulas in φ_V .

We say an $\mathcal{L}$ -implicant φ for the query Ψ is weakest *with respect to* φ_V if there does not exist an $\mathcal{L}$ -implicant φ' for Ψ such that $\varphi' \vee \varphi_V$ is strictly weaker than $\varphi \vee \varphi_V$ —i.e., the $\mathcal{L}$ -implicant φ is incomparable to the already synthesized $\mathcal{L}$ -implicants.

SYNTHWEAKESTIMPLICANT solves the dual of the problem solved by the procedure SYNTHSTRONGESTCONSEQUENCE, and the two algorithms share the same structure. Due to the duality (i) the roles of positive and negative examples are inverted; (ii) CHECKIMPLICATION in line 3 checks whether φ is an *implicant* of Ψ , instead of checking that φ is a *consequence* of Ψ ; and (iii) precision is checked by CHECKWEAKEST instead of CHECKSTRONGEST. These

changes are highlighted in Algorithm 11.

5.4.3 Synthesizing a Best $\mathcal{L}$ -Conjunction and $\mathcal{L}$ -Disjunction

In this section we present detailed algorithms to synthesize a best $\mathcal{L}$ -conjunction (Algorithm 12) and $\mathcal{L}$ -disjunction (Algorithm 13).

Synthesizing a best $\mathcal{L}$ -conjunction

Algorithm 12: SYNTHSTRONGESTCONJUNCTION(Ψ)

```

1  $\varphi_{\wedge} \leftarrow \top$ 
2  $\Pi \leftarrow \emptyset$ 
3  $E^+ \leftarrow \emptyset$ 
4 while  $\top$  do
5    $\varphi, E^+, E^- \leftarrow \text{SYNTHSTRONGESTCONSEQUENCE}(\Psi, \varphi_{\wedge}, \top, E^+, \emptyset)$ 
6   if  $E^- = \emptyset$  then
7      $e^- \leftarrow \text{ISSAT}(\varphi_{\wedge} \wedge \neg \varphi)$ 
8     if  $e^- \neq \perp$  then
9        $E^- \leftarrow \{e^-\}$ 
10    else
11      return  $\Pi$ 
12   $\varphi, E^+, \_ \leftarrow \text{SYNTHSTRONGESTCONSEQUENCE}(\Psi, \top, \varphi, E^+, E^-)$ 
13   $\Pi \leftarrow \Pi \cup \{\varphi\}$ 
14   $\varphi_{\wedge} \leftarrow \varphi_{\wedge} \wedge \varphi$ 

```

The algorithm SYNTHSTRONGESTCONJUNCTION iteratively synthesizes incomparable strongest $\mathcal{L}$ -consequences. At each iteration, SYNTHSTRONGESTCONJUNCTION keeps track of the conjunction of synthesized strongest $\mathcal{L}$ -consequences $\varphi_{\wedge}$, along the set of positive examples E^+ that have been observed so far.

Each iteration SYNTHSTRONGESTCONJUNCTION calls SYNTHSTRONGESTCONSEQUENCE to try to synthesize a strongest $\mathcal{L}$ -consequence for Ψ with respect to $\varphi_{\wedge}$ (line 5). A property φ returned by SYNTHSTRONGESTCONSEQUENCE is checked whether it rejects some example that was not rejected by $\varphi_{\wedge}$ (lines 6 and 7).

If φ does not reject any example that was not already rejected by $\varphi_{\wedge}$, the formula $\varphi_{\wedge}$ is a

Algorithm 13: SYNTHWEAKESTDISJUNCTION(Ψ)

```

1  $\varphi_V \leftarrow \perp$ 
2  $\Pi \leftarrow \emptyset$ 
3  $E^- \leftarrow \emptyset$ 
4 while  $\top$  do
5    $\varphi, E^+, E^- \leftarrow \text{SYNTHWEAKESTIMPLICANT}(\Psi, \varphi_V, \perp, \emptyset, E^-)$ 
6   if  $E^+ = \emptyset$  then
7      $e^+ \leftarrow \text{ISSAT}(\neg\varphi_V \wedge \varphi)$ 
8     if  $e^+ \neq \perp$  then
9        $E^+ \leftarrow \{e^+\}$ 
10    else
11      return  $\Pi$ 
12   $\varphi, \_, E^- \leftarrow \text{SYNTHWEAKESTIMPLICANT}(\Psi, \perp, \varphi, E^+, E^-)$ 
13   $\Pi \leftarrow \Pi \cup \{\varphi\}$ 
14   $\varphi_V \leftarrow \varphi_V \vee \varphi$ 

```

best $\mathcal{L}$ -conjunction, and thus SYNTHSTRONGESTCONJUNCTION returns the set of synthesized $\mathcal{L}$ -consequences (line 11).

If φ rejects some example that was not rejected by $\varphi_\wedge$, SYNTHSTRONGESTCONJUNCTION needs to further strengthen φ to a strongest $\mathcal{L}$ -consequence for Ψ with respect to examples that might already be rejected by $\varphi_\wedge$. Without this step the returned $\mathcal{L}$ -consequence may be imprecise for examples that were not considered by SYNTHSTRONGESTCONSEQUENCE because they were outside of $\varphi_\wedge$. To achieve this further strengthening, SYNTHSTRONGESTCONJUNCTION makes another call to SYNTHSTRONGESTCONSEQUENCE with the example sets E^+ and E^- returned by the previous call to SYNTHSTRONGESTCONSEQUENCE together with φ and $\varphi_{init} := \varphi$, but with $\varphi_\wedge := \top$ (line 12).

Synthesizing a best $\mathcal{L}$ -disjunction

Because SYNTHWEAKESTDISJUNCTION solves the dual problem of the one solved by SYNTHSTRONGESTCONJUNCTION, the two algorithms share the same structure. Due to the duality (i) the roles of positive and negative examples are inverted; (ii) $\top$ is replaced by $\perp$; and (iii) at each iteration a weakest $\mathcal{L}$ -implicant is synthesized by SYNTHWEAKESTIMPLICANT

instead of SYNTHSTRONGESTCONSEQUENCE. These changes are highlighted in *violet* in Algorithm 13.

Using the argument of chapter 4 for SYNTHSTRONGESTCONJUNCTION, assuming all the primitives always terminate, our algorithms satisfy the following soundness and completeness theorems. Specifically, the number of iterations in Algorithm 10 and 11 is bounded by the number of properties in the DSL $\mathcal{L}$ and the number of examples in the domain, whichever is smaller.

Theorem 5.4.2 (Soundness). *If SYNTHSTRONGESTCONJUNCTION terminates, $\varphi_{\wedge}$ is a best $\mathcal{L}$ -conjunction for Ψ . If SYNTHWEAKESTDISJUNCTION terminates, $\varphi_{\vee}$ is a best $\mathcal{L}$ -disjunction for Ψ .*

Theorem 5.4.3 (Relative Completeness). *Suppose that either $\mathcal{L}$ contains finitely many formulas, or the example domain is finite.*

If SYNTHESIZE, CHECKIMPLICATION and CHECKSTRONGEST are decidable on $\mathcal{L}$, then SYNTHSTRONGESTCONSEQUENCE and SYNTHSTRONGESTCONJUNCTION always terminate.

If SYNTHESIZE, CHECKIMPLICATION and CHECKWEAKEST are decidable on $\mathcal{L}$, then SYNTHWEAKESTIMPLICANT and SYNTHWEAKESTDISJUNCTION always terminate.

Proof. Because of the duality between $\mathcal{L}$ -consequences and $\mathcal{L}$ -implicants, we only prove for SYNTHSTRONGESTCONSEQUENCE and SYNTHSTRONGESTCONJUNCTION here (the other case is identical). We first prove the termination of SYNTHSTRONGESTCONSEQUENCE.

If the example domain is finite, there are only finitely many semantically inequivalent $\mathcal{L}$ -properties, so the case of finite example domain can be reduced to the case of finite DSL.

For the case of a finite DSL, the theorem can be proved by first proving CHECKIMPLICATION (Line 3) does not return $\top$ (i.e., φ is not a consequence) only finitely many times, and then proving CHECKSTRONGEST (Line 14) does not return $\top$ (i.e., φ is a consequence but not a strongest one) only finitely many times between successive instances where CHECKIMPLICATION returns $\top$.

Every time CHECKIMPLICATION does not return $\top$, it returns a positive example e^+ that is rejected by the current $\mathcal{L}$ -property φ . However, since e^+ is added to E^+ , the properties in the subsequent iterations must accept e^+ so that φ will not occur again as an argument of CHECKIMPLICATION. Therefore, we only meet finitely many properties that are not consequences.

We can now move to the case where an $\mathcal{L}$ -property φ reaches CHECKSTRONGEST at line 14. In this case, there are two possibilities: (i) CHECKSTRONGEST returns a property φ' that is not a consequence (and a negative example e_1^-); (ii) CHECKSTRONGEST returns a property φ' that is a consequence (and a negative example e_2^-). For case (i), CHECKIMPLICATION in the next iteration will not return $\top$. For case (ii), e_2^- will be merged to E^- so that the properties in the subsequent iterations must reject e_2^- , whereas φ accepts it. Consequently, φ will not occur again as an argument of CHECKSTRONGEST. Therefore, between two CHECKIMPLICATION that does not return $\top$, CHECKSTRONGEST can return non- $\top$ result only finitely many times.

Turning to SYNTHSTRONGESTCONJUNCTION, since it calls SYNTHPROPERTY to grow a set of incomparable $\mathcal{L}$ -consequences, the fact that each call on SYNTHPROPERTY terminates guarantees that SYNTHSTRONGESTCONJUNCTION always terminates. $\square$

Also, our procedure for solving problems expressed in the LOUD framework iteratively synthesizes strongest $\mathcal{L}$ -consequences (resp. $\mathcal{L}$ -implicants) that can strengthen (resp. weaken) the current $\mathcal{L}$ -conjunction (resp. $\mathcal{L}$ -disjunction) until no further strengthening (resp. weakening) is possible. Thanks to this iterative approach, even when the procedure does not terminate, one can output intermediate results, which are properties that are guaranteed to be strongest (resp. weakest), though they might not form a best $\mathcal{L}$ -conjunction (resp. $\mathcal{L}$ -disjunction).

5.5 Counterexample-Guided Quantifier Instantiation

In Section 5.4, when we discussed the main specification-synthesis loop, we assumed we were given implementations of all the needed primitives. In this section, we explain how each primitive can be implemented for existentially quantified queries.

CHECKSTRONGEST (line 14) in SYNTHSTRONGESTCONSEQUENCE and CHECKIMPLICATION (line 3) in SYNTHWEAKESTIMPLICANT check for the existence of a new negative example (along with additional constraints). However, when dealing with an existentially quantified query $\Psi := \exists h. \psi(v, h)$, a negative example e must be such that the formula $\neg\psi(e, h)$ is valid for *all* values of the existentially quantified variable h . Therefore, checking the existence of a negative example e requires solving a formula that has alternating quantifiers. To handle these primitives involving quantifier alternation, we propose a CounterExample-Guided Quantifier Instantiation (CEGQI) algorithm similar to the one by Reynolds et al. [130], which can implement the primitives that require finding negative counterexamples using only existentially-quantified formulas.

Counterexample-Guided Quantifier Instantiation for Weakest $\mathcal{L}$ -implicant

We start with the simpler of the two queries, CHECKIMPLICATION in SYNTHWEAKESTIMPLICANT (line 3), which requires solving a formula with alternating quantifiers of the following form (by negating formulas in §5.4.1):

$$\exists e. \forall h. \neg\psi(e, h) \wedge \varphi(e) \quad (5.14)$$

The CEGQI algorithm for solving Equation (5.14) iteratively builds a set H of possible values for h and finds a value of e that is consistent with the finite set of values H . The set H is updated by repeating the following two operations until a solution that holds for all values of h is found.

Generating Candidate Negative Example

Given formulae φ , ψ , and a finite set H of values the existentially-quantified variable h can take, $\text{GENCANDIDATENEGEX}(\varphi, \psi, H)$ generates an example $e \in \llbracket \varphi \rrbracket$ such that the formula $\psi(e, h)$ does not hold for all the values h in the set H , if such an example e exists. If no such

example exists, $\text{GENCANDIDATENEGEX}(\varphi, \psi, H)$ returns $\perp$. Formally:

$$\text{GENCANDIDATENEGEX}(\varphi, \psi, H) = \exists e. \bigwedge_{h \in H} \neg \psi(e, h) \wedge \varphi(e) \quad (5.15)$$

Checking Candidate Negative Example

Given a formula ψ , and a candidate example e , the $\text{CHECKCANDIDATENEGEX}(\psi, e)$ checks if there does not exist a value for the existentially quantified variable h such that $\psi(e, h)$ holds true (i.e., whether there exists a value of h that makes the example actually positive); it returns $\top$ or the value of h if the check fails. Formally:

$$\text{CHECKCANDIDATENEGEX}(\psi, e) = \neg \exists h. \psi(e, h) \quad (5.16)$$

Counterexample-Guided Quantifier Instantiation

The CEGQI algorithm for CHECKIMPLICATION (Algorithm 14) iteratively generates candidate negative examples using GENCANDIDATENEGEX and checks whether they are actually negative using $\text{CHECKCANDIDATENEGEX}$. Across iterations, it maintains the set of values of h returned by $\text{CHECKCANDIDATENEGEX}$ in H , and uses GENCANDIDATENEGEX to find an example e that behaves well for all the values in H discovered so far—i.e., e satisfies $\neg \psi(e, H) \wedge \varphi(e)$ (line 4).

If GENCANDIDATENEGEX fails to find an example, it means that there is no example e satisfying $\neg \psi(e, H) \wedge \varphi(e)$, thereby a stronger condition in Equation (5.14) also cannot be satisfied. Therefore, CEGQI returns $\top$ (line 6)—i.e., there does not exist a valid negative example.

If GENCANDIDATENEGEX returns an example e , the example is tested by the primitive $\text{CHECKCANDIDATENEGEX}$ to check whether $\psi(e, h)$ does not hold for every possible value of h and not only for values found so far (line 8). The algorithm returns the example e once it passes the check (line 10), but if it fails the check, a new counterexample h returned by $\text{CHECKCANDIDATENEGEX}$ is added to the set H , and the algorithm restarts at line 4. Note that

Table 5.2. Algorithm 14 run table

Iter.	e			h
	y	a	M	x
1	2	2	4	1
2	3	1	4	3
3	0	2	4	2
4	0	3	4	0
5	3	2	4	$\top$

the set of instances H can be cached and reused across different calls to CEGQI.

Example 19 (Algorithm 14 Run). *Consider again the query $\Psi := \exists x. y = ax \bmod M$ and the formula $\varphi := 0 \leq y < M$. The Tab. 5.2 shows a possible execution of how CHECKIMPLICATION $(\varphi, \Psi) := \forall e. \varphi(e) \Rightarrow \exists h. \psi(e, h)$ can find a negative example and prove that φ is not a $\mathcal{U}$ -implicant. Specifically, it shows the values of $e = (y, a, M)$ and $h = x$ at the end of each iteration (Algorithm 14) within Algorithm 14. For each iteration, $\varphi(e)$ is always true for $e = (y, a, M)$, while $\psi(e, h)$ is false for all previous $h = x$ but true for h in the current iteration.*

Algorithm 14: CHECKIMPLICATION(ϕ, Ψ)

```
1 assume  $\Psi = \exists h. \psi(v, h)$ 
2  $H \leftarrow \emptyset$ 
3 while  $\top$  do
4    $e \leftarrow \text{GENCANDIDATENEGEX}(\phi, \psi, H)$ 
5   if  $e = \perp$  then
6     return  $\top$ 
7   else
8      $h \leftarrow \text{CHECKCANDIDATENEGEX}(\psi, e)$ 
9     if  $h = \top$  then
10      return  $e$ 
11     else
12       $H \leftarrow H \cup \{h\}$ 
```

Counterexample-Guided Quantifier Instantiation for Strongest $\mathcal{L}$ -consequence

The call to CHECKSTRONGEST in SYNTHSTRONGESTCONSEQUENCE (line 14) requires solving a formula that has alternating quantifiers and the following form (by negating appropriate formulas in §5.4.1):

$$\exists e, \phi'. \forall h. \neg \psi(e, h) \wedge \phi_{\wedge}(e) \wedge \phi(e) \wedge \neg \phi'(e) \wedge \phi'(E^+) \wedge \neg \phi'(E^-) \quad (5.17)$$

This formula looks more complicated due to the presence of the existential variable ϕ' . However, a similar CEGQI approach to the one presented in Section 5.5 can also be used to solve Equation (5.17), by finding a negative example and a formula in tandem.

The only change in the CEGQI algorithm for solving CHECKSTRONGEST (Algorithm 15) is that GENCANDIDATENEGEX is replaced by a new operation, GENCANDIDATESPECNEGEX,

defined as follows. Given formulae $\varphi, \psi, \varphi_\wedge$, set of examples E^+, E^- , and a finite set H of values the existentially-quantified variable h can take, the primitive $\text{GENCANDIDATESPECNEGEX}(\varphi, \psi, \varphi_\wedge, E^+, E^-, H)$ generates an $\mathcal{L}$ -property φ' and an example e satisfying $\varphi_\wedge$ such that (i) the formula $\psi(e, h)$ does not hold for all the values h in the set H (ii) φ' accepts all the positive examples in E^+ ; (iii) φ' rejects e and all the negative examples in E^- , whereas φ accepts e , if such an example e and formula φ' exist. If no such example exists, then $\text{GENCANDIDATESPECNEGEX}(\varphi, \psi, \varphi_\wedge, E^+, E^-, H)$ returns $\perp$. Stated formally:

$$\begin{aligned} \text{GENCANDIDATESPECNEGEX}(\varphi, \psi, \varphi_\wedge, E^+, E^-, H) = \\ \exists e, \varphi'. \bigwedge_{h \in H} \neg \psi(e, h) \wedge \varphi_\wedge(e) \wedge \varphi(e) \wedge \neg \varphi'(e) \wedge \varphi'(E^+) \wedge \neg \varphi'(E^-) \end{aligned} \quad (5.18)$$

Because the variable h only appears in the constraint $\neg \psi(e, h)$, whether e is indeed a negative example can still be tested using $\text{CHECKCANDIDATENEGEX}$ (5.16).

Similar to the Algorithm 14, if GENCANDIDATENEGEX fails to find an example, it means that there is no example e satisfying Equation (5.18), thereby a stronger condition in Equation (5.17) also cannot be satisfied. The example is only returned after it has been tested by $\text{CHECKCANDIDATENEGEX}$ to ensure that $\psi(e, h)$ does not hold for every possible value of h .

Algorithm 15: CHECKSTRONGEST($\varphi, \varphi_{\wedge}, \Psi, E^+, E^-$)

```
1  assume  $\Psi = \exists h. \psi(v, h)$ 
2   $H \leftarrow \emptyset$ 
3  while  $\top$  do
4       $e, \varphi' \leftarrow \text{GENCANDIDATESPECNEGEX}(\varphi, \psi, \varphi_{\wedge}, E^+, E^-, H)$ 
5      if  $e = \perp$  then
6          return  $\top$ 
7      else
8           $h \leftarrow \text{CHECKCANDIDATENEGEX}(\psi, e)$ 
9          if  $h = \top$  then
10             return  $e, \varphi'$ 
11          else
12              $H \leftarrow H \cup \{h\}$ 
```

Correctness

The above observations are summarized as the following soundness theorem.

Theorem 5.5.1 (Soundness of CEGQI). (i) If CHECKIMPLICATION terminates with an example e , the example e is a valid solution to the existential quantifier in Equation (5.14). If CHECKIMPLICATION terminates with $\top$, there is no example e that satisfies Equation (5.14). (ii) If CHECKSTRONGEST terminates with an example e and φ' , the example e and φ' are valid solution to the existential quantifier in Equation (5.17). If CHECKSTRONGEST terminates with $\top$, there is no example e and φ' that satisfy Equation (5.17).

Because Algorithm 14 and 15 monotonically increases the size of the set H , as long as the domain of one of the variables e and h is finite, both algorithms always terminate.

Theorem 5.5.2 (Completeness of CEGQI). *Suppose at least one of the domains of the variables e or h is finite. If `GENCANDIDATENEGEX` and `CHECKCANDIDATENEGEX` are decidable for ψ and ϕ , then `CHECKIMPLICATION` always terminates. If `GENCANDIDATESPECNEGEX` and `CHECKCANDIDATENEGEX` are decidable for ψ and ϕ , then `CHECKSTRONGEST` always terminates.*

Therefore, when the domain is finite, the specification synthesis for an existentially quantified query can be solved using only calls with the quantifier free part of the query.

Note that, in the worst case, CEGQI can enumerate the entire domain of h . As we demonstrate in our evaluation, this exhaustive enumeration (which is common for CEG-style algorithms [144]) is practically rare and a small number of examples are usually sufficient to solve the problem.

5.6 Implementation

We implemented our algorithms for solving synthesis problems in the LOUD framework in a tool called ASPIRE. ASPIRE is implemented in Java, on top of the SKETCH program synthesizer (v.1.7.6) [145].

Following Section 5.2, ASPIRE takes the following four inputs: (i) A query Ψ for which ASPIRE is to find $\mathcal{L}$ -consequences or $\mathcal{L}$ -implicants where each variable in Ψ should be labeled either as free or existentially quantified. (ii) The context-free grammar of the DSL $\mathcal{L}$ in which properties are to be expressed. (iii) A piece of code in the SKETCH programming language that expresses the concrete semantics of the function symbols in Ψ and $\mathcal{L}$. (iv) The bounded domain of each variable in the query Ψ —i.e., each variable is assigned a range of possible input values.

Take the motivating problem in Section 5.1.1 as an example: input (i) corresponds to the blocks `Variables` and `Query` in Figure 5.1a, input (ii) corresponds to the block `Language` in Figure 5.1b, and input (iii) are SKETCH implementations of `modhash` and `isPrime`—i.e., simple imperative functions. Input (iv) is provided via the `Examples` block in Figure 5.3, which we will


```

Examples {
  int IG → ??(4) | -??(4)
    for a, y, x;
  int PosIG → ??(4) + 1;
    for M;
}

```

Figure 5.3. The bounded domains of variables in problem Section 5.1.1

discuss next.

From Grammars to SKETCH Generators

As synthesis needs to be performed over properties in the DSL $\mathcal{L}$, the context-free grammar for $\mathcal{L}$ is automatically translated to a SKETCH generator. A SKETCH generator is a construct that allows one to describe a recursively defined search space of programs. In a generator, one is allowed to use holes (denoted with $??$) to allow the synthesizer to make choices about what terms to output. In our setting, holes are used to select which production is expanded at each node in a recursively defined derivation tree.

ASPIRE also uses grammars, which in turn are translated to generators, to specify the values each variable in the query Ψ can assume. Figure 5.3 shows how the user specifies the bounded domain of each variable for the problem in Section 5.1.1. The nonterminal IG is translated to a generator that can produce an integer from $[-15, 15]$ (the notation $??(4)$ denotes a 4-bit hole), and is used to define the domain for variables a , y , and x . Similarly, the nonterminal PosIG defines the domain of the positive integer variable M to be the range $[1, 16]$. ASPIRE also supports inductive datatypes, e.g. a generator of lists of integers in the range $[-15, 15]$ can be defined as `[ list LG -> nil() | cons(IG, LG) ]`.

Synthesis Primitives in SKETCH

The primitives SYNTHESIZE, CHECKIMPLICATION in Algorithm 10, and CHECKWEAKEST in Algorithm 11 are implemented as calls to the SKETCH synthesizer. Typically, a SKETCH program contains 3 elements: (i) a harness procedure that defines what should be synthesized, (ii)

holes associated with a corresponding generator, and (iii) assertions. The harness procedure is the entry point of the SKETCH program, and together with the assertion it serves as a specification for what values the holes can assume to form a valid solution to the synthesis problem. Multiple harnesses in one SKETCH program are also allowed, where the goal of the SKETCH synthesizer is to find the same assignment to shared holes that make all assertions pass. For example, when encoding SYNTHESIZE, each example is implemented as a harness with assertions indicating that it should be positive or negative. Both CHECKSTRONGEST in SYNTHSTRONGESTCONSEQUENCE (Algorithm 10) and CHECKIMPLICATION in SYNTHWEAKESTIMPLICANT (Algorithm 11) are implemented using the CEGQI approach described in Section 5.5 (Algorithms 15 and 14, respectively). These algorithms are implemented as separate procedures where each call to GENCANDIDATENEGEX and CHECKCANDIDATENEGEX only has an existential quantifier, and can thus be implemented as a single call to the SKETCH synthesizer.

Bounds

SKETCH allows one to provide bounds for recursion and loops to make synthesis tractable. In ASPIRE, we need to consider two kinds of bounds.

First, one has to bound the depth of each recursive generator. Concretely, this bound means that ASPIRE can only support DSLs where the derivation trees have bounded height (ASPIRE allows one to specify the bound for each DSL). As recursive generators are used to produce examples for inductive datatypes—e.g. list—one also has to bound the height of such examples. Second, one has to bound how many times a loop can be unrolled/executed. We will discuss in Section 5.7 what benchmarks are in theory affected by these bounds. Additionally, these bounds limit DSLs and example domains to a finite size, ensuring that our algorithms in theory terminate (Theorem 5.4.3).

Timeout

We use a timeout of 20 minutes, after which ASPIRE returns the current $\mathcal{L}$ -consequences (or $\mathcal{L}$ -implicants). Although it might not be the strongest $\mathcal{L}$ -consequences (or weakest $\mathcal{L}$ -

implicants), each individual $\mathcal{L}$ -consequence (or $\mathcal{L}$ -implicant) is a strongest (or a weakest) one.

5.7 Evaluation

Our evaluation of ASPIRE consists of two parts. The first part consists of simple deterministic and nondeterministic programs (from [116, 27, 17]) for which we could manually inspect whether ASPIRE computed the correct properties (§5.7.1). The second part showcases the capabilities of ASPIRE and the flexibility of the LOUD framework through three case studies: forward/backward reasoning for incorrectness logic (§5.7.2), synthesizing properties of concurrent programs (§5.7.3), and solving two-player games (§5.7.4). The evaluation highlights that the programmable queries and customized DSLs in the LOUD framework enable it to express a wide range of problems. All our case studies involve underapproximation and/or existential quantifiers. For each case study, we describe how we model the problem in LOUD, how we collected the benchmarks, how we designed the DSLs, and we present an analysis of the running time and effectiveness of ASPIRE and of the quality of synthesized $\mathcal{L}$ -consequences and $\mathcal{L}$ -implicants.

We ran all experiments on an Apple M1 8-core CPU with 8GB RAM. Each benchmark was run 3 times (timeout 20 minutes), with different random seeds for the SKETCH solver.³ All results in this section are for the median of three runs (by synthesis time).

5.7.1 Testing ASPIRE with Simple Programs

We conducted two experiments on simple programs for which we could manually check whether the synthesized properties were sound and strongest/weakest.

We provide a brief description of each experiment, but details about each problem, running times, concrete DSLs, and synthesized properties are discussed in Appendix B.

³Our synthesis primitives are nondeterministic—i.e., SKETCH can return *any* possible valid counterexample or property. The different random seeds will result in the SKETCH solver selecting different such examples/formulas. ASPIRE can therefore generate different sets of properties with different seeds, but as stated in Theorem 5.2.5, all best $\mathcal{L}$ -conjunctions (or $\mathcal{L}$ -disjunctions) are semantically equivalent.

Test Set I: Mining Under-Approximation Specifications

We used ASPIRE to compute $\mathcal{U}$ -implicants for the 35 programs from chapter 4 that are computed $\mathcal{O}$ -consequences using SPYRO. These programs include integer functions, data structure manipulations, and small imperative programs. To get the dual DSL $\mathcal{U}$ of each DSL $\mathcal{O}$ used in the original benchmarks, we replaced every top-level production of the form $S \rightarrow \vee[AP, 0..n]$ with a production $S \rightarrow \wedge[AP, 0..n]$. Note that the queries for all the SPYRO benchmarks *do not* contain existential quantifiers.

ASPIRE could synthesize properties for 35/35 benchmarks, and guaranteed that all of them were best $\mathcal{U}$ -implicants (i.e., CHECKWEAKEST succeeded). Each benchmark takes at most 7 minutes (the largest grammar contained $1.48 \cdot 10^{13}$ properties). Because the DSL was originally designed for overapproximations, the synthesized underapproximation properties were often trivial. For example, for the list reverse function, ASPIRE only tells us any singleton list can be obtained as output when providing the same one as input, but provides no properties describing the behavior for lists of lengths greater than 1. While the synthesized properties were not informative, this simple benchmark set allowed us to test ASPIRE’s ability to synthesize $\mathcal{U}$ -implicants.

Test set II: Nondeterministic programs

To test ASPIRE on problems involving existential quantifiers, we designed 15 simple imperative programs where nondeterministic values serve as operands or array indices—e.g., a nondeterministic sorting algorithm. To model the sequence of nondeterministic choices taken by a program, we use an existentially quantified array of values h in the query. Whenever the program execution reaches a nondeterministic command, the command takes the next value of the array h .

ASPIRE synthesizes both $\mathcal{O}$ -consequences and $\mathcal{U}$ -implicants for 14/15 benchmarks, taking less than 400 seconds for each benchmark and guaranteeing that the synthesized properties are best ones. All benchmarks that terminated for one run terminated for all 3 runs, with fastest

$$\begin{aligned}
\text{I1: } & y = 0 & \text{I2: } & -M \leq a \wedge a < M \wedge a = y \\
\text{I3: } & -M \leq y \wedge y < M \wedge -M < a \wedge a < M \wedge a \neq 0 \wedge \text{isPrime}(M)
\end{aligned}$$

Figure 5.4. Synthesized $\mathcal{U}$ -implicants

and slowest runs differing by at most $2x$. The synthesized properties provide intuition for both the demonic and angelic behaviors of these programs. For example, consider a program that nondeterministically swaps elements of an array of length 4 to sort it: ASPIRE can tell us which initial arrays *may* be sorted within n swaps (for a given n), and also identifies what kind of arrays will never be sorted in less than n swaps. The one timeout benchmark models a merge sort that computes the number of inverse pairs in an array of nondeterministic values; ASPIRE fails due to the complexity of the program, which involves nested recursions and loops.

5.7.2 Application 1: Incorrectness Reasoning

Thanks to the support for both over- and under-approximation, some forms of forward/backward reasoning for both Hoare logic [70] and incorrectness logic [110] can be captured in the LOUD framework. Because there has been a lot of research and there are many tools on precondition/postcondition inference of Hoare triples, we only discuss the relation between the LOUD framework and incorrectness logic in this subsection, along with an evaluation.

```
int remhash (int a, M, x) { return a * x % M; }
```

Let's say we are interested in reasoning about the possible behaviors of the (incorrect) implementation of a modular hash function `remhash` shown above, where `%` is the remainder operator (instead of the modulus). The `%` operator is often misused when implementing a modular operation, as $a\%b$ may yield a negative output when either a or b is negative.

A summary of the behaviors of the `remhash` function can be identified by under-approximating the query $\Psi_{rem} := (\exists x. y = ax \% M)$. From the perspective of incorrectness logic, under-approximating Ψ_{rem} corresponds to performing forward reasoning to find results for y when no presumption on x is given—i.e., the presumption $P(x)$ is $\top$. For capturing under-

<pre> Variables { int a, M, x; exist int y; } Query { y = remhash(a, M, x); y < 0; } </pre> (a) Ψ_{wpp}	<pre> Language { C → /\[AP,0..6]; AP → I {≤ < = ≠} I isPrime(M) !isPrime(M) I → 0 a x M -M } </pre> (b) $\mathcal{U}_{wpp}$	<pre> I1: -M < x /\ x < 0 /\ 0 < a /\ a < M /\ isPrime(M) I2: 0 < x /\ x < M /\ -M < a /\ a < 0 /\ isPrime(M) </pre> (c) $\mathcal{U}_{wpp}$-implicants
---	--	---

Figure 5.5. (a) A query Ψ_{wpp} that allows identifying *weakest possible preconditions* that cause a bug in the `remhash` function (i.e., returning a negative number). (b) A DSL $\mathcal{U}_{wpp}$ for expressing weakest possible preconditions. (c) $\mathcal{U}_{wpp}$ -implicants synthesized by ASPIRE.

approximations of the query Ψ_{rem} , we reuse the DSL $\mathcal{U}$ from the example in Section 5.1.2. A mutually incomparable set of weakest $\mathcal{U}$ -implicants for query Ψ_{rem} is shown in Figure 5.4, which shows that `remhash` can indeed yield negative values for some choice of parameters a and M , as evidenced by the occurrence of a state in both the second and last formulae where y is negative. In other words, we recognize that some choices of input value can result in incorrect outcomes—i.e., negative numbers—but we do not know which ones.

Backward Reasoning: Weakest Possible Precondition

As described in Sec. 5.3.2, backward predicate transformers for incorrectness logic do not always exist because valid presumptions may not exist. When considering the `remhash` function, if Q encodes that the output is negative, $wpp(s, Q)$ will show which input possibly leads to a negative output. Formally, $wpp(s, Q)$ is the weakest P satisfying

$$\forall \sigma. P(\sigma) \Rightarrow [\exists \sigma'. Q(\sigma') \wedge \llbracket s \rrbracket(\sigma, \sigma')] \quad (5.19)$$

We use $wpp_{\mathcal{L}}(s, Q)$ to denote the weakest possible precondition expressible as a *disjunction* of predicates in the DSL $\mathcal{L}$. From Equation (5.19), $wpp_{\mathcal{L}}(s, Q)$ can be obtained by synthesizing weakest $\mathcal{L}$ -implicants of the query $\exists \sigma'. Q(\sigma') \wedge \llbracket s \rrbracket(\sigma, \sigma')$.

For the `remhash` function, we have shown the existence of a bug through forward reasoning—i.e., the output y can be negative. Now we want to compute the weakest possible

precondition (expressible in a DSL) that leads to the error state $Q(y) := y < 0$. Looking at Equation (5.19), we can spell out that a weakest possible precondition of $Q(y)$ for `remhash` is a weakest implicant of the formula $\exists y. [y = ax \% M \wedge Q(y)]$; one can provide to LOUD the corresponding query Ψ_{wpp} in Figure 5.5a.

To capture implicants of the query Ψ_{wpp} , we define the DSL $\mathcal{U}_{wpp}$ in Figure 5.5b by substituting every occurrence of y in $\mathcal{U}$ with x . An incomparable set of weakest $\mathcal{U}_{wpp}$ -implicants for query Ψ_{wpp} is shown in Figure 5.5c, where each formula states sufficient conditions under which $ax \% M$ produces a negative output—i.e., when either a or x falls within the interval $(-M, 0)$ and the other falls within the interval $(0, M)$.

Evaluation on Incorrectness Reasoning

We collected a total of 14 benchmarks: (i) the 2 example problems `remwupo` and `remwpp` from Section 5.3.2. (ii) 6 simple illustrative examples from the incorrectness logic paper [110], and (iii) 6 more complicated problems we crafted to illustrate how ASPIRE’s handling of incorrectness reasoning differs from incorrectness logic. Among these benchmarks, 7 are about $\mathcal{L}$ -weakest under-approximate postcondition and the other 7 are about $\mathcal{L}$ -weakest possible precondition. It takes ASPIRE less than 4 seconds to solve each benchmark from [110] and less than 50 seconds to solve each benchmark we crafted. Evaluation details are shown in Table 5.3.

Analysis of Benchmarks from [110]

We collected all the 3 triples $[P]s[Q]$ from the examples used in [110] where s is a nondeterministic program. However, in these triples, there was no guarantee that Q was the weakest under-approximate postcondition of P , or P was the weakest possible precondition of Q . We used ASPIRE to synthesize $wupo_{\mathcal{L}}(s, P)$ and $wpp_{\mathcal{L}}(s, Q)$ (the DSL $\mathcal{L}$ contained the same primitives appearing in the examples in [110]), thus $3+3=6$ benchmarks. We examined that each of 3 synthesized $wupo_{\mathcal{L}}(s, P)$ by ASPIRE was indeed a subset of $wupo(s, P)$, and for 2 cases the two were equal. For the one that is not equal, $wupo(s, P)$ is the set of all perfect squares numbers, whereas $wupo_{\mathcal{L}}(s, P)$ is the perfect squares numbers lower than a bound (this difference was

Table 5.3. Applications 1 to 3. LoC is the number of lines of SKETCH code used to write the semantics of programs and operators. $|\exists|$ is the size of the domain of the existentially quantified variables. #P and T(s) are the number of properties and synthesis time for both $\mathcal{L}$ -consequences and $\mathcal{L}$ -implicants. Incorrectness reasoning does not require synthesizing $\mathcal{L}$ -consequences.

	Problem	LoC	$ \exists $	$\mathcal{L}$ -cons.		$\mathcal{L}$ -impl.	
				#P	T(s)	#P	T(s)
Incorrectness	remwupo	10	32	/	/	3	27.05
	remwpp	14	32	/	/	2	26.27
	inc1wupo	13	256	/	/	3	1.10
	inc2wupo	28	64	/	/	9	2.15
	inc3wupo	16	2	/	/	2	0.41
	inc1wpp	12	256	/	/	2	0.42
	inc2wpp	33	4,096	/	/	7	6.01
	inc3wpp	21	2	/	/	1	0.15
	arit1wupo	8	32	/	/	2	4.03
	arit2wupo	8	64	/	/	3	2.96
	arit1wpp	8	4	/	/	2	3.76
	arit2wpp	8	8	/	/	5	62.52
	hashcoll	87	16	/	/	3	15.94
	coin	23	1,024	/	/	1	20.33
Concurrency	philo	79	$\sim 10^6$	4	11.85	3	6.15
	race1	82	64	1	0.71	3	0.85
	race2	86	1,024	1	2.08	3	2.06
	race3	88	4,096	1	18.79	5	4.69
	rsrc1	81	16	4	2.73	4	1.67
	rsrc2	85	256	4	5.08	4	5.37
	rsrc3	114	$\sim 10^6$	4	51.83	4	38.31
	rsrc4	96	$\sim 10^5$	6	145.61	6	81.41
	obdet	52	1,057	3	2.11	/	/
Game	num1	47	8	10	16.13	19	13.99
	num2	47	32	25	21.32	15	8.54
	rg	29	32	2	0.38	2	0.49
	nim2	59	$\sim 10^{22}$	2	99.23	4	22.19
	temp	34	120	10	125.24	10	74.65

$$\begin{array}{ll}
C \rightarrow /\backslash [AP, 0..5]; & AP \rightarrow N \{ \leq | < | = | \neq \} N \\
N \rightarrow N' \sqcup \bot - N' & N' \sqcup \rightarrow \sqcup 0 \sqcup \sqcup a \sqcup \sqcup b \sqcup \sqcup x
\end{array}$$

Figure 5.6. The DSL for `arit1`

due to our query limiting the sample space of each variable). The 3 $wpp_{\mathcal{L}}(s, Q)$ synthesized by ASPIRE are equal to $wpp(s, Q)$.

More Complex Benchmarks

The 6 more complex benchmarks for which we performed incorrectness are `arit1-wupo`, `arit1-wpp`, `arit2-wupo`, `arit2-wpp`, `hashcoll`, and `coin`.

The benchmarks `arit1` and `arit2` model two arithmetic functions “ $x' = \text{ite}(h_0, x, -x)$ ” and “ $x' = (h_1 + 1) \cdot x + h_2$ ”, where each $h_i \in \{0, 1\}$ is a nondeterministic value. For both cases, we set $a \leq x \leq b$ as a precondition P (or $a \leq x' \leq b$ as a postcondition Q) to synthesize $wpp_{\mathcal{L}}(s, Q)$ (or $wupo_{\mathcal{L}}(s, P)$), and thus get 4 benchmarks in total. To use ASPIRE, we need to mark x as existentially quantified variables when synthesizing $wupo_{\mathcal{L}}(s, P)$, whereas mark x' as existentially quantified variables when synthesizing $wpp_{\mathcal{L}}(s, P)$. Given a DSL containing basic arithmetic and comparison operators, ASPIRE synthesizes $wpp_{\mathcal{L}}(s, Q)$ and $wupo_{\mathcal{L}}(s, P)$ that are equal to $wpp(s, Q)$ and $wupo(s, P)$.

For example, to synthesize $wpp_{\mathcal{L}}(s, Q)$ for `arit1`, one can construct a query “ $\exists x', h_0. x' = \text{ite}(h_0, x, -x) \wedge a \leq x' \leq b$ ” and supply the DSL in Figure 5.6. ASPIRE will synthesize the $\mathcal{L}$ -implicants $\{-b \leq x, x \leq -a, a \leq x, x \leq b\}$.

We briefly summarize the findings on other benchmarks. The `coin` benchmark models the values one can produce using two coins that have co-prime denominations; ASPIRE can identify a lower bound above which all possible values can be produced using these coins. The `hashcoll` benchmark models a parametric hash function; ASPIRE can synthesize the condition that possibly causes a hash collision.

5.7.3 Application 2: Reasoning about Concurrent Programs

We show how ASPIRE can be used to reason about bugs in concurrent programs by considering 8 variants of 3 concurrency problems by Arpaci-Dusseau and Arpaci-Dusseau [8] (2 problems related to deadlocks, and 1 to race conditions), and one benchmark obdet that requires synthesizing hyperproperties. Similar to how we model nondeterminism, we introduce an array h to represent the order in which threads are scheduled.

In the *philo* benchmark, we show how ASPIRE can synthesize conditions under which deadlock can be reached or avoided for the dining-philosophers problem, where N processes arranged in a circle contend N resources that are shared by neighboring processes. A deadlock happens when no process can access both of their Left and Right resources indefinitely. ASPIRE models this problem with a query $\exists h. dl = \text{schedule}(o_1, \dots, o_N, h)$, where $o_i \in \{L, R\}$ indicates which resource the process i always takes first; dl denotes that a deadlock has happened.

For the case involving three processes/philosophers ($N = 3$), when given a DSL $\mathcal{O}$ in Figure 5.7a that contained predicates of the form $o_i = \{L|R\}$, ASPIRE synthesizes the $\mathcal{O}$ -consequences in Figure 5.7b, which state that deadlock can be prevented by having two of the processes disagree on their fork choice. For the same N , and a dual DSL $\mathcal{U}$ in Figure 5.7a, ASPIRE synthesizes the $\mathcal{U}$ -implicants in Figure 5.7c, which exactly characterize the two cases that lead to a deadlock (first two properties) and also capture that there always exists an execution that does not lead to a deadlock (last property).

Whereas similar tools for concurrent programs only deal with properties over a single schedule [155, 89], the next benchmark obdet demonstrates how ASPIRE can also synthesize properties that involve multiple schedules (i.e., hyperproperties). The obdet benchmark models two threads $p_o := p_i + s \mid p_o := p_i - s$ where s is a secret variable and p_o and p_i are public variables. We say that the system is observational deterministic [162] if the observations made by a public observer (i.e., one that can only observe p_o and p_i) are deterministic, regardless of scheduling orders and secret input data (i.e., the values of the

$O \rightarrow \wedge[AP, 0..5] \Rightarrow D$ $U \rightarrow \wedge[AP, 0..5] \wedge D$ $AP \rightarrow I = 'L' \mid I = 'R'$ $I \rightarrow o1 \mid \dots \mid oN$ $D \rightarrow d1 \mid !d1$	$C1: (o1 = L \wedge o2 = R) \Rightarrow !d1$ $C2: (o2 = L \wedge o3 = R) \Rightarrow !d1$ $C3: (o3 = L \wedge o1 = R) \Rightarrow !d1$	$I1: o1 = L \wedge o2 = L \wedge o3 = L \wedge d1$ $I2: o1 = R \wedge o2 = R \wedge o3 = R \wedge d1$ $I3: !d1$
(a) The DSLs $\mathcal{O}$ (rooted at O) and $\mathcal{U}$ (rooted at U)	(b) Synthesized $\mathcal{O}$ -consequences	(c) Synthesized $\mathcal{U}$ -implicants

Figure 5.7. The DSLs and synthesized properties for the `philo` benchmark.

variables s). The query in the `obdet` benchmark is “ $\exists h_1, h_2. p_{o1} = \text{schedule}(p_{i1}, s_1, h_1) \wedge p_{o2} = \text{schedule}(p_{i2}, s_2, h_2)$ ” where h_1 and h_2 describe 2 different schedules. Given a DSL $\mathcal{L}$ containing only public variables, ASPIRE synthesizes the $\mathcal{L}$ -consequence $p_{i1} = p_{i2} \Rightarrow p_{o1} = p_{o2}$, ensuring the system is observationally deterministic.

Each of the 4 `rsrc` benchmarks describes a simple resource allocator; ASPIRE synthesizes properties describing the minimum number of resources that must (or may) cause a deadlock. Each of the 3 `race` benchmarks describes two threads; ASPIRE can discover the necessary (or sufficient) ways to place a critical section to prevent race conditions. Details are shown in Appendix D.4.

5.7.4 Application 3: Solving Two-Player Games

In this section, we show how ASPIRE can be used to synthesize generalized strategies for solving two-player games. In particular, by carefully designing the DSL, ASPIRE can synthesize sets of winning strategies expressed in a symbolic form, rather than a single concrete strategy.

We illustrate the idea using an example by Bloem et al. [20], called `rg` (for request/grant). The two players take on the roles of client and server, and in each round, the server decides whether to grant (g) or not ($\bar{g}$) the request for that round, and then the client decides whether to send (r) or not ($\bar{r}$) a request in that round. To win the game, the server must grant every request in the same or next round. Bloem et al. [20] show the server player can be in 3 possible states: (i) q_0 : no ungranted request (ii) q_1 : an ungranted request in the last round (iii) q_2 : ungranted requests 2 or more rounds ago. The server should prevent entering state q_2 .

One of the winning strategies from the server side is to always grant on both state q_0 and q_1 . We denote such a strategy as $\alpha[q_0] = g \wedge \alpha[q_1] = g$ —i.e., $\alpha[q] = a$ denotes that strategy α chooses action a when in state q . We can find winning strategies by modeling the `rg` game as a query “ $\exists \beta. w = \text{play}(\alpha, \beta)$ ”, where the client’s strategy β is existentially quantified and $\text{play}(\alpha, \beta)$ is the game controller that takes the strategy of both players and produces a Boolean value w denoting whether the server wins after playing the game. Note that the way we model 2-player games can also be extended to multi-player games by introducing a set of opponent strategies $\{\beta_1, \dots, \beta_k\}$.

The generality of the LOUD framework allows ASPIRE to solve two-player games using the following queries: (i) *Must-win strategy*: what strategy α can guarantee a win for *any* strategy β (Equation (5.20))?

$$\forall \alpha, w. (\exists \beta. w = \text{play}(\alpha, \beta)) \Rightarrow (P_{\text{must}}(\alpha) \Rightarrow w = T) \quad (5.20)$$

and (ii) *May-win strategy*: what strategy α can win for *at least one* strategy β (Equation (5.21))?

$$\forall \alpha, w. (P_{\text{may}}(\alpha) \wedge w = T) \Rightarrow (\exists \beta. w = \text{play}(\alpha, \beta)) \quad (5.21)$$

Looking at Equation (5.20), if we provide a DSL $\mathcal{O}$ that expresses formulas in the form “ $P_{\text{must}}(\alpha) \Rightarrow w = T$ ”, we can extract the must-win strategy in the P_{must} part of the synthesized formulas. By replacing T with F we can get the must-lose strategy. For the `rg` game, ASPIRE synthesized the following $\mathcal{O}$ -consequences, which tells us that the server will always win if they grant requests in either of states q_0 and q_1 —i.e., ASPIRE finds “a set of” winning strategies.

$$\alpha[q_0] = g \Rightarrow w = T \quad \alpha[q_1] = g \Rightarrow w = T \quad (5.22)$$

When provided with the dual DSL $\mathcal{U}$ ASPIRE also synthesized the following $\mathcal{U}$ -

implicants:

$$\alpha[q_0] = \bar{g} \wedge \alpha[q_1] = \bar{g} \wedge w = F \quad w = T \quad (5.23)$$

The first $\mathcal{U}$ -implicant states that the server may lose if they do not grant requests at both states q_0 and q_1 , whereas the second $\mathcal{U}$ -implicant states that whatever strategy the server uses there exists a strategy of the requester (i.e., the one that never issues requests) that causes the server to win.

Other benchmarks

We consider a total of 5 benchmarks: `rg` (discussed above), `nim2` (the Nim game), and `temp` (a temperature controller), which are adapted from linear reachability games by Farzan and Kincaid [48]⁴, and `num1` and `num2`, which are games designed by us in which two players manipulate an integer where one player’s goal is to keep the integer in a certain range. Because of the implementation bounds discussed in section 5.6, we stipulate that player 1 (typically the player that needs to stay in safe states) wins, after a finite number (we set as 15) of rounds of play.

It takes ASPIRE less than 85 seconds to synthesize must/may strategies for each benchmark. Compared to the work by Farzan and Kincaid [48], ASPIRE synthesizes (i) not only *must*-but also *may*-strategies, (ii) properties on desired strategies instead of a concrete strategy, and (iii) general strategies that work for games with parameters (e.g. the initial number of pebbles in `nim2`).

Details of DSL design and synthesized properties of benchmarks are provided in Appendix B.

⁴All other games studied by Farzan and Kincaid cannot be modeled in ASPIRE due to the restricted features of SKETCH languages, such as limited support to floating point numbers.

5.8 Related Work

Abstract Interpretation

Many static program-analysis and verification techniques represent large program state spaces symbolically as predicates, using *abstract interpretation* [30]. While the majority of works on abstract interpretation has been focused on over-approximation, it can also be used to describe under-approximations of the program behavior [62, 134, 22]. In particular, the best $\mathcal{L}$ -conjunction synthesis problem is an instance of *strongest-consequence problem* [128]. Given a formula ϕ in logic $\mathcal{L}_1$ (with interpretation $\llbracket \cdot \rrbracket_1$), the goal of strongest-consequence problem is to determine the strongest formula ψ that is expressible in a different logic $\mathcal{L}_2$ (with interpretation $\llbracket \cdot \rrbracket_2$) such that $\llbracket \phi \rrbracket_1 \subseteq \llbracket \psi \rrbracket_2$. One existing technique to solve this problem identifies a chain of weaker implicants until one becomes a consequence of ϕ [129], whereas other techniques take the opposite direction, identifying a chain of stronger consequences [150, 151]. Unlike these approaches, our framework LOUD (like SPYRO [116]) supports a customizable DSL, avoiding requirements to perform operations on elements within the DSL $\mathcal{L}$, such as join [150]. The ability to modify the DSL is what makes the LOUD framework applicable to many domains.

Best $\mathcal{L}$ -term Synthesis

The idea of synthesizing a “best” term from a customizable DSL $\mathcal{L}$ was first proposed by Kalita et al. [81], where the goal was to synthesize a most-precise abstract transformer for a given abstract domain. SPYRO generalized the idea and introduced the setting to define and solve the problem of synthesizing best $\mathcal{L}$ -conjunctions. In these works, the “best” term should be (i) *sound*: it is a valid approximation to the best transformer in Kalita et al. [81] or the semantics of query in chapter 4, and (ii) *precise*: it is minimal w.r.t. a preorder defined on $\mathcal{L}$.

The LOUD framework takes a step further: it further generalizes the queries to allow existential quantifiers and introduces the problem of synthesizing weakest $\mathcal{L}$ -implicants and best $\mathcal{L}$ -disjunctions. Logically, the LOUD framework subsumes both the work by Kalita et al. and SPYRO.

At the algorithmic level, the tools solving the above problems all use two kinds of examples for synthesis, where one is treated as hard constraints to guarantee soundness and the other one is treated as soft constraints to guarantee precision. SPYRO improved the algorithm by Kalita et al. by introducing the idea of freezing examples, thus avoiding the need for a synthesizer with hard and soft constraints. The CEGQI algorithm we present in Section 5.5 is a new approach that is not present in the aforementioned works as none of them supports existential quantifiers in their queries.

Program Logic

Hoare [70] and incorrectness logic [110, 35] can reason about program properties through preconditions and postconditions. If one treats the DSL $\mathcal{L}$ as an assertion language, the problems of computing *strongest postcondition* [39] and *weakest liberal precondition* [38] in Hoare logic, and *weakest under-approximation postcondition* and *weakest possible precondition* [71] in incorrectness logic, can be expressed within the LOUD framework (see Section 5.3.1).

One key distinction between our approach and the one used in automating computing the above operations in program logics is that in the LOUD framework, one can specify what DSL $\mathcal{L}$ they want their properties to be expressed in. In contrast, the properties produced automatically for, e.g., weakest possible preconditions in incorrectness logic, are the results of syntactic rewrites that often result in complex properties with potentially many quantifiers.

Invariant inference

Many data-driven, CEGIS-style algorithms can infer program invariants—e.g., Elrond [163], abductive inference [41], ICE-learning [52], LoopInvGen [112], Hanoi [106], and Data-Driven CHC Solving [164]. Dynamic techniques like Daikon [46, 47], QuickSpec [143] and PreciS [9] can also synthesize invariants through program traces or random tests. The LOUD framework differs from the above works in three key ways: (i) The language $\mathcal{L}$ is customizable and is not limited to a set of predefined predicates, and thus the LOUD framework can be used in a domain-agnostic way (as showcased by the many applications presented in Section 5.7);

(ii) the LOUD framework supports both over-approximated and under-approximated reasoning, and (iii) the properties synthesized by LOUD are *provably sound strongest $\mathcal{L}$ -consequences and sound weakest $\mathcal{L}$ -implicants*.

Quantifier Elimination

Many algorithms [42, 40, 25, 86] are built on abductive inference, specifically, approximate quantifier elimination. Gulwani and Musuvathi [63] defined overapproximate existential quantifier elimination as a “cover operation”, where the goal is, given a formula $\exists V.\phi$, to find a quantifier-free formula φ such that $(\exists V.\phi) \Rightarrow \varphi$. If φ is restricted to be in a DSL $\mathcal{L}$, the cover problem corresponds to synthesizing $\mathcal{L}$ -consequences for queries with an existential quantifier in LOUD framework. Some algorithms [40, 86] also define underapproximate existential quantifier elimination, which corresponds to synthesis of $\mathcal{L}$ -implicants. Other approaches are limited to specific theories or require nontrivial, theory-specific primitives [48, 18]. LOUD framework differs from above works because it allows custom DSLs that express the target quantifier-free formulas and thus is not restricted to any fixed theory.

Logic-Based Learning

The angelic and demonic behaviors of nondeterminism correspond to the concepts of brave entailment (i.e., entailment from *some* answer set) and cautious entailment (i.e., entailment from *every* answer set) in logic-based learning [91]. However, the angelic notion in LOUD does not precisely correspond to cautious entailment. While angelic specifications represent an under-approximation of possible behavior, cautious entailment does not necessarily imply an under-approximation. This distinction mirrors the difference between forward reasoning in incorrectness logic and backward reasoning in Hoare’s possible correctness, as discussed in Section 5.3.

Under-approximation

The LOUD framework could potentially be combined with existing compositional under-approximate reasoning techniques, such as incorrectness logic [110] or compositional symbolic

execution [58]. An inherited limitation of syntax-directed under-approximate reasoning is the inability to effectively reason about statements or procedures involving constraints beyond the scope of the theory $\mathcal{T}$ assumed by the under-approximate reasoning framework. We expect one could synthesize weakest $\mathcal{L}$ -implicants to approximate such constraints into summaries that are expressible in the theory $\mathcal{T}$ assumed by under-approximation frameworks.

5.9 Conclusion

This chapter presented LOUD, a general framework for synthesizing over- and under-approximated specifications of both deterministic and nondeterministic programs, thus enabling broad applications—e.g., describing sources of bugs in concurrent code and finding winning strategies in two-player games. The chapter also presents general procedures for solving LOUD problems using simple synthesis primitives that do not involve complex quantifier alternations.

Currently, our tool ASPIRE is implemented on top of the SKETCH synthesizer, which results in some limitations. First, synthesized formulas are only sound for inputs up to a given bound. Such an issue could be addressed by combining our approach with an off-the-shelf verifier; however, we are not aware of verifiers that can reason about $\mathcal{L}$ -implicants—i.e., under-approximated specifications. Our work provides a motivation for building such verifiers. Second, SKETCH limits us from exploring applications that involve inputs of unbounded length—e.g., reasoning about infinite traces, LTL formulas, and reactive systems. Our work thus opens an opportunity for the research community: by improving efficiency and providing stronger soundness guarantees for the primitives used to solve LOUD problems, researchers can tackle the many applications supported by the framework.

Acknowledgments

Chapter 5, in full, is based on the paper from Kanghee Park, Xuanyu Peng, and Loris D’Antoni “LOUD: Synthesizing Strongest and Weakest Specifications” published in Proceedings of the ACM on Programming Languages 9(OOPSLA1), pp. 956-983. 2025. The dissertation

author is the primary investigator and author of this paper.

Part III

Modular Program Reasoning

Chapter 6

Modular System Synthesis

Large systems are rarely understood or constructed as a monolithic program. They are divided into modules so that one component can be changed without requiring every client to understand its representation. A module interface, or contract, establishes the correctness boundary between an implementation and its clients: the implementation must provide the promised behavior, and clients may rely on that behavior but not on hidden details.

Today, frontier models can generate code at scale, yet they lack formal guarantees of correctness. To bridge this gap, an emerging paradigm combines generative AI with formal specifications to ensure code provably meets user intent. However, this paradigm confronts the exact same bottleneck that has long plagued traditional program synthesis: it fails to scale to large software systems because it attempts to synthesize and reason about programs in their entirety.

The obvious direction is to try to exploit *compositionality* and synthesize larger programs by having them invoke other (already synthesized) programs. Consider for example the problem of writing a program for a ticket-vendor application that can, among other things, issue and reserve tickets. Building such a system requires creating modules for various data structures—perhaps a stack and queue—and using these modules in a top-level module that processes ticket requests. It is natural to ask whether such modules can be synthesized separately—i.e., in a compositional fashion.

Information hiding provides a crucial *trust boundary* for program reasoning. Without it, higher-level tasks must analyze all accumulated implementation details, leading to search space explosion. Reasoning exclusively through trusted, implementation-agnostic contracts allows automated tools to bound their search space to the current layer. Modularity is thus a key method for controlling formal reasoning rather than a passive software-engineering habit.

The fundamental question is

Can one address the scalability problem of program synthesis by exploiting compositionality, so that (i) larger programs are synthesized from smaller programs, and (ii) each “larger-created-from-smaller” synthesis sub-problem is of a similar nature, so that the essence of each sub-problem (and the size of each search space) has roughly the same character?

A solution to this question is surprisingly tricky to envisage. Most existing synthesis approaches require having a concrete semantics or implementation in hand when reasoning about modules, components, APIs, etc. [49, 137, 141], and such synthesis tools end up reasoning about the entire program all the way down to its lowest-level components. Not only is this approach in fundamental opposition to the “similar-nature/similar-size” principle articulated above, it makes synthesis increasingly hard as more modules are considered.

Instead, when code is synthesized for some module M , all reasoning about lower-level modules $\{M_i\}$ on which M *directly* depends should be carried out in a way that is *agnostic* about the implementations of $\{M_i\}$. This observation leads us to pose two related challenges: (i) How can one carry out program synthesis without having in hand details about the implementations of lower-level modules? (ii) How can one ensure that each synthesis problem results in code that is independent of the implementations of lower-level modules?

In this chapter, we present the case for the following thesis:

Program synthesis can scale using modular system design.

Modular system design is one of the most important concepts in designing software.

A system should be organized in a layered fashion, where information hiding is used to hide implementation choices [118]. The *information-hiding* principle intuitively states that each module exports an interface that does not reveal specific implementation choices used inside the module, and changing the module’s implementation should not force any changes to be made to other modules.

Programmers practice modular system design, or at least aspire to it. In essence, our goal is to provide a level of automation for what good programmers do manually. Of course, we are not trying to automate everything. What is left in the hands of the programmer are architectural decisions and specifications of the intended behavior of individual modules. The programmer is responsible for the overall organization of the system’s design, and must decide such issues as: What are the layers in the system? What are the implementation choices in a given layer (such as choices about data structures and data representations)? What operations are exposed in each layer, and what is the intended behavior of each operation?

We identify *two* opportunities for providing automation for each module and, as a key contribution of this chapter, we formally define these synthesis problems.

Module-Implementation Synthesis

Synthesis can be helpful in creating the implementations of the various functions in each module from some specifications. The key difference from traditional synthesis problems is that implementation details of “lower” modules are not available. Instead, one only has access to *implementation-agnostic specifications* of the semantics of such modules.

Module-Specification Synthesis

Because modules can only expose their semantics to other modules in a way that does not reveal their implementation details, it can be challenging to come up with such semantic definitions. We propose to automate the creation of such implementation-agnostic semantic definitions using synthesis, namely, *synthesis of formulas*.

Note the role of the second kind of synthesis problem: its results provide part of the

specification when one moves on to the task of synthesizing the implementation of functions in the next module. By analogy with the Paul Simon lyric “one man’s ceiling is another man’s floor” [140], we have “one module’s semantics is another module’s primitives.”

We call this approach *modular system synthesis* (MOSS). The visibility restrictions of information hiding provide the key for MOSS to achieve the objective of making synthesis scalable via “similar-nature/similar-size” sub-problems: both of our synthesis problems concern a single module of the system, and a single module’s implementation only. By concealing the implementation of lower-level modules, MOSS ensures that the formula representing the semantics of these layers remains independent of the size of the “accumulated” system as we move to higher-level layers. Moreover, MOSS retains the usual benefit of modular system design, namely, it results in software that (usually) can be readily adapted—in this context, re-synthesized—as requirements change.

This chapter contributes both a framework and solidifying the concept of contract-based design in the context of program synthesis, which abstracts components or sub-systems based on their interfaces. Notably, the study of interface compatibility and composition has not been extensively explored in the context of program synthesis, opening up many opportunities for future developments. Specifically, using the aforementioned ticket-vending application as an example (§6.1), it (i) defines modular system synthesis (§6.2); (ii) defines the two kinds of synthesis problems that arise in MOSS (§6.3); and (iii) describes a proof-of-concept system, called MOSSKIT, that achieves these goals (§6.4).

MOSSKIT is based on two existing program-synthesis techniques: JLIBSKETCH [101] a program-sketching tool that supports algebraic specifications, and SPYRO [115] a tool for synthesizing precise specifications from a given codebase. We used MOSSKIT to carry out case studies based on two-layer modular synthesis problems from Mariano et al. [101], which demonstrated that concealing lower-level components can be advantageous in reducing the complexity of the synthesis problem. Expanding upon their work, our case study in §6.4.2 further explored scenarios involving multiple layers. MOSS exhibits even better scalability compared to

scenarios where executable semantics for all lower layers are exposed. A further case study based on Mariano et al. in §6.4.4 also highlights the challenges of writing correct specifications. Our framework and the act of performing synthesis for both the implementations and specifications of the modules unveiled bugs in the modules synthesized by Mariano et al. and in the module’s specifications, which they manually wrote.

§6.5 discusses related work. §6.6 concludes.

6.1 Illustrative Example

We present an experiment that illustrates the various aspects of MOSS. The problem to be solved is as follows: Synthesize a simple ticket-vendor application that supports the operations `prepSales`, `resTicket`, `issueTicket`, `soldOut`, `numTicketsRem`, and `numWaiting`. (To simplify matters, we assume it is not necessary to cancel a reservation.)

6.1.1 A Modular TicketVendor Implementation

We decompose the system into three modules (Fig. 6.1):

Module 3: The `TicketVendor` module uses a `Queue` of reservations to implement the aforementioned operations.

Module 2: The `Queue` module implements the operations `emptyQ`, `enq`, `front`, `deq`, `sizeQ`, and `isEmptyQ`. In our setting, a `Queue` is implemented using two stacks [73].¹

Module 1: The `Stack` module implements the operations `emptyS`, `push`, `top`, `pop`, `sizeS`, and `isEmptyS`. In our setting, a `Stack` is implemented using linked-list primitives of the programming language.

Moreover, the implementation of each module is to abide by the principle of information hiding: (i) The `TicketVendor` module can use operations exposed by `Queue`, but their actual

¹The invariant is that the second `Stack` holds a prefix of the `Queue`’s front elements, with the top element of the second `Stack` being the `Queue`’s front-most element. The first `Stack` holds the `Queue`’s back elements—with the top element of the first `Stack` being the `Queue`’s back-most element.

implementations are hidden in Module 2. (ii) The Queue module can use operations exposed by Stack, but their actual implementations are hidden in Module 1.

6.1.2 The Input of Modular TicketVendor Synthesis

A MOSSKIT user supplies the following information:

Architectural-design choices

- The decomposition of the problem into TicketVendor, Queue, and Stack modules (gray boxes in Fig. 6.1).
- Which operations are to be exposed by each module, denoted by $\mathcal{P}[module]$ —e.g., in Fig. 6.1, the Queue module exposes $\mathcal{P}[Queue]$, which contains enq and deq operations, but not push and pop operations on the underlying stacks.

Data-structure/data-representation choices

- **Module 3:** TicketVendor uses a Queue.
- **Module 2:** A Queue is implemented using two Stacks.
- **Module 1:** A Stack is implemented using a linked list.

These choices are shown by the green boxes underneath each module in Fig. 6.1. For example, the Queue module is built on top of the Stack module. However, only the Stack interface—i.e., the function symbols in $\mathcal{P}[Stack]$ and its (potentially synthesized) implementation-agnostic specification ϕ_{sem}^{Stack} —is accessible by the Queue module.

Specifications of the module-specific synthesis problems

Module 3: Specifications of the behaviors of prepSales, resTicket, issueTicket, soldOut, numTicketsRem, and numWaiting in terms of the exposed Queue operations (and possibly other TicketVendor operations). For example, the implementation-specific specifications for the TicketVendor module, denoted by the yellow box labeled $\phi_{imp}^{TicketVendor}$ in

Fig. 6.1, might constrain `issueTicket` to dequeue a buyer from the underlying Queue module, but only if `soldOut` (a `TicketVendor` operation) is false.

Module 2: Specifications of the behaviors of the Queue operations in terms of the exposed Stack operations (and possibly other Queue operations). For example, the implementation-specific specification for the Queue module (φ_{imp}^{Queue}), shown in Fig. 6.1, contains, among others, constraints that state that (i) if the first stack st_{in} is empty, so is the second stack st_{out} , (ii) enqueueing 1 on an empty queue and then retrieving the front of the queue yields 1.

Module 1: Specifications of the behaviors of the Stack operations in terms of the programming language’s linked-list operations (and possibly other Stack operations). For example, the implementation-specific specification of the Stack module (φ_{imp}^{Stack}) might specify that push adds an element on the front of the stack’s underlying linked list.

A user must also specify a search space of possible implementations. In MOSSKIT, this is done using a SKETCH file.

6.1.3 The Output of Modular TicketVendor Synthesis

Using the MOSS framework, we synthesize the implementations of three modules: the `TicketVendor` module implementation, which satisfies $\varphi_{imp}^{TicketVendor}$ (and uses Queue); the Queue module implementation, which satisfies φ_{imp}^{Queue} (and uses Stack); and the Stack module implementation, which satisfies φ_{imp}^{Stack} (and uses lists). However, to synthesize the `TicketVendor` module implementation, we need an *implementation-agnostic specification* of Queue, denoted by φ_{sem}^{Queue} . The same can be said for the Queue module implementation, for which we need an implementation-agnostic specification of Stack, denoted by φ_{sem}^{Stack} .²

The user could write φ_{sem}^{Queue} and φ_{sem}^{Stack} manually, but it is more convenient to synthesize these specifications from the Queue and Stack module implementations, respectively. The MOSS methodology is to start with the bottom-most module and work upward, alternately

²Technically, List is part of the programming language; however, so that all sub-problems have the same form, we assume—as shown in Fig. 6.1—that we also have available an implementation-agnostic specification of List, denoted by φ_{sem}^{List} . In our evaluation, we synthesize φ_{sem}^{List} automatically.

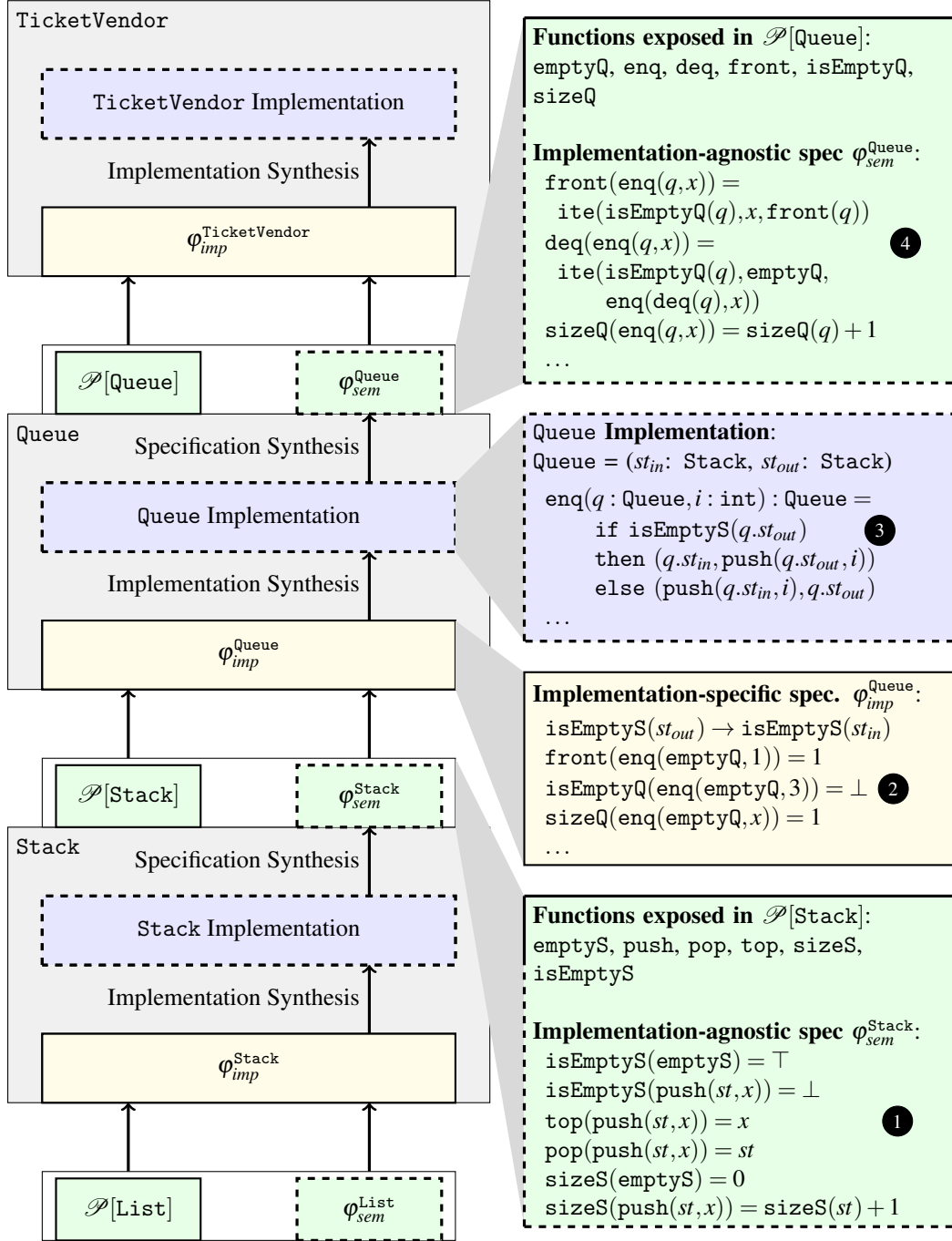


Figure 6.1. Organization of the modular TicketVendor synthesis problem: user-supplied inputs are shown in solid boxes; synthesized outputs are shown in dashed boxes. On the right, the Queue module's specifications and implementation are expanded; the other modules would have similar details.

applying two synthesis procedures: first synthesizing the implementation of a module M and then synthesizing M 's implementation-agnostic specification φ_{sem}^M , which gets exposed to the next higher module.

For the modular TicketVendor-synthesis problem, we start with Stack, the bottommost module, and synthesize a Stack module implementation—a set of $\mathcal{P}[\text{List}]$ programs—that satisfies the implementation-specific specification $\varphi_{imp}^{\text{Stack}}$. (In MOSSKIT, this step is done using program sketching and the tool JLIBSKETCH [101].) This step is depicted in Fig. 6.1 as the Implementation Synthesis problem in the Stack module. We then switch to the Specification Synthesis problem for Stack, and synthesize $\varphi_{sem}^{\text{Stack}}$, an implementation-agnostic specification of Stack. (In MOSSKIT, this step is done by providing a grammar of possible properties and by using the tool SPYRO from chapter 4.) For the Stack module, the resultant $\varphi_{sem}^{\text{Stack}}$ is the conjunction of the equalities shown at ❶ in Fig. 6.1.

Using $\varphi_{sem}^{\text{Stack}}$ (❶), together with the implementation-specific specification $\varphi_{imp}^{\text{Queue}}$ (❷), we now synthesize the Queue module implementation (❸)—a set of $\mathcal{P}[\text{Stack}]$ programs—and the implementation-agnostic specification $\varphi_{sem}^{\text{Queue}}$ (❹) via the same two-step process.

Finally, using $\varphi_{sem}^{\text{Queue}}$ and the implementation-specific specification $\varphi_{imp}^{\text{TicketVendor}}$, we synthesize the TicketVendor module implementation. (If needed by a further client, we would then synthesize the implementation-agnostic specification $\varphi_{sem}^{\text{TicketVendor}}$.) Thus, the last output of the synthesis procedure, shown in Fig. 6.1, consists of implementations of Stack, Queue, and TicketVendor, and the implementation-agnostic specifications $\varphi_{sem}^{\text{Stack}}$ and $\varphi_{sem}^{\text{Queue}}$.

6.1.4 Benefits of Modular System Synthesis

At some point, we might want to decide to modify the implementation of the Queue module to use directly the linked-list primitives provided by the language (shown in Fig. 6.2). Information hiding allows us to do so in a compartmentalized way—i.e., by only changing the specific Queue module. Importantly, the module's interface, composed of the function symbols in $\mathcal{P}[\text{Queue}]$ and its implementation-agnostic specification $\varphi_{sem}^{\text{Queue}}$, does not change when the

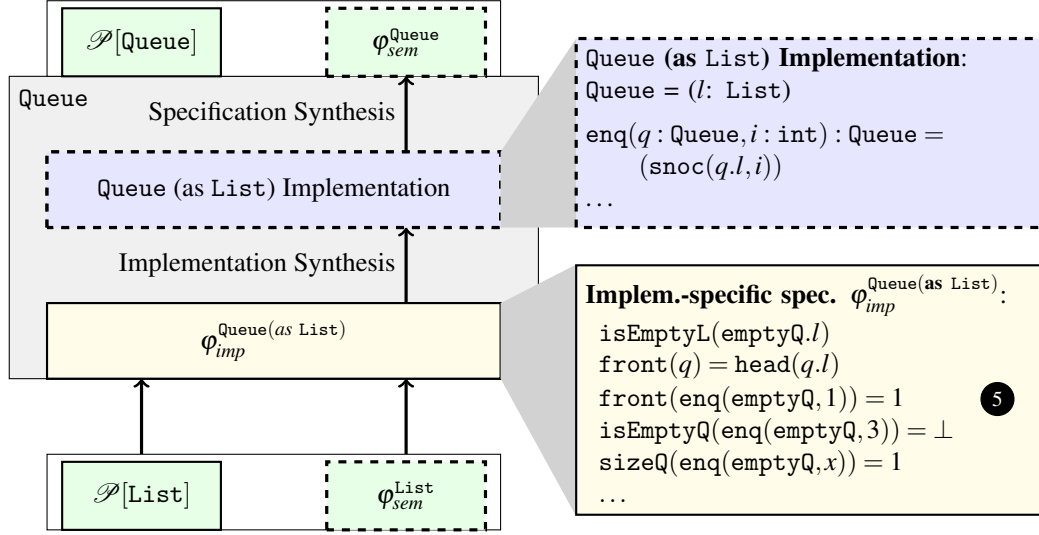


Figure 6.2. Alternative implementation of the Queue module using list primitives instead of two stacks. $\mathcal{P}[\text{Queue}]$ and ϕ_{sem}^{Queue} are the same as in Fig. 6.1.

implementation of the Queue module changes. Because this interface is what the TicketVendor module was synthesized with respect to, changes to the Queue implementation are not visible to TicketVendor.

6.2 Modular System Design

In this section, we formally define modular system design and the corresponding specification mechanisms. A system is organized in modules, and each module exports a module interface MI and a specification ϕ_{sem}^{MI} of the semantics of the module interface. Both MI and ϕ_{sem}^{MI} hide the module's implementation. A module's implementation can also have a set of private functions PF , which can only be used within the module. A program is constructed by stacking layers of such modules.³ For instance, the example in Fig. 6.1 has three modules: Stack, Queue, and TicketVendor. (None of those modules have private functions.)

In the following, we assume a programming language $\mathcal{P}$ (e.g., C with its core libraries), and use $\mathcal{P}[MI]$ to denote $\mathcal{P}$ extended with the functions exposed by module MI .

³In general, the structure of the dependencies among layers can form a directed acyclic graph. However, to reduce notational clutter, throughout this chapter we assume that the layers have a strict linear order.

Definition 6.2.1 (Modular System Design). A system is **implemented modularly** if it is partitioned into disjoint sets of functions $PF_1, MI_1, PF_2, MI_2, \dots, PF_n, MI_n$, such that for each $f \in PF_i \cup MI_i$, f is implemented using $\mathcal{P}[MI_{i-1} \cup PF_i \cup MI_i]$ —i.e., f only uses operations in $\mathcal{P}$, and calls to functions in the interface exported from layer $i-1$, to private functions of layer i , and to functions in the interface exported from layer i .

To reduce notational clutter, we will ignore private functions, and only discuss the functions in module interfaces.

As we saw in §6.1, we need to abide by the principle of *information hiding*—i.e., changing the implementations of any function in MI_{i-1} should not require changing the implementations of functions in MI_i . With this principle in mind, we now describe the different natures of the specification for the module implementation at a given layer i (§6.2.1) and the specification exposed to layer $i+1$ (§6.2.2).

6.2.1 Implementation-specific Specifications

When synthesizing specific implementations of the functions MI_i at layer i , the specifications are allowed to use symbols in $\mathcal{P}[MI_{i-1} \cup MI_i]$ —i.e., the specification can refer to the functions we are specifying and to the ones in the interface exported from the previous layer—as well as implementation-specific details from layer i (e.g., data-structure declarations).

Definition 6.2.2. An **implementation-specific specification** for a set of functions MI_i at layer i is a predicate $\phi_{imp}^{MI_i}$ that only uses symbols in $\mathcal{P}[MI_{i-1} \cup MI_i]$.

Example 20. In the implementation-specific specification of *Queue* from Fig. 6.1, where *Queue* is implemented using two *Stacks*, one of the properties is as follows:

$$isEmptyQ(q) \iff isEmptyS(q.st_{in}) \wedge isEmptyS(q.st_{out}).$$

For the version from Fig. 6.2, where *Queue* is implemented using a *List*, the analogous property

is

$$isEmptyQ(q) \iff isEmptyL(q.l).$$

A specification might also contain a set of examples, e.g., $front(enq(emptyQ, 1)) = 1$ and $front(enq(enq(emptyQ, 1), 2)) = 1$.

6.2.2 Implementation-agnostic Specifications

While implementation-specific details are needed to converge on an implementation with which the programmer is happy, when exposing the specification of MI_i at layer $i + 1$, to abide to the principle of information hiding, one cannot provide specifications that involve function symbols in $\mathcal{P}[MI_{i-1} \cup MI_i]$, but only those in $\mathcal{P}[MI_i]$.

Definition 6.2.3. An *implementation-agnostic specification* for a set of functions MI_i at layer i is a predicate $\phi_{sem}^{MI_i}$ that only uses symbols in $\mathcal{P}[MI_i]$.

Example 21. Because of the vocabulary restrictions imposed by Def. 6.2.3, it is natural for implementation-agnostic specifications to take the form of algebraic specifications [146, 65, 59, 96, 66]. For instance, for the *Queue* module, the conjunction of the following equalities is an implementation-agnostic specification ϕ_{sem}^{Queue} for *Queue*:

$$\begin{aligned} isEmptyQ(emptyQ) &= \top \quad isEmptyQ(enq(q, x)) = \perp \\ sizeQ(emptyQ) &= 0 \quad sizeQ(enq(q, x)) = sizeQ(q) + 1 \\ front(enq(q, x)) &= ite(isEmptyQ(q), x, front(q)) \\ deq(enq(q, x)) &= ite(isEmptyQ(q), emptyQ, enq(deq(q), x)) \end{aligned} \tag{6.1}$$

Note that Eq. (6.1) serves as ϕ_{sem}^{Queue} both for the version of *Queue* from Fig. 6.1, where *Queue* is implemented using two *Stacks*, and for the version of *Queue* from Fig. 6.2, where *Queue* is implemented using a *List*.

6.3 Synthesis in Modular System Synthesis

In this section, we define the *implementation-synthesis* problem (§6.3.1) and *specification-synthesis* problem (§6.3.2) that enable our scheme for modular system synthesis.

6.3.1 Synthesis of Implementations

The obvious place in which synthesis can be helpful is in synthesizing the implementations of the various functions at each layer from their implementation-specific specifications. For example, in Fig. 6.1, an implementation of Queue (the function `enq` is shown in the second box on the right) is synthesized from the implementation-agnostic specification ϕ_{sem}^{Stack} of Stack, and an implementation-specific specification ϕ_{imp}^{Queue} that is allowed to talk about how the two Stacks used to implement a Queue are manipulated (e.g., $isEmptyS(st_{out}) \rightarrow isEmptyS(st_{in})$).

Definition 6.3.1 (Implementation synthesis). *For module interface MI_i , the **implementation-synthesis** problem is a triple $(S_i, \phi_{sem}^{MI_{i-1}}, \phi_{imp}^{MI_i})$, where*

- S_i is the set of possible implementations we can use for MI_i (every program in S_i uses only symbols in $\mathcal{P}[MI_{i-1} \cup MI_i]$).
- $\phi_{sem}^{MI_{i-1}}$ is an implementation-agnostic specification of the module-interface functions in MI_{i-1} .
- $\phi_{imp}^{MI_i}$ is an implementation-specific specification that uses only symbols in $\mathcal{P}[MI_{i-1} \cup MI_i]$.

A solution to the implementation-synthesis problem is an implementation of MI_i in S_i that satisfies $\phi_{imp}^{MI_i}$.

This particular form of synthesis where one draws a program from a search space to match a specification is fairly standard in the literature. However, we observe that a particular aspect of modular system design makes most synthesis approaches inadequate—i.e., the specification $\phi_{sem}^{MI_{i-1}}$ can talk about functions in MI_{i-1} only in an implementation-agnostic way. For example,

when synthesizing functions in Queue, we do not have direct access to a stack implementation—i.e., we cannot actually execute the implementation. Instead, we have access to the semantics of Stack through implementation-agnostic properties such as $\text{isEmptyS}(\text{push}(st, x)) = \perp$.

We are aware of only one tool, JLIBSKETCH, that can perform synthesis with algebraic specifications [101], and we use it in our evaluation. In JLIBSKETCH, one provides S_i as a program sketch (i.e., a program with integer holes that need to be synthesized), $\phi_{sem}^{MI_{i-1}}$ as a set of rewrite rules over the functions in MI_{i-1} , and $\phi_{imp}^{MI_i}$ as a set of assertions.

6.3.2 Synthesis of Implementation-agnostic Specifications

Because the implementation of layer $i-1$ is hidden when performing synthesis at layer i , the user has to somehow come up with implementation-agnostic specifications like the ones shown in Fig. 6.1. Our next observation is that such specifications can also be synthesized! With this observation, modular system design becomes a fairly automatic business where the programmer mostly has to decide how to structure modules and provide implementation-specific specifications and search spaces (typically as regular-tree grammars [28]).

In Fig. 6.1, the implementation-agnostic specification $\phi_{sem}^{\text{Queue}}$ of Queue is synthesized from the Queue implementation. (The same $\phi_{sem}^{\text{Queue}}$, or one equivalent to it, is synthesized from the alternative Queue implementation of Fig. 6.2.)

Definition 6.3.2 (Specification synthesis). *For module interface MI_i , a **specification-synthesis problem** is a pair (F_i, Φ_i) where*

- F_i is a set of programs, written in $\mathcal{P}[MI_{i-1} \cup MI_i]$, that is a concrete implementation of MI_i .
- Φ_i is the set of possible properties we can use for $\phi_{sem}^{MI_i}$ (every property in Φ_i uses only symbols in $\mathcal{P}[MI_i]$). (Typically, Φ_i is given as a regular-tree grammar for a fragment of logic in which terms can only use symbols in $\mathcal{P}[MI_i]$.)

A solution to the specification-synthesis problem is a set of properties $\varphi_{sem}^{MI_i} \subseteq \Phi_i$ such that for every $\alpha \in \varphi_{sem}^{MI_i}$:

Soundness: The implementation F_i satisfies α .

Precision: There is no property $\alpha' \in \Phi_i$ that implies α and such that the implementation F_i satisfies α' .

In general, there might not be just one answer to this synthesis problem because there could be multiple ways to build the set of properties $\varphi_{sem}^{MI_i}$. Furthermore, it can be the case that there are infinitely many properties in Φ_i that are sound, precise, and mutually incomparable. While in this chapter we do not worry about these details, the tool we use in our evaluation SPYRO is always guaranteed to find a maximal set of properties in Φ_i whenever such a set is finite (SPYRO uses a regular-tree grammar to describe the set of possible properties Φ_i , but requires such a set to be finite.) In practice, even when the set is infinite, one can build tools that find a “good” set of properties and stop without trying to find an exhaustive set.

Discussion

When the goal is to build a system structured in a modular fashion, modular system synthesis enables defining “small” synthesis problems of similar nature that concern only a single module’s implementation.

While implementation-agnostic specifications can be synthesized via the synthesis problem defined in Def. 6.3.2, one should be aware that there is additional flexibility to be gained if one is willing to write implementation-agnostic specifications manually. In particular, if all of the implementation-agnostic specifications are synthesized, then it is necessary to create the system *bottom-up*, synthesizing the module implementations in the order $MI_1, MI_2, \dots, MI_n$ (interleaved with the synthesis of $\varphi_{sem}^{MI_1}, \varphi_{sem}^{MI_2}, \dots, \varphi_{sem}^{MI_n}$). In contrast, when the user is willing to write the implementation-agnostic specifications manually (in addition to the implementation-specific specifications $\{\varphi_{imp}^{MI_i}\}$), then the module implementations for $MI_1, MI_2, \dots, MI_n$ can be synthesized in any order.

6.4 Implementation and Case-Study Evaluation

We carried out case studies of MOSS for the simple three-layer system that has been used as a running example and for some of the modular-synthesis problems presented in the paper that introduced JLIBSKETCH [101].

6.4.1 Implementation

Our implementation, called MOSSKIT, uses JLIBSKETCH [101] to synthesize the implementation code for each layer k (from the implementation-specific specification for layer k) and SPYRO to synthesize the implementation-agnostic specification for use at layer $k + 1$.

JLIBSKETCH is a program-synthesis tool for Java that allows libraries to be described with collections of algebraic specifications. Similar to its popular C counterpart SKETCH [145], JLIBSKETCH allows one to write programs with holes and assertions, and then tries to find integer values for the holes that cause all assertions to hold. Each specification is a rewrite rule of the form $pattern \Rightarrow result$. For instance, one of the rewrite rules in the specification of a stack could be $pop(push(st, k)) \Rightarrow st$. To prevent infinite rewrite loops, a set of rewrite rules provided to JLIBSKETCH must not form a cycle. For instance, the rule $a + b \Rightarrow b + a$ is not allowed. The synthesis problem that JLIBSKETCH addresses is to find a program that is correct for any program input, for any library implementation that satisfies the algebraic specifications.

MOSSKIT inherits the limitations of JLIBSKETCH and SPYRO—i.e., the synthesized implementations and specifications are sound up to a bound. Despite this limitation, the authors of JLIBSKETCH and SPYRO have shown that these tools typically do not return unsound results in practice. §6.4.6 provides a detailed discussion of the limitations of MOSS and MOSSKIT.

6.4.2 Ticket-vendor Case Study

Our first benchmark is the ticket-vending application described throughout this chapter. Our goal is to synthesize the four module implementations in Fig. 6.1 (except the bottom one),

```

1 void snoc(list l, int val, ref list ret_list) {
2     boolean is_empty_ret;
3
4     ret_list = new list();
5     is_empty(l, is_empty_ret);
6     if (is_empty_ret) {
7         ret_list.hd = val;
8         nil(ret.tl);
9     } else {
10         ret_list.hd = l.hd;
11         snoc(l.tl, val, ret.tl);
12     }
13 }

```

Figure 6.3. Implementation of snoc supplied to SPYRO. Returning a value from a function is done by storing the value into a reference parameter of the function.

```

1 var {
2     int v1;
3     int v2;
4     list l;
5     list cons_out;
6     list snoc_out;
7 }
8 relation {
9     cons(v1, l, cons_out);
10    snoc(cons_out, v2, snoc_out);
11 }
12 generator {
13     boolean AP → !GUARD ∨ RHS;
14     boolean GUARD → true
15                     | is_empty(l) | !is_empty(l);
16     boolean RHS → equal_list(snoc_out, L);
17     int I → v1 | v2;
18     list L → l | nil()
19             | snoc(l, I) | cons(I, L);
20 }

```

Figure 6.4. Grammar for the domain-specific language in which SPYRO is to express an extracted List property. The relation definition in lines 8-11 specifies that the variables snoc_out l, v1 and v2 are related by $\text{snoc_out} = \text{snoc}(\text{cons}(l, v1), v2)$. From the grammar (“generator”) in lines 12-20, SPYRO synthesizes best implementation-agnostic properties of form $\text{GUARD} \rightarrow \text{snoc_out} = L$ (implicitly conjoined with $\text{snoc_out} = \text{snoc}(\text{cons}(v1, l), v2)$). In this case, the only expression for GUARD that succeeds is $\top$, and the property synthesized is $\text{snoc_out} = \text{cons}(v1, \text{snoc}(l, v2))$ (with the additional implicit conjunct $\text{snoc_out} = \text{snoc}(\text{cons}(v1, l), v2)$).

as well as the specification of each module that needs to be exposed to a higher-level module.

When synthesizing specifications, due to the scalability limitations of SPYRO, we called SPYRO multiple times with different smaller grammars instead of providing one big grammar of all possible properties of each module. In each call to SPYRO, we provided a grammar in which we fixed a left-hand-side expression of an equality predicate, and asked SPYRO to search for a right-hand-side expression for the equality. We allowed the right-hand-side expression to contain a conditional where the guard can be selected from the outputs of Boolean operators in the module, their negation, or constants. For instance, Figures 6.3 and 6.4 illustrate two inputs provided to SPYRO to solve the specification-synthesis problem for `List`: (i) a program describing the implementation of `List` (Fig. 6.3), and (ii) a grammar describing the set of possible properties (Fig. 6.4).

Because we wanted to use the synthesized equalities as input to JLIBSKETCH when synthesizing the implementation of the next higher-level module, we provided grammars of equalities that avoided generating cyclic rewrite rules. We addressed this issue by limiting the search space for the right-hand-side expression. The function symbols permitted in the right-hand-side expression are one of the functions in the left-hand-side expression, functions used in the implementation of a function in the left-hand-side expression, or constants. Also, the outermost function symbol of the left-hand side can only be applied to a strictly smaller term.

To illustrate some of the properties synthesized by MOSSKIT (that are not shown in Fig. 6.1) the complete set of equalities in the implementation-agnostic specification ϕ_{sem}^{List} synthesized by SPYRO is the following:

$$\begin{aligned}
\text{head}(\text{cons}(hd, tl)) &= tl & \text{isEmptyL}(\text{nil}) &= \top \\
\text{tail}(\text{cons}(hd, tl)) &= hd & \text{isEmptyL}(\text{cons}(hd, tl)) &= \perp \\
\text{sizeL}(\text{nil}) &= 0 & \text{snoc}(\text{nil}, x) &= \text{cons}(x, \text{nil}) \\
\text{sizeL}(\text{cons}(hd, tl)) &= \text{sizeL}(tl) + 1 \\
\text{snoc}(\text{cons}(hd, tl), x) &= \text{cons}(hd, \text{snoc}(tl, x))
\end{aligned}$$

```

1 public void enq(int x) {
2     Stack st_in = this.st_in;
3     Stack st_out = this.st_out;
4
5     assume !st_out.isEmpty() ∨ st_in.isEmpty();
6
7     if (genGuard(st_in, st_out)) {
8         st_in = genStack2(st_in, st_out, x);
9         st_out = genStack2(st_in, st_out, x);
10    } else {
11        st_in = genStack2(st_in, st_out, x);
12        st_out = genStack2(st_in, st_out, x);
13    }
14
15    assert !st_out.isEmpty() ∨ st_in.isEmpty();
16
17    this.st_in = st_in;
18    this.st_out = st_out;
19 }

```

Figure 6.5. JLIBSKETCH sketch of `enq`. Lines 5 and 15 assert the implementation-specific property $\text{isEmptyS}(st_{out}) \rightarrow \text{isEmptyS}(st_{in})$. JLIBSKETCH generates an expression to fill in each occurrence of the generators, `genStack2` and `genGuard`—the reader can think of each of these generators as being grammars from which JLIBSKETCH can pick an expression. For these generators, expressions can be variables or single function calls to functions of the appropriate type—e.g., `genStack2` can generate expressions such as `st_in`, `st_out`, `st_in.pop()`, `st_out.pop()`, etc.

When considering the cumulative time taken to synthesize the algebraic specification of each module, SPYRO took 41 seconds for ϕ_{sem}^{List} (longest-taking property 7 seconds), 34 seconds for ϕ_{sem}^{Stack} (longest-taking property 7 seconds), and 44 seconds for ϕ_{sem}^{Queue} (longest-taking property 13 seconds).

We used JLIBSKETCH to synthesize implementations of the modules. In addition to the implementation-agnostic specification of the module below the one we were trying to synthesize, we provided an implementation-specific specification of the module to be synthesized. For example, the ϕ_{imp}^{Stack} specification involved JLIBSKETCH code with 17 assertions, and the following examples are an excerpt from the ϕ_{imp}^{Stack} specification (x, y , and z are universally

quantified integers that are allowed to be in the range 0 to 10):

$$\begin{aligned} \text{top}(\text{push}(\text{emptyS}, x)) &= x & \text{top}(\text{push}(\text{push}(\text{emptyS}, x), y)) &= y \\ \text{sizeS}(\text{emptyS}) &= 0 & \text{sizeS}(\text{push}(\text{emptyS}, x)) &= 1 \end{aligned}$$

Besides the assertions, we provided JLIBSKETCH with a fairly complete sketch of the structure of the implementation: we provided loops and branching structures, and only asked JLIBSKETCH to synthesize basic statements and expressions. For example, the sketch provided for the operation `enq` of module `Queue` = $(st_{in} : \text{Stack}, st_{out} : \text{Stack})$ is shown in Fig. 6.5. This sketch of `enq` of module `Queue` uses two `Stack`s: st_{in} , which stores elements in the rear part of the queue, and st_{out} , which stores elements in the front part of the queue. Stack st_{in} holds the rearmost element on top, and Stack st_{out} stores the frontmost element on top. To make the front operation more efficient, we decided to make sure that the frontmost element is always at the top of st_{out} . This implementation decision is expressed as assertions in lines 5 and 15, constituting an implementation-specific specification $\phi_{imp}^{\text{Queue}}$, shown as ② in Fig. 6.1.

Afterward, based on the implementation synthesized by JLIBSKETCH, SPYRO was able to solve each `Queue` specification-synthesis problem within 40 seconds, yielding the following implementation-agnostic specification $\phi_{sem}^{\text{Queue}}$:

$$\begin{aligned} \text{isEmptyS}(\text{emptyQ}) &= \top & \text{isEmptyQ}(\text{enq}(q, i)) &= \perp \\ \text{sizeQ}(\text{emptyQ}) &= 0 \\ \text{sizeQ}(\text{enq}(q, i)) &= \text{sizeQ}(q) + 1 \\ \text{isEmptyQ}(q) &\rightarrow \text{front}(\text{enq}(q, i)) = i \\ \neg \text{isEmptyQ}(q) &\rightarrow \text{front}(\text{enq}(q, i)) = \text{front}(q) \\ \text{isEmptyQ}(q) &\rightarrow \text{deq}(\text{enq}(q, i)) = q \\ \neg \text{isEmptyQ}(q) &\rightarrow \text{deq}(\text{enq}(q, i)) = \text{enq}(\text{deq}(q), i) \end{aligned}$$

A `TicketVendor` is implemented using a `Queue`, which stores the id numbers of clients

who have reserved tickets. Each issued ticket contains the id of the buyer. The implementation-specific specification $\varphi_{imp}^{\text{TicketVendor}}$ consisted of JLIBSKETCH code with 24 assertions, and contains multiple examples, such as the following (again, x and y are universally quantified integers that are allowed to be in the range 0 to 10):

```

numTicketsRem(preSales(2)) = 2
numWaiting(preSales(2)) = 0
numWaiting(resTicket(preSales(2),x)) = 1
issueTicket(resTicket(preSales(2),x)).owner = x

```

Again, we provided JLIBSKETCH with a fairly complete sketch of the program structure, and JLIBSKETCH was able to synthesize the implementations of all the TicketVendor functions within 10 seconds. For example, the function `prepSales` for `TicketVendor = (numticket : int, qwaiting : Queue)` was synthesized as `prepSales(n : int) := (n, emptyQ)`.

We compared the time needed to synthesize each module from the algebraic specification of the previous module to the time needed to synthesize using the implementation of all previous modules. Synthesizing `Stack` from the specification $\varphi_{sem}^{\text{List}}$ took 3 seconds instead of the 2 seconds needed when the implementation of `List` was provided. Synthesizing `Queue` from the specification $\varphi_{sem}^{\text{Stack}}$ took 188 seconds instead of the 799 seconds needed when the concrete implementations of `Stack` and `List` were provided. Synthesizing `TicketVendor` from the specification $\varphi_{sem}^{\text{Queue}}$ took 7 seconds, but JLIBSKETCH crashed when the concrete implementations of `Queue`, `Stack` and `List` were provided.

Key finding: This experiment shows that modular synthesis takes 1-5 minutes per module, whereas the time taken to synthesize a module from the underlying module implementations grows with the number of modules—to the point where synthesis is unsuccessful with existing tools.

As discussed in §6.1.4, we also synthesized an implementation of `Queue` that uses `List`

instead of two Stacks. The List holds the oldest element of the Queue at its head. The implementation-specific specification $\varphi_{imp}^{\text{Queue (as List)}}$ consisted of JLIBSKETCH code with 19 assertions, including examples similar to those shown at 5 in Fig. 6.2. We used JLIBSKETCH to verify whether the specification $\varphi_{sem}^{\text{Queue}}$ still held true for the new implementation. Because it did (confirmation took <1 second), TicketVendor does not need to be changed to use the Queue (as List) implementation.

6.4.3 Case Studies from Mariano et al. [101]

Our second set of benchmarks is collected from the paper that introduced synthesis from algebraic specifications via JLIBSKETCH [101]. In that work, Mariano et al. used a number of benchmarks that involve two modules—e.g., synthesizing a backend cryptographic component for a tool that brings NuCypher to Apache Kafka, using ArrayList and HashMap as underlying modules. The goal of their paper was to show that in JLIBSKETCH it was easier/faster to synthesize the module at layer 1 when the module of layer 0 was exposed through an algebraic specification (rather than a concrete implementation). The current implementation of MOSSKIT does not support strings, so we used only the benchmarks for which the algebraic specifications for the layer-0 module (i) did not use string operations, and (ii) did not use auxiliary functions that were not in the signature of the module. In total, we considered four layer-0 modules: ArrayList, TreeSet, HashSet, and HashMap. Each JLIBSKETCH benchmark consisted of (i) an algebraic specification of the layer-0 module (written by hand), (ii) a SKETCH-like specification of the layer-1 module, and (iii) a mock implementation of the layer-0 module—i.e., a simplified implementation that mimics the module’s intended behavior (e.g., HashSet is implemented using an array). The mock is not needed by JLIBSKETCH, but allowed Mariano et al. to compare synthesis-from-algebraic-specifications against synthesis-from-mocks [101, §5].

We used these items in a different manner from the JLIBSKETCH experiments. From just the mock implementation of layer 0, we asked MOSSKIT to synthesize a most-precise algebraic specification, which we compared with the algebraic specification manually written by Mariano

et al. From that algebraic specification and the SKETCH-like specification of the layer-1 module, we asked MOSSKIT to synthesize the implementation of layer 1. (The second step essentially replicated the algebraic-synthesis part of the JLIBSKETCH experiments.)

For the layer-0 synthesis step of each benchmark, we synthesized algebraic specifications using grammars similar to the ones used in §6.4.2.

When considering the time taken to synthesize the entire algebraic specification of each module, SPYRO took 626 seconds for $\varphi_{sem}^{ArrayList}$, 54 seconds for $\varphi_{sem}^{HashSet}$, and 1,732 seconds for $\varphi_{sem}^{HashMap}$. Because mock implementations are simplified versions of actual implementations, the mock implementation of TreeSet is identical to the mock implementation of HashSet—i.e., they both represent sets as arrays. Furthermore, the two implementations have the same algebraic specifications—i.e., $\varphi_{sem}^{HashSet} = \varphi_{sem}^{TreeSet}$ —which can thus be synthesized in the same amount of time.

Key finding: For all but two benchmarks, the $\mathcal{L}$ -conjunctions synthesized by MOSSKIT were equivalent to the algebraic properties manually written by Mariano et al. For the mock implementation of HashMap and ArrayList provided in JLIBSKETCH, for specific grammars, MOSSKIT synthesized empty $\mathcal{L}$ -conjunctions (i.e., the predicate *true*) instead of the algebraic specifications provided by Mariano et al.—i.e., $k_1 = k_2 \Rightarrow \text{get}(\text{put}(m, k_1, v), k_2) = v$ and $i = j \Rightarrow \text{get}(\text{set}(l_{in}, i, v), j) = v$, for HashMap and ArrayList, respectively. Upon further inspection, we discovered that JLIBSKETCH’s mock implementation of HashMap was incorrect, and did not satisfy the specification that Mariano et al. gave, due to an incorrect handling of hash collision! After fixing the bug in the mock implementation of HashMap, we were able to synthesize the expected algebraic specification. However, when inspecting the implementation of ArrayList, we found that for this benchmark the implementation was correct but the algebraic specification provided by Mariano et al. was incorrect! After modifying the grammar, we could synthesize the correct algebraic specification $(i = j) \wedge (0 \leq i) \wedge (i \leq \text{sizeL}(l_{in})) \Rightarrow \text{get}(\text{set}(l_{in}, i, v), j) = v$. However, this modification revealed a bug in one of the implementations of HashMap that Mariano et al. had synthesized from the earlier erroneous specification! We discuss this finding

further in the next section.

This finding illustrates how modular system synthesis can help to *identify* and *avoid* bugs in module implementations.

6.4.4 Additional Case Studies Based on Mariano et al. [101]

We noticed that the JLIBSKETCH benchmarks provided an opportunity to build a more complicated benchmark that involved 3 modules (instead of 2). In particular, two of the benchmarks involved synthesizing the implementation of a (layer-1) HashMap module from a (layer-0) algebraic specification of ArrayList. (The two benchmarks synthesized different implementations that handled collisions differently and we refer to the corresponding modules as HashMap1 and HashMap2.) The third benchmark involved synthesizing the implementation of a (layer-2) Kafka from a (layer-1) algebraic specification of HashMap. Thus, we built two 3-layer benchmarks in which the goal was to synthesize Kafka using an implementation of HashMap that used an implementation of ArrayList. For us, each 3-layer benchmark involved four synthesis problems: (1) the algebraic specification $\phi_{sem}^{ArrayList}$ of ArrayList (from the mock); (2) the implementation of either HashMap1 or HashMap2; (3) the algebraic specification of HashMap; and (4) the implementation of Kafka (this part was already synthesized in [101]).

As discussed in the previous section, we identified a bug in the specification $\phi_{sem}^{ArrayList}$ manually provided by Mariano et al., and were able to use MOSSKIT to synthesize a correct algebraic specification—i.e., step (1). For step (2), the implementation synthesized by Mariano et al. for HashMap2 was still correct, and we could also use MOSSKIT to synthesize it from the corrected specification $\phi_{sem}^{ArrayList}$. However, the implementation of HashMap1 synthesized by JLIBSKETCH was incorrect because it depended on the original, erroneous specification $\phi_{sem}^{ArrayList}$ for ArrayList—(1) put could store values to negative indices; and (2) get could search key from incorrect index after rehashing. We manually changed the implementation of the rehashing function in the sketch of HashMap1 to fix the bug, but the change was large enough that we did not attempt to rewrite the program sketch needed to synthesize this specification

(i.e., we manually wrote the implementation of HashMap1 instead of synthesizing it). Synthesis problem (3) is at the heart of handling a multi-module system in a modular fashion: we used MOSSKIT to synthesize algebraic specifications of HashMap1 and HashMap2—in each case, giving MOSSKIT access to the (correct) implementations of HashMap1 and HashMap2 and the (correct) algebraic specification of ArrayList (but not an implementation of ArrayList).

Key finding: MOSSKIT failed to synthesize the same algebraic specification we had obtained for HashMap in §6.4.3 when attempting to synthesize a specification for HashMap1 and HashMap2. When inspecting the synthesized properties, we realized that the algebraic specification $\phi_{sem}^{ArrayList}$ exposed by ArrayList still had a problem! In particular, $\phi_{sem}^{ArrayList}$ was too weak to prove the algebraic specifications needed by HashMap1 and HashMap2—i.e., $\phi_{sem}^{ArrayList}$ did not characterize properties that were needed by HashMap1 and HashMap2 to satisfy the algebraic specification $\phi_{sem}^{HashMap}$. We used Sketch itself to produce a violation of the algebraic specification $\phi_{sem}^{HashMap}$ for HashMap1 under the weaker assumption that ArrayList only satisfied the specification $\phi_{sem}^{ArrayList}$, and used the violations generated by SKETCH to identify what properties we needed to add to strengthen $\phi_{sem}^{ArrayList}$. In particular, $sizeL(ensureCapacity(l_{in}, n)) = sizeL(l_{in})$ and $get(ensureCapacity(l_{in}, n), i) = get(l_{in}, i)$ were added to describe the behavior of ensureCapacity. We were then able to modify the grammar used to synthesize algebraic specifications for $\phi_{sem}^{ArrayList}$ and synthesize the missing property. After obtaining $\phi_{sem}^{ArrayList}$, we successfully synthesized the full algebraic specification for HashMap2 (i.e., $\phi_{sem}^{HashMap}$) and most of the algebraic specification for HashMap1. Because the corrected implementation of HashMap1 was particularly complicated—e.g., each call to put requires rehashing when the load factor is greater than a predefined value—MOSSKIT timed out while synthesizing every property, with the exception of the property $get(emptyMap, k) = err$.

This finding illustrates how modular system synthesis can help identify when module specifications are not strong enough to characterize the behavior of other modules.

6.4.5 Implementation Synthesis with JLibSketch

We present the three inputs provided to JLIBSKETCH to solve the implementation-synthesis problem for Queue: (i) a program sketch describing the search space of possible programs (Fig. 6.6), (ii) an implementation-agnostic specification ϕ_{sem}^{Stack} of the Stack module in the form of rewrite rules (Fig. 6.7), and (iii) an implementation-specific specification ϕ_{imp}^{Queue} of the Queue module in the form of assertions (Fig. 6.8).

```

1 public void enqueue(int x) {
2     Stack st_in = this.st_in;
3     Stack st_out = this.st_out;
4     assume !st_out.isEmpty() ∨ st_in.isEmpty();
5
6     if (genGuard(st_in, st_out)) {
7         st_in = genStack2(st_in, st_out, x);
8         st_out = genStack2(st_in, st_out, x);
9     } else {
10        st_in = genStack2(st_in, st_out, x);
11        st_out = genStack2(st_in, st_out, x);
12    }
13
14    assert !st_out.isEmpty() ∨ st_in.isEmpty();
15    this.st_in = st_in;
16    this.st_out = st_out;
17 }
18
19 private static void rev(Stack in, Stack out) {
20     while(!in.isEmpty()) {
21         out.push(in.top());
22         in.pop();
23     }
24 }
25
26 public void dequeue() {
27     Stack st_in = this.st_in;
28     Stack st_out = this.st_out;
29     assume !st_out.isEmpty() ∨ st_in.isEmpty();
30
31     st_in = genStack1(st_in, st_out);
32     st_out = genStack1(st_in, st_out);
33
34     if (genGuard(st_in, st_out)) {
35         rev(st_in, st_out);
36     }
37
38     this.st_in = st_in;
39     this.st_out = st_out;
40     assert !st_out.isEmpty() ∨ st_in.isEmpty();
41 }

```

Figure 6.6. JLIBSKETCH sketch of enq and deq. Line 5, 14, 29 and 40 assert the implementation-specific property $\text{isEmptyS}(st_{out}) \rightarrow \text{isEmptyS}(st_{in})$. JLIBSKETCH generates an expression to fill in each occurrence of the generators `genStack1`, `genStack2` and `genGuard`—the reader can think of each of these generators as being grammars from which JLIBSKETCH can pick an expression. For these generators, expressions can be variables or function calls to functions of the appropriate type—e.g., `genStack1` can generate expressions such as `st_in`, `st_out`, `st_in.pop()`, `st_out.pop()`, etc.

6.4.6 Limitations of MOSSKIT

JLIBSKETCH and SPYRO represent the algebraic specifications of modules as rewrite rules for algebraic datatypes (ADTs). Reasoning about ADTs is a challenging problem, and to the best of our knowledge, SKETCH and JLIBSKETCH are only frameworks capable of handling problems involving ADTs effectively. Therefore, MOSSKIT uses them as the underlying solver and inherits limitations of SKETCH.

The primary limitation of MOSSKIT is its bounded soundness guarantee. SKETCH ensures soundness only for a bounded number of loop/recursion unrollings, and bounded input sizes. Verifying the unbounded correctness of the synthesized programs poses a significant challenge, as semantics of lower-level modules are represented as rewrite rules on ADTs. As a future direction, we plan to integrate MOSSKIT with verifiers such as Dafny to perform full verification, as was done in [115] for the properties synthesized by SPYRO. However, it is worth noting that MOSSKIT has already been useful in finding bugs in existing implementations: specification synthesis has helped find implementation errors in the case studies of Mariano et al. [101], as demonstrated in §6.4.3 and §6.4.4.

Although the case studies in §6.4.2 and reference [101] show satisfactory performance of SKETCH for most problems, scalability issues persist. In particular, unrolling nested loops significantly increases the number of holes of the SKETCH problem, which increases the problem’s difficulty.

Besides the limitations inherited from SKETCH, MOSS has a specific requirement for the system’s modular structure, which should be a directed acyclic graph (DAG)—i.e., the implementation-agnostic specifications of all dependent modules must be provided to synthesize a particular module. MOSS addresses the challenges in writing accurate specifications by using the synthesis of implementation-agnostic specifications. However, in this approach one needs to synthesize all dependent modules and their specifications before attempting to synthesize a new module. Alternatively, to synthesize higher-level modules without the lower-level implementa-

```

1 @rewriteClass
2 class Stack {
3     @alg
4     Stack push(int x);
5
6     @alg
7     @pure
8     int top();
9
10    @alg
11    Stack pop();
12
13    @alg
14    @pure
15    int size();
16
17    @alg
18    @pure
19    boolean isEmpty();
20
21    rewrite int size(Stack Stack()) { return 0; }
22    rewrite int size(
23        Stack push!(Stack st, int x)) {
24        return size(st) + 1;
25    }
26    rewrite boolean isEmpty(Stack Stack()) {
27        return true; }
28    rewrite boolean isEmpty(
29        Stack push!(Stack st, int x)) {
30        return false;
31    }
32    rewrite int top(
33        Stack push!(Stack st, int x)) {
34        return x; }
35    rewrite Stack pop!(
36        Stack push!(Stack st, int x)) {
37        return st; }
38 }

```

Figure 6.7. JLIBSKETCH rewrite class Stack for the synthesis of Queue. Lines 3-19 are function signatures of Stack operations, and lines 21-37 are implementation-agnostic properties φ_{sem}^{Stack} of Stack. The constructor Stack() plays the same role as what was referred to in the body of the paper as emptyS.


```

1 harness void test(int x, int y, int z) {
2     assume x ≠ y ∧ x ≠ z ∧ y ≠ z;
3     assume x > 0 ∧ x < 10;
4     assume y > 0 ∧ y < 10;
5     assume z > 0 ∧ z < 10;
6
7     Queue queueUnderTest = Queue.empty();
8     // size_q(empty_q) = 0
9     assert queueUnderTest.size() = 0;
10    // is_empty_q(empty_q) = true
11    assert queueUnderTest.isEmpty();
12
13    queueUnderTest.enqueue(x);
14    // size_q(enqueue(empty_q,x)) = 1
15    assert queueUnderTest.size() = 1;
16    // front(enqueue(empty_q,x)) = x
17    assert queueUnderTest.front() = x;
18    // is_empty_q(enqueue(empty_q,x)) = false
19    assert !queueUnderTest.isEmpty();
20
21    queueUnderTest.enqueue(y);
22    assert queueUnderTest.size() = 2;
23    assert queueUnderTest.front() = x;
24    assert !queueUnderTest.isEmpty();
25
26    queueUnderTest.enqueue(z);
27    assert queueUnderTest.size() = 3;
28    assert queueUnderTest.front() = x;
29    assert !queueUnderTest.isEmpty();
30
31    queueUnderTest.dequeue();
32    assert queueUnderTest.size() = 2;
33    assert queueUnderTest.front() = y;
34    assert !queueUnderTest.isEmpty();
35
36    queueUnderTest.dequeue();
37    assert queueUnderTest.size() = 1;
38    assert queueUnderTest.front() = z;
39    assert !queueUnderTest.isEmpty();
40 }

```

Figure 6.8. JLIBSKETCH harness corresponding to the implementation-specific specification ϕ_{imp}^{Queue} for Queue operations. Lines 2-5 specify a range of integers to be tested, and lines 7-39 checks the behavior of various functions using a specific test cases. We include comments describing what property is being tested.

tions, the user can manually supply the implementation-agnostic specifications of the lower-level modules.

6.5 Related Work

A problem related to ours is that of component-based synthesis (CBS), where the goal is *assembling* pre-existing components/APIs to generate more complex programs. Many existing approaches for solving CBS problems scale reasonably well [49, 137, 141], but require the individual components to be executable. In our setting, this approach is not possible because the details of lower-level components (e.g., how a Stack is implemented) need not be observable.

A few tools have abstracted components and modules using specifications. JLIBSKETCH [101] uses algebraic properties to represent the semantics of modules and is a key component of our implementation. (CL)S [14] and APIphany [64] use types to represent the behavior of components and can be used in tandem with specialized type-directed synthesizers. The key differences between our work and these tools is that MOSS provides two well-defined synthesis primitives that support composing multiple modules, rather than synthesizing just one implementation for one module. Furthermore, the aforementioned types are limited in how they can represent relations between multiple components in an implementation-agnostic way, thus making us opt for algebraic specifications.

Many synthesis tools perform some kind of “compositional” synthesis by breaking an input specification into sub-specifications that are used to separately synthesize sub-components of a target program [126, 3]. This notion of “compositionality” is orthogonal to ours, and is more of a divide-and-conquer approach to solving *individual* synthesis problems. MOSS can make use of such a divide-and-conquer approach when synthesizing a module’s implementation.

For the task of synthesizing an algebraic specification, MOSSKIT uses SPYRO. Besides SPYRO, there are a number of works about discovering specifications from code, based on both static techniques [50, 142] and dynamic techniques [47, 68]. The static approaches mostly

target predicates involving individual functions (instead of algebraic properties and equalities involving multiple functions). The dynamic techniques are flexible and can identify algebraic specifications (e.g., for Java container classes [68]), but require some “bootstrapping” inputs, and only guarantee soundness with respect to behaviors that are covered by the tests that the inputs exercise.

6.6 Conclusion

This chapter presented MOSS, a framework for modular system synthesis that makes information hiding part of the synthesis process. MOSS organizes system construction around two recurring problems: synthesizing a module implementation from its own requirements and the implementation-agnostic specifications of lower-level modules, and synthesizing an implementation-agnostic specification from a completed module for use by higher layers. By exposing contracts rather than accumulated implementation details, the framework keeps each synthesis task focused on one module and allows lower-level implementations to change without requiring higher-level modules to reason about their internal representations.

The main contribution of this chapter is not an immediate solution to the modular-synthesis problem, but rather the identification of two key synthesis primitives that are required to realize MOSS in practice: 1) synthesis from an implementation-agnostic specification, and 2) synthesis of an implementation-agnostic specification.

We implemented these two synthesis primitives in MOSSKIT by combining JLIBSKETCH [101], which synthesizes implementations from algebraic specifications, with SPYRO from Chapter 4. The ticket-vendor and JLIBSKETCH case studies provide evidence that this decomposition can improve scalability: synthesis through algebraic interfaces remained tractable in cases where exposing all lower-level implementations made synthesis substantially slower or unsuccessful. Specification synthesis also exposed an incorrect mock implementation, an incorrect manually written specification, and an interface that was too weak to justify behavior

required by a client. These findings illustrate a second benefit of the framework: synthesizing both sides of a module boundary can reveal mismatches that implementation synthesis alone may preserve.

While our tool implements both of these primitives using tools based on SKETCH (thus inheriting its limitations), an interesting research directions is whether other synthesis approaches (enumeration, CEGIS, etc.) can be extended to handle our synthesis problems, perhaps by leveraging the popular egg framework [158] which allows one to reason about equivalence of terms with respect to a term-rewriting system—i.e., our algebraic specifications.

MOSSKIT is nevertheless a proof of concept rather than a complete solution to large-scale modular synthesis. Because its underlying solvers use bounded reasoning, the synthesized implementations and specifications are guaranteed only within configured bounds, while nested loops, recursion, and large specification grammars remain difficult. Future work should also combine modular synthesis with unbounded verification, develop synthesis engines that reason more directly and efficiently over algebraic interfaces, and automate the discovery and strengthening of contracts needed by higher-level clients. These advances would test how far the two MOSS synthesis primitives can support deeper and larger software systems.

Acknowledgments

Chapter 6, in full, is based on the paper from Kanghee Park, Keith J.C. Johnson, Loris D’Antoni, and Thomas Reps “Modular System Synthesis” published in 2023 Formal Methods in Computer-Aided Design, pp. 257-267. IEEE, 2023. The dissertation author is the primary investigator and author of this paper.

Chapter 7

Extracting Reusable Proof Structure

Formal verification proves that a model of an implementation satisfies a specification under given semantics and explicit assumptions. Yet, even with formal specifications in place, constructing proofs remains labor-intensive and scales poorly to large systems.

Automated reasoning tools mitigate this challenge in complementary ways. SMT solvers like Z3 and cvc5 [34, 12], as well as verification-oriented languages like Dafny [93] and Verus [90], automate significant classes of verification conditions. However, because this automation is bounded by underlying decision procedures, complex developments still demand modular decomposition, auxiliary invariants, and expert human direction.

Large language models offer a novel form of automation by synthesizing proof scripts or proof terms for interactive proof assistants such as Lean [109] and Rocq [149]. In this setting, the proof assistant acts as a strict gatekeeper: a generated proof is accepted only when validated by the underlying checker. This hybrid strategy bridges the gap between neural generation and mathematical rigor, leveraging the generative strength of LLMs while relying on proof assistants to ensure soundness.

However, machine-generated proofs can be unnecessarily concrete. They may contain long sequences of low-level steps, specialized constants, and computations that solve one theorem without identifying the reusable lemma underlying the argument. Such proofs remain checkable, but they are harder to inspect, consume more generation context, and provide less guidance

for later proof search. A useful post-processing step is therefore to turn a checked proof into a shorter theorem whose parameters and premises expose the proof idea at an appropriate level of abstraction.

This chapter studies that post-processing problem as *proof abstraction*. Given a checked source proof and a user-supplied generalized target type, we replace selected source subterms by metavariables and use type inference and unification to determine the types of the resulting holes. Those hole types are explicit obligations sufficient to replay the selected proof structure.

As an example, consider a formal proof establishing the irrationality of $\sqrt{2}$. The concrete proof assumes $\sqrt{2} = a/b$ for coprime natural numbers a and b , derives $2b^2 = a^2$, and concludes that 2 divides both a and b , contradicting their coprimality. Abstracting the constant 2 to a variable p exposes a critical proof dependency: inferring $p \mid a$ from $p \mid a^2$. While the concrete proof dispatches this step via computation or parity arguments specific to 2, abstracting the proof term exposes a hole whose type becomes the logical premise `Irreducible  $p$` .

The chapter formalizes abstractions of dependently typed proof terms and a preorder that captures refinements known to preserve provability. It then presents a constructor-guided, top-down enumerator. Refinements certified to preserve provability are normalized without branching, whereas refinements that may eliminate valid completions are explored as separate candidates. Matching environments record how every frontier metavariable reconstructs the original proof, and a metavariable dependency graph identifies groups of holes whose types constrain one another.

7.1 Background

This section provides the core proof assistant terminology used throughout this chapter.

7.1.1 The Calculus of Constructions

The Calculus of Constructions (CoC), introduced by Coquand and Huet [29], combines higher-order typed λ -calculus with dependent function types, allowing types to depend on

$$\begin{array}{c}
\frac{}{[] \text{ valid}} \quad \frac{\Gamma \vdash A : \kappa}{\Gamma[x : A] \text{ valid}} \quad \frac{\Sigma(c) = A \quad \Gamma \text{ valid}}{\Gamma \vdash c : A} \quad \frac{\Gamma(x) = A \quad \Gamma \text{ valid}}{\Gamma \vdash x : A} \\
\\
\frac{\Gamma \text{ valid}}{\Gamma \vdash \kappa_n : \kappa_{n+1}} \quad \frac{\Gamma \vdash A : \kappa_m \quad \Gamma[x : A] \vdash B : \kappa_n}{\Gamma \vdash \Pi x : A. B : \text{Type}_{\text{imax}(m,n)}} \\
\\
\frac{\Gamma[x : A] \vdash e : B}{\Gamma \vdash \lambda x : A. e : \Pi x : A. B} \quad \frac{\Gamma \vdash M : \Pi x : A. B \quad \Gamma \vdash N : A}{\Gamma \vdash MN : [N/x]B} \\
\\
\frac{\Gamma \vdash M : A \quad \Gamma \vdash B : \kappa \quad \Gamma \vdash A \equiv B}{\Gamma \vdash M : B}
\end{array}$$

Figure 7.1. The Calculus of Constructions with Non-cumulative Universes. $\text{imax}(m, n)$ determines the universe level of a dependent function type $\Pi x : A. B$, evaluating to level 0 (Prop) if the codomain B is in Prop, and to $\max(m, n)$ otherwise, implementing the impredicativity of propositions.

terms and propositions to be represented by types. CoC and extensions such as the Calculus of Inductive Constructions are foundational models for dependently typed proof assistants, although the kernels of systems such as Lean and Rocq include additional features and differ in details such as inductive types, universes, quotient constructions, and conversion [109, 149].

For exposition, we use the simplified CoC-style core in Figure 7.1. It has a hierarchy of non-cumulative universes and dependent Π -types. A dependent type $\Pi x : A. B$ is the type of functions that map each $a : A$ to a term of type $B[a/x]$. The ordinary function type $A \rightarrow B$ is the special case in which B does not depend on x . When B is proposition-valued, the same type is commonly written $\forall x : A. B$.

In dependently typed proof assistants, equality is fundamentally divided into *definitional equality* and *propositional equality*. Definitional equality ($\equiv$) holds when two terms are symbolically identical up to computation, such as unfolding definitions or evaluating functions (e.g., $1 + 1 \equiv 2$). Because these terms reduce to the exact same normal form, the proof assistant’s kernel checks definitional equality automatically without requiring explicit proof steps from the user. In contrast, propositional equality ($=$) represents a formal assertion within the logic that two terms are equal, which may not be reducible by computation alone (e.g., $x + y = y + x$). Proving a propositional equality requires an explicit proof term or tactic sequence. While every definitional

equality trivially implies a propositional one, propositional equalities generally require explicit logical reasoning, making them the primary subject of formal proofs.

We omit inductive types and their eliminators from the core syntax because the abstraction algorithm operates on the resulting proof-term structure rather than on any particular collection of inductive definitions.

Definition 7.1.1 (Term). *A term t in the core calculus is generated by*

$$t ::= x \mid c \mid tt \mid \lambda x : t. t \mid \Pi x : t. t \mid \kappa_n, \quad (7.1)$$

where x is a bound or local variable, c is a global constant, and κ_n is a universe.

During elaboration and proof construction, theorem provers also use metavariables, written $?m$. A declaration $[\Gamma, ?m : \tau]$ records the local context Γ on which $?m$ may depend and its expected type τ . We write $\langle \Delta, t \rangle$ for a term together with an ordered, dependency-closed metavariable telescope $\Delta = [\Gamma_1, ?m_1 : \tau_1] \cdots [\Gamma_k, ?m_k : \tau_k]$, which we refer to as an *abstract term* (or *aterm*). A closed term is the special case $\langle \cdot, t \rangle$.

The typing rules are summarized in Figure 7.1. A term is well typed when $\Gamma \vdash t : \tau$ for some τ . An aterm is well typed when its metavariable telescope is well formed and the corresponding judgment $\Gamma; \Delta \vdash t : \tau$ is derivable. We suppress the external context Γ when it is empty.

Definition 7.1.2 (Metavariables of a term). *The set $MV(t)$ of metavariables occurring syntactically in a term is defined by*

- $MV(x) = MV(c) = MV(\kappa_n) = \emptyset$
- $MV(?m) = \{?m\}$
- $MV(t_1 t_2) = MV(t_1) \cup MV(t_2)$
- $MV(\lambda x : t_1. t_2) = MV(t_1) \cup MV(t_2)$
- $MV(\Pi x : t_1. t_2) = MV(t_1) \cup MV(t_2)$.

The telescope Δ additionally contains metavariables that occur in the declarations of metavariables.

ables reachable from $MV(t)$; this dependency closure is needed in a dependent type theory.

Definition 7.1.3 (Metavariable substitution). *Let $\Delta = [\Gamma_1, ?m_1 : \tau_1] \cdots [\Gamma_n, ?m_n : \tau_n]$ and let Δ' be a target metavariable telescope. After systematic renaming, assume that the domains of Δ and Δ' are disjoint. A valid substitution $\eta : \Delta \Rightarrow \Delta'$ is a sequence $(t_1, \dots, t_n)$ such that, for every i ,*

$$\eta_{<i}(\Gamma_i); \Delta' \vdash t_i : \eta_{<i}(\tau_i),$$

where $\eta_{<i} = [t_{i-1}/?m_{i-1}] \cdots [t_1/?m_1]$. Its action on a term is the capture-avoiding substitution

$$\eta(t) = [t_n/?m_n] \cdots [t_1/?m_1]t.$$

Mappings omitted from a displayed partial substitution are identities, up to fresh renaming.

We use the standard right-to-left convention for composition: $(\theta \circ \eta)(t) = \theta(\eta(t))$.

Unless a conversion relation such as $\equiv_{\beta\eta}$ is written explicitly, equality between reconstructed terms means syntactic equality up to α -renaming.

7.1.2 Proof Terms and Provability

The Curry–Howard correspondence [74] identifies propositions with types and proofs with inhabitants of those types. Under this correspondence, checking a proof reduces to checking that its proof term has the theorem’s type according to the rules of the underlying type theory. For example, a proof of $A \rightarrow B$ is a function that maps proofs of A to proofs of B , and a proof of $\forall x : A. P(x)$ is a dependent function of type $\Pi x : A. P(x)$. Products, sums, and inductive types provide the usual encodings of conjunction, disjunction, and data propositions in richer calculi.

An aterm $\langle \Delta, t \rangle$ represents a partial term: each metavariable declaration in Δ is an unresolved obligation with a specified local context and expected type.

Definition 7.1.4 (Completeness). *Let $\Gamma; \Delta \vdash t : \tau$. The aterm $\langle \Delta, t \rangle$ is completable in Γ if there exists a valid ground substitution $\eta : \Delta \Rightarrow \cdot$ such that $\Gamma \vdash \eta(t) : \eta(\tau)$. When τ is a proposition,*

we also call the aterm provable.

Instantiation is monotone in one direction: if $\eta(a)$ is completable, then a is completable, because a completion η' of $\eta(a)$ yields the composite completion $\eta' \circ \eta$ of a . The converse does not hold for an arbitrary substitution: committing to an unsuitable instance may destroy all completions.

Definition 7.1.5 (Safe substitution). *A valid substitution η is safe for an aterm a in context Γ when it preserves completability:*

$$a \text{ completable in } \Gamma \implies \eta(a) \text{ completable in } \Gamma.$$

Proposition 7.1.6 (Prefix closure of safety). *If $\eta_2 \circ \eta_1$ is safe for an aterm a , then η_1 is safe for a .*

Proof. Assume that a is completable. Safety of $\eta_2 \circ \eta_1$ implies that $\eta_2(\eta_1(a))$ is completable. The substitution η_2 , followed by that completion, is therefore a completion of $\eta_1(a)$. Hence $\eta_1(a)$ is completable. $\square$

7.1.3 Type Inference by Unification

In simply typed and Hindley–Milner-style calculi, constraint-based type inference assigns fresh type variables to unknown types, generates equations from the typing rules, and solves those equations by unification. For an application $e_1 e_2$, for example, if e_1 has inferred type τ_1 and e_2 has inferred type τ_2 , inference introduces a fresh result type α and generates the constraint $\tau_1 \equiv \tau_2 \rightarrow \alpha$. For a sound and complete constraint-generation procedure, the source term is typable exactly when the generated first-order type constraints are unifiable. This is the basis of principal type inference in systems such as Hindley–Milner [69, 105].

Definition 7.1.7 (Unifier and most general unifier). *Fix the equality used by a unification problem (e.g., definitional equality). A substitution θ is a unifier of terms t_1 and t_2 when $\theta(t_1)$ and $\theta(t_2)$*

are equal in that theory. A unifier θ is a most general unifier (MGU) when every other unifier θ' is an instance of θ : there exists a substitution ϕ such that $\theta' = \phi \circ \theta$.

First-order syntactic unification is decidable. Two arbitrary first-order terms need not be unifiable—terms with different rigid head symbols are a simple counterexample. If a finite first-order unification problem is solvable, however, it has an MGU, unique up to the usual renaming equivalence [132].

The situation changes in higher-order settings. General typed λ -term unification may have multiple incomparable solutions and is not decidable in unrestricted form; second-order unification is already undecidable [78, 60]. Decidable and unitary fragments therefore restrict how metavariables may occur. Miller’s higher-order patterns provide such a fragment for the simply typed λ -calculus [104], and related pattern-unification techniques have been developed for the Calculus of Constructions [120].

7.2 Overview

We first describe the approach using a simple lemma stating that two natural numbers with the common divisor 2 are not coprime (Figure 7.2a).

7.2.1 Generalization by Abstraction and Unification

A direct way to seek a more reusable theorem is to replace selected constants or proof subterms by parameters. The theorem in Figure 7.2a mentions natural numbers, divisibility, equality, and the numeral 2. Replacing 2 by an unconstrained variable, however, makes the statement false when that variable is 1. The source proof reveals where the special property of 2 is used: the contradiction closes with a proof that $2 \neq 1$.

Suppose that we want to generalize `even_numbers_not_coprime` while retaining the conclusion that `a` and `b` are not coprime. We first inspect a simplified proof term corresponding to the tactic proof. Figure 7.2b uses `absurd`, of type $P \rightarrow \neg P \rightarrow Q$, to combine the equation

```

1 def coprime (a b : Nat) : Prop :=  $\forall n : \text{Nat}, n \mid a \rightarrow n \mid b \rightarrow n = 1$ 
2 theorem even_numbers_not_coprime (a b : Nat) (ha : 2  $\mid$  a) (hb : 2  $\mid$ 
   b)
3   :  $\neg$  coprime a b := by
4   intro hcp
5   let hcontra := hcp 2 ha hb
6   contradiction

```

(a) Tactic-style proof of `even_numbers_not_coprime`.

```

1 fun (a b : Nat) (ha : 2  $\mid$  a) (hb : 2  $\mid$  b) (hcp : coprime a b) =>
2   let hcontra := hcp 2 ha hb;
3   absurd hcontra (of_decide_eq_false (Eq.refl (decide (2 = 1))))

```

(b) A simplified proof term for `even_numbers_not_coprime`.

```

1 fun (a : ?m1) (b : ?m2) (ha : ?m3 a) (hb : ?m4 b) (hcp : coprime a
   b) =>
2   let hcontra := hcp ?m5 ha hb;
3   absurd hcontra ?m6

```

(c) Abstract proof term before type unification.

```

1 fun (a b : Nat) (ha : ?m5  $\mid$  a) (hb : ?m5  $\mid$  b) (hcp : coprime a b) =>
2   let hcontra := hcp (?m5 : Nat) ha hb;
3   absurd hcontra (?m6 :  $\neg$ (?m5 = 1))

```

(d) Abstract proof term after type unification.

```

1 fun (a b x0 : Nat) (x1 :  $\neg$ (x0 = 1))
2   (ha : x0  $\mid$  a) (hb : x0  $\mid$  b) (hcp : coprime a b) =>
3   let hcontra := hcp x0 ha hb;
4   absurd hcontra x1

```

(e) Abstract proof after eliminating metavariables.

Figure 7.2. Generalization of `even_numbers_not_coprime` by subterm abstraction and unification.

produced from coprimality with a proof of its negation. Variables prefixed by $?m$ in the subsequent listings are metavariables.

The initial abstraction in Figure 7.2c replaces several concrete types and subterms by metavariables. Many assignments to those metavariables are ill typed. Elaborating the partial term generates constraints from their uses. In line 2, for example, `hcp` has type `coprime a b`, whose definition is shown in line 1. Applying `hcp` determines that $?m5$ must have type `Nat`, and that the assumptions `ha` and `hb` must have types $?m5 \mid a$ and $?m5 \mid b$, respectively. The resulting typed partial term is shown in Figure 7.2d.

After solving these typing constraints, two unresolved metavariables remain: $?m5$, representing the common divisor, and $?m6$, representing a proof that this divisor is not 1. Generalizing the unresolved metavariables as function parameters yields Figure 7.2e. The resulting theorem states that any $x0$ dividing both inputs prevents coprimality provided that $x0 \neq 1$.

7.2.2 Choosing an Abstraction Boundary

The main search problem is deciding which source subterms to replace by metavariables. Different boundaries can all produce type-correct theorems, but they expose different interfaces and retain different implementation details.

Consider the second argument to `absurd` in line 3 of Figure 7.2b: `of_decide_eq_false (Eq.refl (decide (2 = 1)))`. In Figure 7.2c, the entire proof of $2 \neq 1$ is replaced by the single metavariable $?m6$.

At a finer boundary, one might replace only the numeral 2. The retained proof fragment would be

```
of_decide_eq_false (Eq.refl (decide (?m7 = 1))).
```

For this expression to elaborate by definitional reduction, $?m7$ must be instantiated by a term for which the equality test with 1 computes to `false`; a rigid variable leaves the computation stuck. Moreover, exact reconstruction of the source proof fixes this metavariable to 2 up to definitional

equality. This boundary therefore preserves the concrete computation rather than producing the desired parameterized theorem.

At an intermediate boundary, one could abstract `Eq.refl (decide (?m7 = 1))` while retaining `of_decide_eq_false`. The exposed obligation then has the form `decide (x0 = 1) = false`. For natural-number equality this proposition is logically equivalent to $x0 \neq 1$, but it exposes the particular decision procedure and representation used by the source proof.

Abstracting the whole proof subterm removes that computational detail. The expected type of `?m6` is inferred as $\neg(x0 = 1)$, producing the clean premise shown in Figure 7.2e. Among the three boundaries just considered, this is the most abstract interface. The chapter’s algorithm searches systematically over such constructor-guided boundaries. Unification determines the types forced by each selected proof skeleton; when the generated problem lies in a unitary fragment, an MGU gives a principal typing solution. Note that this does not imply that the exposed premise is globally logically weakest across all possible proofs.

7.3 Abstraction Synthesis

Having illustrated the workflow informally, we now define the abstraction relation and the synthesis objective. The distinction between source reconstruction up to definitional equality, type correctness, and semantic safety is important: a candidate may reconstruct the source proof and typecheck at the generalized target without being comparable to another candidate by a completability-preserving substitution.

Definition 7.3.1 (Abstraction). *Let a_1 and a_2 be well-typed aterms. We say that a_2 is an abstraction of a_1 , with witness η , when η is a valid metavariable substitution and $a_1 \equiv_{\beta\eta} \eta(a_2)$. Thus a_1 is an instance of a_2 and a_2 is at least as general as a_1 .*

Intuitively, an abstraction is a term context whose holes can be filled to recover the source term. The equality in Definition 7.3.1 is definitional equality in the core calculus, not merely propositional equality: the algorithm records a concrete substitution whose result is convertible

to the checked source proof. A reconstruction witness need not be globally unique, especially in higher-order settings or when different fresh names and convertible terms represent the same template. The algorithms therefore carry an explicit matching environment instead of relying on uniqueness.

The objective of this chapter is to derive abstractions of an existing proof that can be checked at a more general target type and reused as lemmas.

Example 22. *Revisit the example from Section 7.2.1. The concrete theorem has type*

$$\Pi a : \text{Nat}. \Pi b : \text{Nat}. (2 \mid a) \rightarrow (2 \mid b) \rightarrow \neg \text{coprime } a \ b. \quad (7.2)$$

We want to replace the fixed common divisor 2 by an arbitrary natural number k . The corresponding target interface begins with

$$\Pi a : \text{Nat}. \Pi b : \text{Nat}. \Pi k : \text{Nat}. (k \mid a) \rightarrow (k \mid b) \rightarrow \neg \text{coprime } a \ b, \quad (7.3)$$

but this formula is not valid for $k = 1$. A synthesized proof may therefore extend the target context with an additional premise such as $k \neq 1$.

To separate parameters from the body of the theorem, represent the concrete body as $\langle \Delta, \tau \rangle$, where

$$\Delta = [\text{?}a : \text{Nat}][\text{?}b : \text{Nat}], \quad \tau = (2 \mid \text{?}a) \rightarrow (2 \mid \text{?}b) \rightarrow \neg \text{coprime } \text{?}a \ \text{?}b.$$

Use fresh metavariables for the generalized body $\langle \hat{\Delta}, \hat{\tau} \rangle$:

$$\hat{\Delta} = [\hat{\text{?}}a : \text{Nat}][\hat{\text{?}}b : \text{Nat}][\text{?}k : \text{Nat}], \quad \hat{\tau} = (\text{?}k \mid \hat{\text{?}}a) \rightarrow (\text{?}k \mid \hat{\text{?}}b) \rightarrow \neg \text{coprime } \hat{\text{?}}a \ \hat{\text{?}}b.$$

The valid substitution

$$\rho = [\text{?}a / \hat{\text{?}}a, \text{?}b / \hat{\text{?}}b, 2 / \text{?}k]$$

witnesses that the generalized type abstracts the concrete type.

A useful synthesized result has a context of the form $\Delta^* = \hat{\Delta}, \Omega$, where Ω contains additional premises; a term t^* such that the original proof is an instance of t^* ; and a typing derivation $\Delta^* \vdash t^* : \hat{\tau}$. In the running example, Ω contains a term of $?k \neq 1$.

Under Definition 7.3.1, every proof has the degenerate abstraction consisting of one metavariable for the entire proof. This template is maximally general but not useful: applying it to a new theorem requires supplying the whole proof again. Refining a template makes more of the source structure explicit, but an arbitrary refinement can eliminate completions. We use safe substitutions to distinguish refinements that preserve completability.

Definition 7.3.2 (Safe-instantiation preorder). *Let a_2 be an abstraction of a_1 with witness η . We write $a_1 \preceq_\eta a_2$ when η is safe for a_2 , and write $a_1 \preceq a_2$ when the witness is unimportant. Thus the left-hand side is at least as concrete as the right-hand side, and the two terms are equi-completable under the safety assumption. We write $a_1 \prec a_2$ when $a_1 \preceq a_2$ but $a_2 \not\preceq a_1$.*

The identity substitution is safe, and safe instantiation witnesses compose, so $\preceq$ is reflexive and transitive. It becomes a partial order after quotienting by the induced equivalence $a \simeq b$ iff $a \preceq b$ and $b \preceq a$. The ideal synthesis objective can now be stated as follows.

Given a checked source aterm $\langle \Delta, t \rangle$ of type τ , a target type $\langle \hat{\Delta}, \hat{\tau} \rangle$, and a witness ρ that the target type abstracts the source type, enumerate one representative of every $\simeq$ -class of aterms $\langle \Delta^*, t^* \rangle$ satisfying:

1. **Context extension:** $\Delta^* = \hat{\Delta}, \Omega$ for some well-formed telescope Ω ;
2. **Abstraction:** the source proof is an instance of the candidate, with the target parameters instantiated by ρ ;
3. **Target-typing:** $\Delta^* \vdash t^* : \hat{\tau}$; and
4. **Minimality:** there is no other candidate a' satisfying (1)–(3) with $a' \prec \langle \Delta^*, t^* \rangle$.

This semantic objective is not directly computable in unrestricted dependent type theory because it quantifies over completability and arbitrary safe substitutions. The remainder of the chapter therefore implements a restricted, explicit search: candidates are generated by constructor-guided refinements, and safety is under-approximated by certified rules and component-completion certificates.

7.4 Top-Down Enumeration of Abstract Proof Terms

Choosing abstraction boundaries by hand is difficult: retaining more source structure reduces the obligations exposed to the user, but it can also preserve accidental details that prevent reuse. We therefore use a top-down search procedure, motivated by top-down library learning [21], that begins with one metavariable for the entire proof and incrementally exposes source constructors.

The worklist Q stores candidate abstractions together with matching environments. Each candidate is first saturated by refinements that are known to preserve completability. The resulting safe-normal form is recorded as one representative, and constructor-guided refinements whose safety is not established are placed on separate worklist branches. The algorithm enumerates the refinements generated by its explicit schemas; it does not range over arbitrary proof terms.

7.4.1 Safe Metavariable Refinements

The semantic preorder $\preceq$ is defined through completability and is not decidable in general. We therefore underapproximate semantic safety with certified one-step rules. Saturating these rules avoids branching on choices that cannot remove all completions. Because the rules are only an underapproximation, safe normality is a rule-relative property: by itself it does not imply global minimality or unrestricted completeness. The precise soundness, termination, and worklist-exhaustiveness guarantees are stated in Theorems 7.4.4, 7.4.5 and 7.4.6.

Matching environments and certified refinements.

A matching environment M contains one quadruple $(\Gamma, ?m, \tau, s)$ for each unresolved frontier metavariable $?m$ of the current candidate. It induces a source-matching map η_M with $\eta_M(?m) = s$. This map need not be a well-typed substitution in the generalized target context; it records only which source subterm each frontier hole represents.

Fix a substitution ρ witnessing that the target type $\langle \hat{\Delta}, \hat{\tau} \rangle$ abstracts the source type, and make the target metavariables rigid during enumeration. We write $\hat{\eta}_M = \rho \uplus \eta_M$ for the combined source interpretation. A state reconstructs the source when $\hat{\eta}_M(a) \equiv \langle \Delta, t \rangle$. During search, the rigidified target telescope is treated as an external context. We write $\hat{\Delta}; \Omega \vdash u : \tau$ when Ω is the current search telescope; after the target variables are restored, the output telescope is $\hat{\Delta}, \Omega$.

Definition 7.4.1 (Certified refinement). *Let $(a_{\text{pre}}, M_{\text{pre}})$ and $(a_{\text{post}}, M_{\text{post}})$ be refinement states. A valid metavariable substitution σ is a refinement witness from the former state to the latter when*

$$\begin{aligned} a_{\text{post}} &= \sigma(a_{\text{pre}}), \\ \eta_{M_{\text{pre}}} &= \eta_{M_{\text{post}}} \circ \sigma \quad \text{on } MV(a_{\text{pre}}). \end{aligned}$$

The second equality says that refining a frontier metavariable and then matching the newly introduced metavariables to the original proof reconstructs exactly the same source subterm as before.

Nondependent function application.

The implemented safe application schema replaces a hole $?m : \tau$ by $?m_{\text{fun}} ?m_{\text{arg}}$ and introduces a fresh type metavariable $? \alpha$ with

$$?m_{\text{arg}} : ? \alpha, \quad ?m_{\text{fun}} : \Pi x : ? \alpha. \tau.$$

Algorithm 16: $\text{commitSafe}(a, M, o, r, N)$

```
1  $(\Gamma, ?m, \tau, s) \leftarrow o$  // selected frontier obligation
2  $\tau_r \leftarrow \text{inferType}(\Gamma \cup \text{decl}(N), r)$  // type-check  $r$  in the extended context
3 if  $\tau_r = \perp \vee \tau_r \not\equiv \tau$  then
4   return  $\perp$  // do not instantiate surrounding type constraints
5  $\sigma \leftarrow [r/?m]$  // replace the selected hole
6  $M' \leftarrow \text{residual}(\sigma, (M \setminus \{o\}) \cup N)$  // update the unresolved source matches
7  $a' \leftarrow \sigma(a)$  // preserve source reconstruction
8 return  $(a', M')$ 
```

This substitution is safe independently of the eventual shape of a completion. If a completion maps $?m$ to $u : \theta(\tau)$, it can be lifted by assigning $?a := \theta(\tau)$, $?m_{\text{fun}} := \lambda x : \theta(\tau).x$, and $?m_{\text{arg}} := u$. The refined application then β -reduces to u .

To expose the immediate source subterms, the matching environment maps $?m_{\text{fun}}$ to the source function and $?m_{\text{arg}}$ to the source argument. This particular schema therefore succeeds only when the source function can be typed with the nondependent codomain τ in the current generalized context. The call to `commitSafe` checks that condition. A genuinely dependent application, whose result type is obtained by substituting the argument into a codomain family, is handled by the branching rule in Section 7.4.2.

Lambda abstraction.

When the expected type is already definitionally a dependent product $\Pi x : \tau_1. \tau_2$, the rule may replace $?m$ by $\lambda x : \tau_1. ?m_{\text{body}}$. Any completion f of the original hole lifts by setting $?m_{\text{body}} := f x$; the resulting term is an η -expansion of f . The rule is not classified as safe when the expected type must first be instantiated to a Π -type, because that unification can constrain other metavariables. That case is handled by the branching procedure in Section 7.4.2.

The procedure `refineSafe` repeatedly applies the implemented safe rules to a state (a, M) until no such rule applies. For a fixed traversal order and fresh-name policy it returns a deterministic safe-normal state; we do not assume confluence or claim a unique normal form independently of that strategy. Each successful call to `commitSafe` checks both type preservation

Algorithm 17: $\text{refineSafe}(a, M)$

```
1 while true do
2    $b \leftarrow \perp$  // first certified successor in this pass
3   for  $o = (\Gamma, ?m, \tau, s) \in M$  do
4     if  $b = \perp$  then
5       // application
6       case  $s \equiv s_1 s_2$  do
7          $x \leftarrow \text{freshVar}$ 
8          $?m_{\text{fun}}, ?m_{\text{arg}}, ?\alpha \leftarrow \text{freshMVar}$ 
9          $r \leftarrow ?m_{\text{fun}} ?m_{\text{arg}}$ 
10         $N \leftarrow \{ (\Gamma, ?\alpha, \text{mvarType}(\alpha), \text{inferType}_\Gamma(s_2)),$ 
11                   $(\Gamma, ?m_{\text{fun}}, \Pi x : ?\alpha. \tau, s_1), (\Gamma, ?m_{\text{arg}}, ?\alpha, s_2) \}$ 
12         $b \leftarrow \text{commitSafe}(a, M, o, r, N)$ 
13      //  $\lambda$ -term with a known  $\Pi$ -type
14      case  $s \equiv \lambda y : s_A. s_B \wedge \tau \equiv \Pi x : A. B$  do
15         $?m_{\text{body}} \leftarrow \text{freshMVar}$ 
16         $r \leftarrow \lambda x : A. ?m_{\text{body}}$ 
17         $N \leftarrow \{ (\Gamma[x : A], ?m_{\text{body}}, B, [x/y]s_B) \}$ 
18         $b \leftarrow \text{commitSafe}(a, M, o, r, N)$ 
19   if  $b = \perp$  then
20     return  $(a, M)$  // no rule applies: safe-normal state
21    $(a, M) \leftarrow b$  // continue from the certified successor
```

and source reconstruction, so the returned state remains a type-correct abstraction of the same source proof.

The two central invariants are:

Invariant 1a. If $(a_{\text{post}}, M_{\text{post}}) = \text{refineSafe}(a_{\text{pre}}, M_{\text{pre}})$, then $\hat{\eta}_{M_{\text{post}}}(a_{\text{post}}) \equiv \hat{\eta}_{M_{\text{pre}}}(a_{\text{pre}})$.

Invariant 2a. The call returns a valid refinement substitution σ with $a_{\text{post}} = \sigma(a_{\text{pre}})$. Hence typing is preserved by dependent substitution; when the target context and target type are rigid, both states have the same target type.

These invariants are instances of the following lemma.

Lemma 7.4.2 (Invariant by certified refinement). *Suppose that σ witnesses a refinement from $(a_{\text{pre}}, M_{\text{pre}})$ to $(a_{\text{post}}, M_{\text{post}})$. Then:*

1. $\eta_{M_{\text{post}}}(a_{\text{post}}) = \eta_{M_{\text{pre}}}(a_{\text{pre}})$; and
2. if $a_{\text{pre}} = \langle \Delta_{\text{pre}}, t_{\text{pre}} \rangle$, $a_{\text{post}} = \langle \Delta_{\text{post}}, t_{\text{post}} \rangle$, and $\hat{\Delta}; \Delta_{\text{pre}} \vdash t_{\text{pre}} : \tau$, then $\hat{\Delta}; \Delta_{\text{post}} \vdash t_{\text{post}} : \sigma(\tau)$.

Proof. For the first claim,

$$\eta_{M_{\text{post}}}(a_{\text{post}}) = \eta_{M_{\text{post}}}(\sigma(a_{\text{pre}})) = (\eta_{M_{\text{post}}} \circ \sigma)(a_{\text{pre}}) = \eta_{M_{\text{pre}}}(a_{\text{pre}}).$$

The second claim is the standard dependent substitution lemma applied to the valid substitution σ , followed by the equality $t_{\text{post}} = \sigma(t_{\text{pre}})$. $\square$

Termination of `refineSafe` follows from the same well-founded source-obligation measure used in Theorem 7.4.5: every successful constructor exposure replaces one source obligation by obligations for proper substructure of its annotated term or typing tree. Ordinary syntax-tree size suffices when only term obligations are stored; auxiliary type obligations require the annotated measure stated in that theorem.

Proposition 7.4.3 (Safety of the nondependent application and lambda schemas). *The nondependent application and known- Π lambda refinements used by `refineSafe` are safe substitutions.*

Proof. For the application rule, let $?m : \tau$ be replaced by $?m_1 ?m_2$, where $?m_2 : ?\alpha$ and $?m_1 : \Pi x : ?\alpha. \tau$. Let θ be any completion of the pre-state and put $u = \theta(?m)$. Extend θ to the fresh metavariables by

$$?\alpha := \theta(\tau), \quad ?m_1 := \lambda x : \theta(\tau). x, \quad ?m_2 := u.$$

The resulting application is well typed and β -reduces to u .

For the lambda rule, suppose $?m : \Pi x : \tau_1. \tau_2$ is replaced by $\lambda x : \tau_1. ?m_{\text{body}}$. Given a completion θ , put $f = \theta(?m)$ and assign $?m_{\text{body}} := f x$ in the local context $x : \theta(\tau_1)$. The refined lambda is well typed at the same dependent function type. Thus every completion of the pre-state lifts to a completion of the post-state in both cases. $\square$

Algorithm 18: $\text{commitUnsafe}(a, M, o, r, N)$

```
1  $R \leftarrow \emptyset$  // one successor per certified unifier
2  $(\Gamma, ?m, \tau, s) \leftarrow o$ 
3  $\tau_r \leftarrow \text{inferType}(\Gamma \cup \text{decl}(N), r)$  // reject an ill-typed proposal
4 if  $\tau_r = \perp$  then
5   return  $R$ 
   // enumerate all admissible type unifiers
6 for  $\theta \in \text{unifyAll}_{\Gamma \cup \text{decl}(N)}(\tau_r, \tau)$  do
7    $\sigma \leftarrow \theta \circ [r / ?m]$  // replace the hole, then apply  $\theta$ 
8    $M' \leftarrow \text{residual}(\sigma, (M \setminus \{o\}) \cup N)$  // update residual source matches
9    $a' \leftarrow \sigma(a)$ 
10   $R \leftarrow R \cup \{(a', M')\}$ 
11 return  $R$ 
```

7.4.2 Potentially Unsafe Metavariable Refinements

A refinement is placed in the branching phase when the chapter has no generic proof that it preserves every completion. Such a refinement is *potentially unsafe*; the label does not mean that every resulting branch is unprovable. It means that committing to the constructor or atom can instantiate shared type metavariables and thereby remove completions available before the commitment. Each successful commitment is therefore retained as a separate search state.

Example 23 (Order-sensitive refinement). *Consider the application $\text{hcp } ?m \text{ ha hb}$ in the running example. Exposing its applications yields a skeleton of the form $(((?m_1 ?m_2) ?m_3) ?m_4)$. In the generalized target context, ha and hb have types $k \mid a$ and $k \mid b$. If the divisor argument $?m_2$ is first committed to 2, the intended matches $?m_3 := \text{ha}$ and $?m_4 := \text{hb}$ are ill typed while k is rigid. A branch in which the argument position remains generalized can instead use k , which reconstructs 2 after applying ρ . The choices therefore belong to different branches; one can reconstruct a useful generalized proof while the other is rejected by typing or source matching.*

The procedure refineUnsafe in Algorithm 19 exposes one source constructor or atom at each frontier obligation and invokes commitUnsafe . A branch is retained only when type inference succeeds, a returned unifier makes the replacement well typed, and residual matching

Algorithm 19: $\text{refineUnsafe}(a, M)$

```
1  $Q \leftarrow \emptyset$  // certified one-step successors
2 for  $o = (\Gamma, ?m, \tau, s) \in \text{frontier}(a, M)$  do
  // atom
3   case  $s \equiv x \mid s \equiv c \mid s \equiv \text{sort}$  do
4      $Q \leftarrow Q \cup \text{commitUnsafe}(a, M, o, s, \emptyset)$ 
  // dependent  $\Pi$ -type
5   case  $s \equiv \Pi y : s_A. s_B$  do
6      $x \leftarrow \text{freshVar}$ 
7      $?A \leftarrow \text{freshMVar}(\Gamma)$ 
8      $?B \leftarrow \text{freshMVar}(\Gamma[x : ?A])$ 
9      $r \leftarrow \Pi x : ?A. ?B$ 
10     $N \leftarrow \{ (\Gamma, ?A, \text{mvarType}(?A), s_A), (\Gamma[x : ?A], ?B, \text{mvarType}(?B), [x/y]s_B) \}$ 
11     $Q \leftarrow Q \cup \text{commitUnsafe}(a, M, o, r, N)$ 
  //  $\lambda$ -term with unknown type
12  case  $s \equiv \lambda y : s_A. s_B$  do
13     $x \leftarrow \text{freshVar}$ 
14     $B_s \leftarrow \text{type}_{\Gamma[x : s_A]}([x/y]s_B)$ 
15     $?A \leftarrow \text{freshTypeMVar}(\Gamma)$ 
16     $? \beta \leftarrow \text{freshTypeFamilyMVar}(\Gamma, ?A)$ 
17     $?m_{\text{body}} \leftarrow \text{freshMVar}$ 
18     $r \leftarrow \lambda x : ?A. ?m_{\text{body}}$ 
19     $N \leftarrow \{ (\Gamma, ?A, \text{mvarType}(?A), s_A), (\Gamma, ? \beta, \text{mvarType}(? \beta), \lambda x : s_A. B_s),$ 
20       $(\Gamma[x : ?A], ?m_{\text{body}}, ? \beta x, [x/y]s_B) \}$ 
21     $Q \leftarrow Q \cup \text{commitUnsafe}(a, M, o, r, N)$ 
22 return  $Q$ 
```

obligations remain well formed.

Constants, sorts, and variables.

An atomic refinement replaces a hole by the corresponding source atom. The atom's inferred type is unified with the hole's expected type. This unification may instantiate other metavariables, so matching and well-formedness must be rechecked after applying the complete substitution; type compatibility alone is not sufficient.

Dependent applications.

For a source application $s_1 s_2$ whose function type is definitionally $\Pi x : A_s. B_s$, the branching rule introduces a fresh domain $? \alpha$, a fresh codomain family $? \beta$, and holes

$$?m_{\text{fun}} : \Pi x : ? \alpha. ? \beta x, \quad ?m_{\text{arg}} : ? \alpha.$$

The application has type $? \beta ?m_{\text{arg}}$, which must be unified with the expected type τ . Unlike the nondependent safe schema, this equation can constrain the argument or other shared metavariables. The matching environment maps the fresh domain and codomain family to A_s and $\lambda x : A_s. B_s$, and maps the two term holes to s_1 and s_2 .

Lambda abstractions.

When a lambda source has an expected type already definitionally equal to a Π -type, Section 7.4.1 handles it without branching. The unsafe lambda case is the remaining case, in which exposing the lambda first requires unifying the expected type with a fresh dependent function type. That unification can constrain other holes, so it is explored as a branch.

Dependent Π -types.

A Π -term is itself a type, not a function value that “takes an argument.” Exposing a Π constructor commits a type-valued hole to dependent-function structure. Unlike the lambda rule, there is no general η -expansion that turns an arbitrary type completion into a Π -type. The algorithm therefore introduces fresh metavariables for the domain and codomain and treats the resulting unification as potentially unsafe. The core model keeps universe levels fixed rather than abstracting them in this rule.

Every branch produced by `refineUnsafe` is accepted only through `commitUnsafe`, which constructs the full refinement substitution, computes residual matching obligations, and checks source reconstruction. Consequently, the following invariants are properties of retained branches rather than of every syntactically proposed constructor exposure.

Invariant 1b. If $(a_{\text{post}}, M_{\text{post}}) \in \text{refineUnsafe}(a_{\text{pre}}, M_{\text{pre}})$, then $\hat{\eta}_{M_{\text{post}}}(a_{\text{post}}) \equiv \hat{\eta}_{M_{\text{pre}}}(a_{\text{pre}})$.

Invariant 2b. If $(\langle \Delta_{\text{post}}, t_{\text{post}} \rangle, M_{\text{post}}) \in \text{refineUnsafe}(\langle \Delta_{\text{pre}}, t_{\text{pre}} \rangle, M_{\text{pre}})$, then there exists a valid refinement witness σ such that $\langle \Delta_{\text{post}}, t_{\text{post}} \rangle = \sigma(\langle \Delta_{\text{pre}}, t_{\text{pre}} \rangle)$. Consequently, if $\hat{\Delta}; \Delta_{\text{pre}} \vdash t_{\text{pre}} : \tau$, then $\hat{\Delta}; \Delta_{\text{post}} \vdash t_{\text{post}} : \sigma(\tau)$.

In particular, when the target context has been made rigid, σ fixes the target type $\hat{\tau}$, and the post-state still has type $\hat{\tau}$.

Proof of Invariants 1b and 2b. For every proposed atom or constructor, `commitUnsafe` first infers the replacement type and enumerates unifiers with the selected hole's expected type. For each returned unifier θ , it forms the complete refinement witness $\sigma = \theta \circ [r/?m]$. The residual-matching procedure rejects the branch unless all declarations remain well formed and the new matching map factors the old one through σ . The final reconstruction check establishes Invariant 1b, and validity of σ together with the dependent substitution lemma establishes Invariant 2b. The application, Π , and lambda cases additionally introduce matching obligations for every fresh metavariable occurring in the replacement or its type; their source assignments are the immediate term and type components shown in Algorithm 19. $\square$

7.4.3 The Main Loop

The procedure `enumerateAbstractTerms` (Algorithm 20) takes a checked source `aterm`, a generalized target type, and the fixed source-type matching witness ρ . It returns the safe-normal, source-compatible representatives reachable through the implemented constructor-guided refinement schemas.

The main loop first validates ρ and the target type. It then initializes Q with one hole for the entire proof. Metavariables belonging to the target telescope $\hat{\Delta}$ are temporarily made rigid so that search-time unification cannot specialize the theorem parameters (line 2). In each iteration, the algorithm removes a state from Q (line 10), applies the fixed safe-normalization strategy (line 11), records the resulting representative (line 14), and generates all branches returned by

Algorithm 20: $\text{enumerateAbstractTerms}(\langle \Delta, t \rangle, \langle \hat{\Delta}, \hat{t} \rangle; \rho)$

```

1   $\tau_{\text{src}} \leftarrow \text{inferType}(\Delta, t)$  // source proof type
2   $(\iota, \Gamma_{\text{init}}, \tau_{\text{init}}) \leftarrow \text{rigidify}(\hat{\Delta}, \hat{t})$  // freeze target parameters during search
3   $?m_{\top} \leftarrow \text{freshMVar}$ 
4   $a_{\text{init}} \leftarrow \langle [\Gamma_{\text{init}}, ?m_{\top} : \tau_{\text{init}}], ?m_{\top} \rangle$ 
5   $M_{\text{init}} \leftarrow \{(\Gamma_{\text{init}}, ?m_{\top}, \tau_{\text{init}}, t)\}$  // one hole represents the complete source proof
6   $Q \leftarrow \{(a_{\text{init}}, M_{\text{init}})\}$  // pending states
7   $S \leftarrow \emptyset$  // visited safe-normal states
8   $\mathcal{A} \leftarrow \emptyset$  // output representatives
9  while  $Q \neq \emptyset$  do
10    $(a, M) \leftarrow Q.\text{pop}()$ 
11    $(a_{\text{nf}}, M_{\text{nf}}) \leftarrow \text{refineSafe}(a, M)$  // normalize by safe rules
12   if  $(a_{\text{nf}}, M_{\text{nf}}) \notin S$  then
13      $S \leftarrow S \cup \{(a_{\text{nf}}, M_{\text{nf}})\}$  // mark this normal state as visited
14      $\mathcal{A} \leftarrow \mathcal{A} \cup \{a_{\text{nf}}\}$  // record the representative
15      $Q_{\text{unsafe}} \leftarrow \text{refineUnsafe}(a_{\text{nf}}, M_{\text{nf}})$  // generate unsafe branches
16      $Q \leftarrow Q \cup Q_{\text{unsafe}}$ 
17 return  $\{\text{unrigidify}(a) \mid a \in \mathcal{A}\}$  // restore the target metavariables

```

refineUnsafe under the chosen unification procedure (line 15). Those branches are added to Q for further exploration (line 16).

For the following statements, call an aterm *source-compatible* when it is well typed at the target type and admits a matching environment that reconstructs the original proof term. We write $\rightarrow_s$ for one of the safe refinements implemented by refineSafe , and call an aterm *safe-normal* when no such refinement is applicable.

Theorem 7.4.4 (Soundness of enumeration). *Fix a valid source-type abstraction witness ρ . Assume that the input proof is well typed, the target context is rigid during enumeration, every returned unifier is a valid substitution, and every transition produced by refineSafe and refineUnsafe is a certified refinement. If*

$$\mathcal{A} = \text{enumerateAbstractTerms}(\langle \Delta, t \rangle, \langle \hat{\Delta}, \hat{t} \rangle; \rho),$$

then every $a^* = \langle \Delta^*, t^* \rangle \in \mathcal{A}$ satisfies:

1. $\hat{\Delta}; \Delta^* \vdash t^* : \hat{\tau}$;
2. *there exists a matching environment M^* such that $\hat{\eta}_{M^*}(a^*) \equiv \langle \Delta, t \rangle$; and*
3. *a^* is safe-normal with respect to $\longrightarrow_s$.*

After changing the rigid variables in $\hat{\Delta}$ back to metavariables, a^* is therefore a type-correct abstraction of the original proof in a context extending $\hat{\Delta}$.

Proof. The initial state is well typed at $\hat{\tau}$ and its unique matching obligation maps $?m_\top$ to t , so it is source-compatible. By Invariants 1a–2a and 1b–2b, source compatibility and typing are preserved by every transition inserted into the worklist. Hence they hold whenever a candidate is removed from the worklist. The call to `refineSafe` runs until no safe rule applies, so the candidate inserted into $\mathcal{A}$ is safe-normal. Replacing the rigid target variables by their original metavariables gives context consistency. $\square$

Theorem 7.4.5 (Termination). *Suppose that every unification call terminates and returns finitely many answers. Also suppose that source matching is well founded: there is a weight w from source obligations to positive integers such that, whenever a refinement replaces an obligation with obligations for $s_1, \dots, s_k$,*

$$w(s) > \sum_{i=1}^k w(s_i).$$

Then `enumerateAbstractTerms` terminates.

Proof. For a matching environment M , define

$$\mu(M) = \sum_{(\Gamma, ?m, \tau, s) \in M} w(s).$$

An atomic refinement removes an obligation, and every constructor refinement replaces one obligation by obligations of strictly smaller total weight. Consequently, every elementary safe or unsafe refinement decreases the positive-integer measure μ by at least one. The depth of

every branch is therefore bounded by the initial value of μ . At each state there are finitely many frontier metavariables and, by assumption, finitely many returned unifiers, so the search tree is finitely branching. A finitely branching tree with this uniform finite depth is finite; hence the worklist eventually becomes empty. $\square$

When M contains only term-structure obligations, one may take $w(s)$ to be the ordinary syntax-tree size of s . If matching obligations are also stored for auxiliary type metavariables, use instead the size of the finite annotated typing tree: the children of an obligation are exactly the term and type obligations introduced by exposing its outer constructor. The theorem only requires this decomposition relation to be acyclic; it does not require an inferred type to be a literal syntactic subterm of the proof term.

The termination theorem makes the concrete search graph finite, but it does not by itself establish completeness with respect to arbitrary valid unifiers or arbitrary interleavings of safe and branching steps. We therefore state the exact exhaustiveness property of the implemented worklist.

For a refinement state $X = (a, M)$, let $\text{NF}_s(X) = \text{refineSafe}(X)$ under the fixed traversal and fresh-name policy. Between safe-normal states, write $X \Longrightarrow Y$ when there exists $Z \in \text{refineUnsafe}(X)$ such that $Y = \text{NF}_s(Z)$, with states compared modulo the canonicalization used by the visited set S .

Theorem 7.4.6 (Exhaustiveness of the implemented search). *Assume the hypotheses of Theorem 7.4.5. Let X_0 be the initialized state in Algorithm 20, and let $X_0^{\text{nf}} = \text{NF}_s(X_0)$. If a safe-normal state $X = (a, M)$ is reachable by*

$$X_0^{\text{nf}} \Longrightarrow^* X,$$

then the aterm component a , after restoring the rigid target metavariables, occurs in $\mathcal{A}$ up to the definitional equality.

Proof. By induction on the length of the phase derivation. The base state is obtained when

the initialized state is first removed from Q , inserted into S , and recorded in $\mathcal{A}$. For the induction step, suppose a reachable safe-normal state X has been recorded. The algorithm evaluates $\text{refineUnsafe}(X)$ and inserts every returned successor Z into Q . When Z is removed, the algorithm computes $\text{NF}_s(Z)$. If its canonical form is new, it is recorded; if it is already in S , the same canonical state was recorded earlier. Thus every state reachable by one more phase is represented in $\mathcal{A}$. Termination guarantees that this finite exploration completes. $\square$

Theorem 7.4.6 concerns only branches returned by the implemented unsafe-refinement procedure. A stronger theorem covering constructor-guided derivations that use arbitrary valid unifiers would require more than termination and a complete set of type unifiers. In particular, residual source matching and branch certification would have to preserve factorization through a returned unifier, and NF_s would have to be stable under instantiation: normalizing a more general state and then specializing it would need to represent normalization of the specialized state. These properties do not follow from confluence of the safe rules alone, so the chapter does not claim that stronger form of completeness.

Corollary 7.4.7 (Rule-relative minimality). *No returned candidate has a strict descendant reachable by a nonempty sequence of the implemented safe rules. If, additionally, these rules recognize every semantically safe constructor-guided one-step refinement, then each returned candidate is minimal under $\preceq$ among its structural descendants.*

Proof. The first claim is safe-normality. For the second, suppose that a returned candidate had a strict safe descendant. Consider the first step on a constructor-guided path to that descendant. By prefix closure of safety (Proposition 7.1.6), that first step is safe. Completeness of the safe rules would therefore make it applicable to the returned candidate, contradicting safe-normality. $\square$

7.5 Dependency-Guided Metavariable Instantiation

A refinement can constrain several metavariables at once. For example, after refining $?m$ to an application $?m_1 ?m_2$, the fresh argument type $?a$ occurs both in the type of $?m_1$ and

in the type of $?m_2$. Instantiating any one of these metavariables can therefore constrain the other two. We make this propagation explicit with a metavariable dependency graph. We assume throughout that the metavariable context is *constraint-closed*: every postponed typing or unification constraint has either been solved or has been reified as a dependency.

7.5.1 Metavariable Dependency Graph

Definition 7.5.1 (Metavariable dependency). *Let $[\Gamma_{?m}, ?m : \tau_{?m}]$ be a metavariable declaration. The dependencies of $?m$ are*

$$D(?m) = MV(\tau_{?m}) \cup \bigcup_{d \in \Gamma_{?m}} MV(d).$$

Definition 7.5.2 (Metavariable dependency graph). *Given a metavariable telescope Δ , its metavariable dependency graph is the undirected graph $\mathcal{G}_\Delta = (MV(\Delta), E_\Delta)$, where*

$$E_\Delta = \{(?m_1, ?m_2) \mid ?m_1 \in D(?m_2) \vee ?m_2 \in D(?m_1)\}.$$

For $?m \in MV(\Delta)$, let

$$\llbracket ?m \rrbracket = \{?m' \in MV(\Delta) \mid (?m, ?m') \in E_\Delta^*\}$$

denote the connected component of $?m$.

The graph records dependencies in declarations, rather than mere co-occurrence in the proof term. Two metavariables may occur in the same application and still belong to different components when neither one's declaration depends on the other. Conversely, an indirect chain through expected types places metavariables in the same component even when they never occur adjacent in the term.

7.5.2 Componentwise Safe Instantiation

For $C \subseteq MV(\Delta)$, write $\Delta|_C$ for the restriction of the metavariable telescope to declarations in C . A *concrete component substitution* for C is a valid substitution for $\Delta|_C$ whose range contains no metavariables. Validity is checked against each metavariable's own local context and against the previously instantiated declarations in the restricted telescope; merely checking every assignment in isolation is not sufficient in a dependent context.

Theorem 7.5.3 (Safety of connected-component instantiation). *Let $\langle \Delta, t \rangle$ be a well-typed aterm in an external context Γ , let $C = \llbracket ?m \rrbracket$ be a connected component of $\mathcal{G}_\Delta$, and let η_C be a concrete component substitution for $\Delta|_C$. Then η_C is safe for $\langle \Delta, t \rangle$ in Γ .*

Proof. Assume that $\langle \Delta, t \rangle$ is completable in Γ , and let θ be a concrete completion of all metavariables in Δ . Put $\bar{C} = MV(\Delta) \setminus C$ and let $\theta_{\bar{C}}$ be the restriction of θ to $\bar{C}$.

Because C is a connected component, no declaration in C contains a metavariable from $\bar{C}$, and no declaration in $\bar{C}$ contains a metavariable from C . This statement includes expected types, the types of local assumptions, and the values of local definitions. Hence the two restricted metavariable telescopes are independent and may be exchanged. In particular, replacing the assignments $\theta|_C$ by the independently valid assignments η_C does not change any type that must be checked for a metavariable in $\bar{C}$.

Define the disjoint union

$$\theta^* = \eta_C \uplus \theta_{\bar{C}}.$$

The preceding independence argument shows that θ^* is a valid concrete substitution for all of Δ . By the dependent substitution lemma, $\theta^*(t)$ is well typed in Γ . Equivalently, $\theta_{\bar{C}}$ is a completion of the partially instantiated aterm $\eta_C(\langle \Delta, t \rangle)$. Thus completability of the original aterm implies completability after applying η_C , which is exactly safety. $\square$

The theorem also certifies partial structural refinements. Suppose σ_C refines only metavariables in C and introduces a new set of holes, and suppose there is a substitution

κ_C for those holes such that $\kappa_C \circ \sigma_C$ is a concrete component substitution. By Theorem 7.5.3, the composite $\kappa_C \circ \sigma_C$ is safe. By prefix closure of safety (Proposition 7.1.6), σ_C itself is safe. Importantly, κ_C need not reconstruct the corresponding subterms of the original proof: any well-typed completion of the component is a certificate that the partial refinement is safe.

Lemma 7.5.4. *Suppose a refinement of a metavariable in a component C introduces only fresh metavariables and generates unification constraints mentioning only metavariables in C together with those fresh metavariables. Then the refinement may split or eliminate C , but it cannot merge C with a distinct pre-refinement connected component.*

Proof. Every new declaration mentions only rigid variables, old metavariables from C , and fresh metavariables introduced by the same refinement. Therefore all new dependency edges have both endpoints in C augmented with the fresh metavariables. The unifier is computed from constraints over the same set, so applying it cannot introduce an occurrence of a metavariable from another old component. No new cross-component edge can therefore be created. $\square$

Corollary 7.5.5 (Commutation of independent refinements). *Refinements acting on two distinct connected components commute up to renaming of fresh metavariables, provided their unifiers are chosen canonically.*

Proof. The substitutions have disjoint domains and, by Lemma 7.5.4, disjoint dependency closures. Each leaves the constraints and matching obligations of the other unchanged. Their compositions therefore agree up to the names chosen for fresh metavariables. $\square$

7.5.3 Pruning the Enumeration

The enumerative search described in Section 7.4 can be computationally expensive because many candidate refinements require branching and generate a large search tree. To make the synthesis tractable, we introduce several additional pruning strategies that prune branches early in the search process.

Component-completion pruning

A refinement that can be extended to a concrete completion of its entire connected component is safe by Theorem 7.5.3 and prefix closure. Once such a concrete completion certificate has actually been found, an implementation may absorb the refinement into safe normalization rather than represent it as a branching choice. This criterion can certify refinements beyond the two syntactic safe rules from Section 7.4.1, including sequences containing constants, variables, or other locally constraining choices. Finding the certificate is itself a proof-search problem; the pruning rule is sound for returned certificates and does not assume that component completion is decidable in general.

Partial-order reduction

By Corollary 7.5.5, different interleavings of refinements in independent components lead to the same state up to renaming. Under the hypotheses of Corollary 7.5.5, the enumerator may therefore impose a canonical order on components—for example, always refine the component whose least metavariable has the smallest creation index—and omit the other interleavings. This removes permutation blow-up without removing a distinct result modulo fresh-variable renaming and the chosen canonical unifiers. If a refinement splits a component, the same canonical policy is applied to the newly created components.

Equivalence and Propositional Equality

Syntactic component completion is not the only way to prove safety. For example, given an equivalence $e : A \simeq B$ and a metavariable $?m : A$ that has no dependent metavariables outside the transformed component, the refinement

$$[e^{-1}(?m_B)/?m], \quad ?m : A, \quad ?m_B : B,$$

is safe: a completion $u : A$ of $?m$ lifts to the completion $e(u) : B$ of $?m_B$. When other metavariables have types depending on $?m$, the same construction must transport their assignments as part of a

component-wide lift. An equality $h : A = B$ yields the corresponding rule using transport along h , and for propositions an equivalence $A \leftrightarrow B$ uses its two implication directions. These certificates allow rewriting or changing representation without branching.

Such rules preserve completability, but they may reconstruct the original source term only up to propositional equality rather than definitional equality. Accordingly, they should either insert the explicit transport terms into the proof and matching certificate, or the abstraction relation should be extended to carry a proof of equality between the reconstructed and original terms.

Deferred implicit metavariables

Implicitness is an elaboration convention, not a kernel-level proof of safety. It is nevertheless useful for pruning when made explicit as an oracle assumption. Let I be a set of metavariables designated as implicit and let solve_I be a finalization procedure. The search may leave the metavariables in I unresolved and suppress structural refinements below them. Whenever solve_I returns a valid substitution, applying it preserves type correctness by the substitution lemma. The pruning is complete relative to solve_I when that procedure is complete for the class of implicit constraints generated by the search.

Alternatively, unresolved implicit metavariables may be generalized as implicit parameters of the output theorem, provided the remaining metavariable telescope is well formed and generalized in dependency order. This preserves type correctness, but it preserves an optimality claim only when the cost model and the abstraction order explicitly quotient away such implicit parameters. Merely marking a metavariable as implicit is therefore not, by itself, sufficient justification for discarding its refinements.

7.6 Related Work

Weakest preconditions and predicate transformers.

Dijkstra’s weakest-precondition (WP) calculus maps postconditions backward through program syntax to identify the largest set of valid initial states [37, 38]. In contrast to WP calculus—which relies on operational semantics to synthesize logically weakest preconditions over execution traces—our method operates directly on the structure of an existing proof term in dependent type theory. Consequently, the exposed premises represent sufficient conditions for replaying *that specific proof skeleton* rather than logically weakest assertions across all possible proofs. Our guarantee is structural and rule-relative: returned candidates are safe-normal, and stronger minimality requires the additional safe-rule coverage assumption stated in Corollary 7.4.7. Proof abstraction is therefore complementary to WP calculation, compressing a completed derivation into a reusable theorem interface rather than computing a globally weakest assertion.

Abduction and bi-abduction.

Logical abduction infers missing premises to complete entailments, with bi-abduction extending this to separation logic for compositional shape analysis [24], and abductive specification synthesis deriving library bounds from client verifier counterexamples [163]. While conventional abduction searches an assertion logic for candidate formulas, our framework leverages dependent type checking to compute the exact type of a missing subproof once an abstraction boundary is chosen. The primary challenge we address is searching over structural cuts to find boundaries that expose useful, type-correct premises while retaining a concrete reconstruction witness, rather than computing weakest anti-frames in an assertion logic.

Library learning.

Library-learning frameworks compress program corpora by discovering reusable domain-specific abstractions using top-down synthesis [21], anti-unification with e-graphs [26], or task-corpus compression [45]. Unlike traditional library learning, which requires a multi-task

corpus to identify recurring routines, our method operates on a *single proof term* and target type. Furthermore, rather than emitting unverified code fragments optimized for corpus compression, our framework produces theorem candidates with explicit premises whose completed terms are checked by the underlying kernel. This single-proof setting is useful in domains that do not yet have a large proof corpus.

Meta-tactic synthesis.

Procedural proof reuse techniques by tactic synthesis, such as TacticToe [53], Peano [121], and TacMiner [160], learn reusable meta-tactics and tactic-dependence subgraphs to guide automated proof search across corpora. In contrast, our framework extracts declarative, kernel-checkable theorems with explicit, inspectable logical interfaces rather than procedural search scripts. While meta-tactics reduce the computational cost of re-executing proof steps, proof abstraction distills the underlying mathematical facts established by those steps, making the two approaches complementary for automated proof engineering.

7.7 Conclusion

This chapter presented proof generalization as an abstraction-synthesis problem over dependently typed proof terms. Given one checked proof and a generalized target type, the framework searches for proof templates that still reconstruct the source proof, typecheck at the target, and expose only the additional premises needed to preserve validity. In contrast to approaches that learn recurring tactic sequences from a corpus, the method derives logical structure from a single proof and represents the result as another kernel-checkable theorem.

Several practical challenges remain. The generalized target type and its source-matching witness are currently inputs rather than inferred end to end. Template enumeration can still be expensive, and the usefulness of an extracted interface depends on how transparent the source proof is: opaque computations and specialized library lemmas may hide the structure that should become an explicit premise. Result ranking is also separate from correctness; among several

sound candidates, users may prefer the one with fewer parameters, standard typeclasses, or premises already supported by library lemmas.

Future work could use e-graphs [159] to share equivalent template structure, strengthen semantic safety certificates for rewriting and representation changes, and rank candidates by interface quality. Language models could propose target types or promising abstraction boundaries, while the proof assistant supplies precise feedback about ill-typed proposals, over-specialization, and unresolved dependency components. Successfully checked abstractions could then be added to a lemma library for later proof search.

More broadly, proof abstraction provides a route from proof completion to proof reuse. It can turn a long concrete derivation into a shorter theorem plus explicit obligations while retaining the guarantee actually provided by the underlying assistant: the resulting proof term is accepted by its checker relative to the formal statement, logic, assumptions, and trusted computing base.

Chapter 8

Conclusion

Modern generative models can produce code and proofs at unprecedented speed, but generation alone provides no guarantee that the resulting artifacts are correct or well-structured. Formal methods provide such guarantees, but often at substantial manual and computational cost. This dissertation contributes techniques in different layers of program reasoning: grammar-based control of LLM generation, synthesis of formal specifications from source code, and modular synthesis and proof reuse for larger systems. The sections below detail each contribution.

8.1 Summary of Contributions

Syntactic Control of LLM Generation

Chapter 2 addressed a practical obstacle in grammar-constrained decoding: the tokens used by an LLM do not align with the terminals recognized by a conventional lexer and parser. GREATGRAMMA resolves this mismatch by precomputing, for each lexer state, the terminal sequences realizable by the LLM vocabulary. This representation made offline preprocessing faster while retaining the lowest online masking overhead among the compared systems. The evaluation also exposed crashes, nontermination, and incorrect behavior in existing implementations.

Grammar-Aligned Decoding

Chapter 3 showed that token masking can substantially distort the probability distribution learned by an LLM, even when every generated output is grammatical. The chapter formalized

grammar-aligned decoding as sampling from the LLM distribution conditioned on grammaticality and introduced ASAp, which progressively estimates the expected probability that a prefix can be completed to a grammatical output. Across program-synthesis and constituency-parsing benchmarks, ASAp moved the empirical distribution toward the aligned target while ordinary grammar-constrained decoding retained a persistent bias.

Synthesizing Specifications

Chapter 4 introduced a customizable framework for synthesizing sound, most-precise specifications in a user-defined language. Rather than requiring a programmer to state a complete specification manually, SPYRO searches for strongest consequences of a semantic query that are expressible in the chosen DSL. Its counterexample-guided algorithm synthesizes individual strongest properties, distinguishes negative examples that must be rejected from those that may be rejected, and combines incomparable properties into a best conjunction.

Synthesizing Specifications for Nondeterministic Programs

Chapter 5 extended specification synthesis to nondeterministic programs and to both over- and under-approximate reasoning. The LOUD framework supports strongest over-approximations, which describe behavior that must hold, and weakest under-approximations, which describe behavior that may occur for some execution. Its implementation, ASPIRE, combines specification synthesis loop with counterexample-guided quantifier instantiation to handle existentially quantified queries. The case studies further instantiated the same framework for incorrectness reasoning, concurrent-program analysis, and symbolic strategies for two-player games, demonstrating that a single logical interface can expose both unavoidable and possible behavior.

Modular System Synthesis

Chapter 6 introduced MOSS, a framework for modular system synthesis based on implementation-agnostic specifications. The framework identifies two complementary synthesis problems: synthesizing a module implementation from lower-level contracts, and synthesizing a contract from a completed module. Alternating these operations allows a system to be

constructed layer by layer while hiding lower-level representation choices. In the running multi-layer case study, modular synthesis remained on the order of one to five minutes per module, whereas synthesis with all underlying implementations exposed grew with the accumulated system and eventually failed with the available tools. The MOSSKIT case studies also revealed bugs in synthesized implementations, in manually written specifications, and in specifications that were too weak for the next layer. The principal result is therefore both computational and methodological: contracts bound the current search problem and make cross-module assumptions explicit enough to test.

Extracting Reusable Proof Structure

Chapter 7 addressed the proof reuse problem. It formalized proof abstraction as the replacement of selected subterms of a checked, dependently typed proof with metavariables, while preserving a reconstruction path back to the source proof. The top-down enumeration procedure distinguishes refinements that preserve provability without branching from refinements whose safety depends on additional commitments, and the metavariable dependency graph identifies choices that must be resolved together. The case studies illustrate how concrete computations and specialized operations can be replaced by explicit logical premises or algebraic interfaces, yielding typed proof templates whose remaining obligations state what must be proved in a new context.

8.2 Directions for Future Work

From Search to Statement and Check

The frameworks in this dissertation were built at a time when the dominant approach to producing formal artifacts was enumerative or symbolic search. Enumerative synthesizers suffered from state-space explosion, and symbolic ones were constrained by the theory support of their solvers. Frontier LLMs change this calculus: an LLM can propose candidates in spaces far larger than any solver can explore, and the role of formal methods shifts accordingly, from

doing the search itself to stating what counts as a correct or high-quality candidate and then checking each proposal against that statement.

A future research in this direction is to express user intent as formal, checkable constraints and use LLMs as the generative engine that produces artifacts a checker accepts. Two concrete near-term instances build directly on Parts II and III. The first is a proof abstraction pipeline that combines user-facing preferences, such as which parameters a proof should be polymorphic in and which computations should be hidden behind lemmas, with formal machinery for inferring the weakest additional premises under which the generalized statement remains provable. The second is a reformulation of specification synthesis in the same style, expressing soundness, strength, and simplicity as separate formal constraints and letting an LLM propose candidates that a checker accepts or rejects.

Formal Quality Criteria as Training Signals

The soundness, precision, and modularity criteria developed in this dissertation are not only useful as filters after generation; they can also serve as automated reward signals for training. Rather than training a model to imitate human-written proofs, one could train it to produce proofs that satisfy formal quality criteria: reusable, appropriately abstract, and computationally transparent. Because such signals require only a formal criterion and a checker for it, they can be produced at scale without human-authored ground truth.

End-to-End Evaluation on Real Systems

The techniques developed in this dissertation, and the extensions above, need eventual evaluation across realistic software lifecycles rather than only on isolated benchmarks and proof-of-concept case studies. Important questions include whether synthesized specifications remain readable, whether module contracts survive implementation changes, whether proof templates transfer across libraries, and whether structured formal feedback measurably reduces the labor and compute required to reach a checked, maintainable system. Measuring these factors would connect the theoretical notions of soundness, precision, and relative completeness to the practical

costs borne by developers and proof engineers.

8.3 Closing Perspective

One vision of automated software verification looks as follows. A human interprets a program’s semantics inside an interactive theorem prover, states in the same setting what correctness means for that program, and an AI model discharges the resulting proof obligations. This resembles how mathematical theorems are increasingly proved with machine assistance, but the software case involves upstream steps that a mathematical proof typically does not: building a formal model of what the program does, and stating in that model what correct behavior means.

Such a workflow becomes practical only when each of its steps can be performed at reasonable cost. Frontier models are already lowering the cost of the proof step. The techniques developed in this dissertation aim at the remaining challenges: reducing the manual burden of writing specifications, and making verification modular enough that large programs and their proofs stay within reach.

Formal reasoning need not compete with generative AI in such a workflow; it can define the surface on which AI operates. As models grow more capable at proof search, the residual bottleneck may move upstream, toward the human tasks of semantic modeling and specification writing. Relieving that bottleneck is the direction this dissertation contributes to.

Appendix A

Specification Synthesis Evaluation Details

In this chapter, we first describe the benchmarks in detail and then present detailed evaluation metrics.

A.1 Specification-Mining Benchmarks

SyGuS

We “inverted” the roles from the original benchmarks. We selected a few representative reference implementations of functions from the SYGUS repository, and applied SPYRO to synthesize specifications for them. In this case, we have a “ground-truth” property against which we can compare, namely, the specification that a SYGUS solver would start from to synthesize the function. We bounded the size of integers to 5 bits.

Max

`max2`, `max3`, and `max4` are problems where the goal is to find a function that computes the maximum of 2, 3, or 4 integers, respectively. For each problem, we provided the reference implementation given in the SYGUS repository, and synthesized properties expressed in the grammar

$$\begin{aligned} B &:= \top \mid AP \mid AP \vee AP \mid AP \vee AP \vee AP \mid \dots \\ AP &:= I \{ \leq \mid < \mid \geq \mid > \mid = \mid \neq \} I \\ I &:= x_1 \mid x_2 \mid \dots \mid o \end{aligned} \tag{A.1}$$

The nonterminal I derives all input and output variables appearing in the given query. The grammar for `max2` and `max3` allows up to three disjuncts in B , the grammar for `max4` allows four disjuncts, and the grammar for `max5` allows five disjuncts.

Diff

`diff` is a SYGUS problem where the goal is to find a commutative function that returns the difference of two input variables—i.e. $o = x - y \vee o = y - x$ whenever $o = \text{diff}(x, y)$.

First, the `diff1` benchmark synthesizes consequences of the query $o = \text{diff}(x, y)$ using the following grammar:

$$\begin{aligned} B &:= \top \mid AP \mid AP \vee AP \mid AP \vee AP \vee AP \\ AP &:= I \{ \leq \mid < \mid \geq \mid > \mid = \mid \neq \} I + I \\ I &:= x \mid y \mid o \mid 0 \end{aligned} \tag{A.2}$$

The nonterminal I derives all input and output variables appearing in the given query, and also the constant 0, which appears in the implementation of the function.

Second, the `diff2` benchmark synthesizes consequences of the query $o_1 = \text{diff}(x_1, y_1) \wedge o_2 = \text{diff}(x_2, y_2)$, using the following grammar:

$$\begin{aligned} B &:= \top \mid AP \mid AP \vee AP \mid AP \vee AP \vee AP \\ AP &:= I \{ = \mid \neq \} I \\ I &:= x_1 \mid y_1 \mid o_1 \mid x_2 \mid y_2 \mid o_2 \mid 0 \end{aligned} \tag{A.3}$$

Using a query in which a function symbol appears twice allows one to express properties such as commutativity.

Array

`arrSearch2` and `arrSearch3` are SYGUS problems where the goal is to synthesize an expression that finds the index of an element in a sorted tuple of size 2 and 3, respectively. For example, $o = \text{arrSearch3}(x_1, x_2, x_3, k)$ implies that o is the index of the value k in the

sorted tuple (x_1, x_2, x_3) . For each problem, we synthesize properties expressed in the following grammar:

$$\begin{aligned}
B &:= \top \mid AP \mid AP \vee AP \mid AP \vee AP \vee AP \mid \dots \\
AP &:= I \{ \leq \mid < \mid \geq \mid > \mid = \mid \neq \} I \mid S \{ \leq \mid < \mid \geq \mid > \mid = \mid \neq \} S \\
I &:= x_1 \mid x_2 \mid \dots \mid k \\
S &:= 0 \mid 1 \mid 2 \mid \dots \mid o
\end{aligned} \tag{A.4}$$

The nonterminal I derives all input variables, while the nonterminal S derives the output and all index constants appearing in the implementation. The grammar for `arrSearch2` (resp. `arrSearch3`) allows up to three (resp. four) disjuncts.

A.2 Synquid

We collected all the problems from the SYNQUID paper [123] that involve Lists, Binary Trees, and Binary Search Trees (BSTs), and that do not use other datatypes or higher-order functions. In these cases, we have a “ground-truth” property provided by the polymorphic refinement type used to synthesize the function. The reference code that we used for each problem is a version of the code synthesized by SYNQUID, rewritten in a C-style language. We modified `elemIndex` to return -1 instead of raising an exception when there is no matched element in the list, and added a new problem `deleteFirst` that deletes only the first occurrence of an element.

List

We considered 13 different methods for Lists constructed by `Nil` and `Cons` (we bounded the length of lists generated by list constructors to 10): `append`, `deleteFirst`, `delete`, `drop`, `elem`, `elemIndex`, `ith`, `min`, `replicate`, `reverse`, `snoc`, `stutter`, and `take`. Given the reference implementation from the SYNQUID benchmark, we synthesized properties in the

grammar:

$$\begin{aligned}
B &:= \top \mid AP \mid AP \vee AP \mid AP \vee AP \vee AP \\
AP &:= \text{isEmpty}(L) \mid \neg \text{isEmpty}(L) \mid \text{eq}(L, L) \mid \neg \text{eq}(L, L) \\
&\quad \mid S \{ \leq \mid < \mid \geq \mid > \mid = \mid \neq \} S + \dots + S + \{0 \mid 1\} \\
&\quad \mid \{ \forall \mid \exists \} x \in L. x \{ \leq \mid < \mid \geq \mid > \mid = \mid \neq \} I \\
&\quad \mid \langle \text{input variables} \rangle \mid \langle \text{output variables} \rangle \\
I &:= \langle \text{input variables} \rangle \mid \langle \text{output variables} \rangle \\
S &:= 0 \mid \text{len}(L) \mid \langle \text{input variables} \rangle \mid \langle \text{output variables} \rangle \mid \langle \text{constants from code} \rangle \\
L &:= \langle \text{input variables} \rangle \mid \langle \text{output variables} \rangle
\end{aligned} \tag{A.5}$$

The nonterminal I derives all input and output variables denoting list elements, the nonterminal S derives all integer input and output variables and constants, and the nonterminal L derives all list input and output variables in the implementation. Size comparisons are used only when the output variable is a list, index, or size. The number of occurrences of the nonterminal S on the right-hand side of a comparison is equal to the number of input variables of type list, index, or size.

In addition to the above 13 problems, `reverse_twice` synthesizes properties for the query, $ol_1 = \text{reverse}(l_1) \wedge ol_2 = \text{reverse}(l_2)$, using the following grammar:¹

$$\begin{aligned}
B &:= \top \mid AP \mid AP \vee AP \mid AP \vee AP \vee AP \\
AP &:= \text{isEmpty}(L) \mid \neg \text{isEmpty}(L) \mid \text{eq}(L, L) \mid \neg \text{eq}(L, L) \\
L &:= l_1 \mid l_2 \mid ol_1 \mid ol_2
\end{aligned} \tag{A.6}$$

¹Notice that in the example described in Section 4.1 we used a more complex grammar (Eq. (4.2)) that encompasses both the grammars above and can directly synthesize all the properties that are true for a single invocation of `reverse` in the first grammar as well as for a double invocation. We choose to use the grammar in Eq. (A.6) because we are interested in detecting properties that relate two invocations of `reverse`.

Binary Tree

The first set of Binary Tree problems consists of the method `elem` and the constructors `emptyTree` and `branch`. We provided the reference implementations from the SYNQUID benchmark, and synthesized properties expressed in the following grammar:

$$\begin{aligned}
B &:= \top \mid AP \mid AP \vee AP \mid AP \vee AP \vee AP \\
AP &:= \text{isEmpty}(T) \mid \neg \text{isEmpty}(T) \mid \text{eq}(T, T) \mid \neg \text{eq}(T, T) \\
&\quad \mid S \{ \leq \mid < \mid \geq \mid > \mid = \mid \neq \} S + \dots + S + \{0 \mid 1\} \\
&\quad \mid \{ \forall \mid \exists \} x \in T. x \{ \leq \mid < \mid \geq \mid > \mid = \mid \neq \} I \\
&\quad \mid \langle \text{input variables} \rangle \mid \langle \text{output variables} \rangle \\
I &:= \langle \text{input variables} \rangle \mid \langle \text{output variables} \rangle \\
S &:= 0 \mid \text{len}(T) \mid \langle \text{input variables} \rangle \mid \langle \text{output variables} \rangle \mid \langle \text{constants from code} \rangle \\
T &:= \langle \text{input variables} \rangle \mid \langle \text{output variables} \rangle
\end{aligned} \tag{A.7}$$

The nonterminal I derives all input and output variables for tree elements, the nonterminal S derives all integer input and output variables and constants, and the nonterminal T derives all tree input and output variables that appear in the implementation.

In the second set of Binary Tree problems we synthesize properties that capture relationships between the constructor `branch` and the three methods `rootval`, `left`, and `right`, which return the respective elements of `branch`. Properties are expressed in the following grammar:

$$\begin{aligned}
B &:= \top \mid AP \mid AP \vee AP \mid AP \vee AP \vee AP \\
AP &:= \text{isEmpty}(T) \mid \neg \text{isEmpty}(T) \mid \text{eq}(T, T) \mid \neg \text{eq}(T, T) \mid I \{ = \mid \neq \} I \\
I &:= \langle \text{input variables} \rangle \mid \langle \text{output variables} \rangle \\
T &:= \langle \text{input variables} \rangle \mid \langle \text{output variables} \rangle
\end{aligned} \tag{A.8}$$

The nonterminal I derives all input and output variables for tree elements, and the nonterminal T derives all input and output variables of type `Tree`.

Binary Search Tree

We considered four different methods for BSTs: `emptyBST`, `insert`, `delete`, and `elem`. We provided the reference implementations from the SYNQUID benchmarks, and synthesized properties expressed in the grammar in Eq. (A.7). We provided a data-structure constructor that can generate all valid BSTs up to size 4 using the following operations (n stands for an integer):

$$BST := \text{emptyBST}() \mid \text{insert}(BST, n) \quad (\text{A.9})$$

Other

We designed 14 problems to cover missing categories. (i) Stack and Queue are data structures implemented using other data structures (i.e., Lists), (ii) Integer Arithmetic cover more integer arithmetic problems, and in particular, cases where the input program uses loops.

Stack

We considered three Stack methods: `emptyStack`, `push` and `pop` and synthesized properties in the following grammar:

$$\begin{aligned} B &:= \top \mid AP \mid AP \vee AP \mid AP \vee AP \vee AP \\ AP &:= \text{isEmpty}(ST) \mid \neg \text{isEmpty}(ST) \mid \text{eq}(ST, ST) \mid \neg \text{eq}(ST, ST) \\ &\quad \mid S \{ \leq \mid < \mid \geq \mid > \mid = \mid \neq \} S + \{0 \mid 1\} \\ S &:= 0 \mid \text{len}(ST) \mid \langle \text{input variables} \rangle \mid \langle \text{output variables} \rangle \\ ST &:= \langle \text{input variables} \rangle \mid \langle \text{output variables} \rangle \end{aligned} \quad (\text{A.10})$$

The nonterminal S derives all input and output variables for indices or size, and the nonterminal ST derives all input and output variables of Stack type.

The data-structure constructor generates all stacks up to size 10 using the following operations:

$$ST := \text{emptyStack}() \mid \text{push}(ST, n) \quad (\text{A.11})$$

Finally, the problem `push_pop` synthesizes properties for the query $os_1 = \text{push}(s_1, x_1) \wedge (os_2, x_2) = \text{pop}(s_2)$ using the following grammar:

$$\begin{aligned}
B &:= \top \mid AP \mid AP \vee AP \mid AP \vee AP \vee AP \\
AP &:= I \{ = \mid \neq \} I \mid \text{isEmpty}(ST) \mid \neg \text{isEmpty}(ST) \mid \text{eq}(ST, ST) \mid \neg \text{eq}(ST, ST) \\
I &:= x_1 \mid x_2 \\
L &:= s_1 \mid s_2 \mid os_1 \mid os_2
\end{aligned} \tag{A.12}$$

Queue

We considered three Queue methods: `emptyQueue`, `enqueue` and `dequeue`. We defined a functional-style Queue using two Lists, and provided implementations of `emptyQueue`, `enqueue`, and `dequeue` using List methods. We also provided a function `toList` that converts a Queue to a single List in which elements of Queue are in the intended order, and synthesized properties expressed in the following grammar:

$$\begin{aligned}
B &:= \top \mid AP \mid AP \vee AP \mid AP \vee AP \vee AP \\
AP &:= \text{isEmpty}(L) \mid \neg \text{isEmpty}(L) \mid \text{eq}(L, L) \mid \neg \text{eq}(L, L) \\
L &:= \text{toList}(Q) \mid \text{cons}(I, \text{toList}(Q)) \mid \text{snoc}(\text{toList}(Q), I) \\
I &:= \langle \text{input variables} \rangle \mid \langle \text{output variables} \rangle \\
Q &:= \langle \text{input variables} \rangle \mid \langle \text{output variables} \rangle
\end{aligned} \tag{A.13}$$

The nonterminal I derives all input and output variables for Queue elements, and the nonterminal Q derives all input and output variables of type Queue. We provided a data-structure constructor that can generate all valid queues up to size 4 using the following operations:

$$Q := \text{emptyQueue}() \mid \text{enqueue}(Q, n) \mid \text{dequeue}(Q) \tag{A.14}$$

Integer Arithmetic

We considered three additional integer-manipulating functions: (i) $\text{abs}(x)$ computes the absolute value of the integer x (the semantics is expressible in SMT-Lib); (ii) $\text{linSum}(x)$ outputs the value of x using a loop that counts up to x ; (iii) $\text{nonlinSum}(x)$ computes the sum of all integers from 0 to the value of x using a loop.

First, we synthesize properties expressed in the following grammar where the nonterminal I derives all input and output variables, constants that appear in the code, and their negations:

$$\begin{aligned} B &:= \top \mid AP \mid AP \vee AP \mid AP \vee AP \vee AP \\ AP &:= I \{ \leq \mid < \mid \geq \mid > \mid = \mid \neq \} I \\ I &:= x \mid -x \mid o \mid -o \mid \langle \text{constants from code} \rangle \end{aligned} \tag{A.15}$$

Then we synthesize properties in the following grammar, which can express more general properties involving linear combinations of the integer terms of interest:

$$\begin{aligned} B &:= \top \mid AP \mid AP \vee AP \mid AP \vee AP \vee AP \\ AP &:= Ct_1 + \dots + Ct_n \{ \leq \mid < \mid \geq \mid > \mid = \mid \neq \} 0 \\ C &:= -3 \mid -2 \mid \dots \mid 3 \mid 4 \end{aligned} \tag{A.16}$$

The terms from t_1 to t_n are the input and output variables and constants that appear in the implementation. In the case of nonlinSum , the quadratic term $x \cdot x$ was also included.

Finally, for the case of $\text{abs}(x)$, we also considered the following grammar, which was introduced in Section 4.2.2 to illustrate a case in which the given language forms a well-quasi-order but is not finite.

$$\begin{aligned} AP &:= -20 \leq x \leq 10 \Rightarrow o \leq C \\ C &:= 0 \mid 1 + C \end{aligned} \tag{A.17}$$

A.3 Algebraic-Specification Benchmarks

For each module, we considered queries that were relevant to synthesize the algebraic properties described in the JLIBSKETCH benchmarks. We considered fairly small grammars as the module’s implementations contained many lines of code and were otherwise hard to perform synthesis for. For example, the implementation of HashMap contained 150 lines of code and involved arrays.

ArrayList

For the query $ov = \text{get}(\text{add}(arr, v), k)$, we considered the following grammar:

$$\begin{aligned}
 B &:= \top \mid G \Rightarrow ov = I \\
 G &:= \top \mid S \{ \leq \mid < \mid \geq \mid > \mid = \mid \neq \} S \\
 L &:= \text{emptyArray} \mid arr \\
 S &:= \text{len}(L) \mid \text{len}(L)+1 \mid k \mid k+1 \mid 0 \mid 1 \\
 I &:= v \mid \text{get}(arr, k)
 \end{aligned} \tag{A.18}$$

In this case, SPYRO synthesized the algebraic properties

$$\begin{aligned}
 \text{len}(arr) = k &\Rightarrow ov = v \\
 \text{len}(arr) \neq k &\Rightarrow ov = \text{get}(arr, k)
 \end{aligned} \tag{A.19}$$

For the query $n = \text{len}(\text{emptyArray})$, we considered the following grammar:

$$\begin{aligned}
 B &:= \top \mid n = S \\
 S &:= 0 \mid 1
 \end{aligned} \tag{A.20}$$

In this case, SPYRO synthesized the algebraic property

$$n = 0 \tag{A.21}$$

For the query $n = \text{len}(\text{add}(\text{arr}, v))$, we considered the following grammar:

$$\begin{aligned} B &:= \top \mid n = S \\ S &:= \text{size}(L) \mid \text{size}(L) + 1 \mid 0 \mid 1 \\ L &:= \text{emptyArray} \mid \text{arr} \end{aligned} \tag{A.22}$$

In this case, SPYRO synthesized the algebraic property

$$n = \text{len}(\text{arr}) + 1 \tag{A.23}$$

HashMap

We implemented the `KeyNotFound` exception of `get` using an additional output flag *err*—i.e., `get(m, k)` refers the output value when the error flag is not set to true, and *err* is true when any of the functions that called has thrown a `KeyNotFound` exception. As mentioned in Section 4.5.2, the initial mock implementation for `HashMap` provided in `JLIBSKETCH` was incorrect. In the rest of this section, we assume the correct mock implementation we fixed before synthesizing properties.

For the query $ov = \text{get}(\text{emptyMap}, k)$, we considered the following grammar:

$$B := \top \mid \text{err} \tag{A.24}$$

In this case, SPYRO synthesized the algebraic property

$$\text{err} \tag{A.25}$$

For the query $ov = \text{get}(\text{put}(m, k_1, v), k_2)$, we considered the following grammar:

$$\begin{aligned}
B &:= \top \mid err \mid ov = V \\
G &:= \top \mid K \{= \mid \neq\} K \\
M &:= \text{emptyMap} \mid m \\
K &:= k_1 \mid k_2 \\
V &:= v \mid \text{get}(M, K)
\end{aligned} \tag{A.26}$$

In this case, SPYRO synthesized the algebraic properties

$$\begin{aligned}
k_1 = k_2 &\Rightarrow ov = v \\
k_1 \neq k_2 &\Rightarrow ov = \text{get}(m, k_2)
\end{aligned} \tag{A.27}$$

For the query $om = \text{put}(\text{put}(m, k_1, v_1), k_2, v_2)$, we considered the following grammar:

$$\begin{aligned}
B &:= \top \mid G \Rightarrow \text{eq}(om, M) \\
G &:= \top \mid K \{= \mid \neq\} K \mid V \{= \mid \neq\} V \\
M &:= \text{emptyMap} \mid m \mid \text{put}(m, K, V) \\
K &:= k_1 \mid k_2 \\
V &:= v_1 \mid v_2
\end{aligned} \tag{A.28}$$

In this case, SPYRO synthesized the algebraic property

$$k_1 = k_2 \Rightarrow \text{eq}(om, \text{put}(m, k_1, v_2)) \tag{A.29}$$

HashSet

For the query $n = \text{size}(\text{emptySet})$, we considered the following grammar:

$$\begin{aligned} B &:= \top \mid n = S \\ S &:= 0 \mid 1 \end{aligned} \tag{A.30}$$

In this case, SPYRO synthesized the algebraic property

$$n = 0 \tag{A.31}$$

For the query $n = \text{size}(\text{add}(s, x))$, we considered the following grammar:

$$\begin{aligned} B &:= \top \mid G \Rightarrow n = S \\ G &:= \top \mid \text{contains}(s, x) \mid \neg \text{contains}(s, x) \\ S &:= \text{size}(s) \mid \text{size}(s) + 1 \mid 0 \mid 1 \end{aligned} \tag{A.32}$$

In this case, SPYRO synthesized the algebraic properties

$$\begin{aligned} \text{contains}(s, x) &\Rightarrow n = \text{size}(s) \\ \neg \text{contains}(s, x) &\Rightarrow n = \text{size}(s) + 1 \end{aligned} \tag{A.33}$$

For the query $b = \text{contains}(\text{emptySet}, x)$, we considered the following grammar:

$$\begin{aligned} B &:= \top \mid b = AP \\ AP &:= \top \mid \perp \end{aligned} \tag{A.34}$$

In this case, SPYRO synthesized the algebraic property

$$b = \perp \tag{A.35}$$

For the query $b = \text{contains}(\text{add}(s, x_1), x_2)$, we considered the following grammar:

$$\begin{aligned}
B &:= \top \mid G \Rightarrow b = AP \\
G &:= \top \mid I \{= \mid \neq\} I \\
I &:= x_1 \mid x_2 \\
AP &:= \text{contains}(s, I) \mid \top \mid \perp
\end{aligned} \tag{A.36}$$

In this case, SPYRO synthesized the algebraic properties

$$\begin{aligned}
x_1 = x_2 &\Rightarrow b = \top \\
x_1 \neq x_2 &\Rightarrow b = \text{contains}(s, x_2)
\end{aligned} \tag{A.37}$$

For the query $os = \text{remove}(\text{emptySet}, x)$, we considered the following grammar:

$$\begin{aligned}
B &:= \top \mid os = H \\
H &:= \text{emptySet} \mid \text{add}(H, x)
\end{aligned} \tag{A.38}$$

In this case, SPYRO synthesized the algebraic property

$$os = \text{emptySet} \tag{A.39}$$

For the query $os = \text{remove}(\text{add}(s, x_1), x_2)$, we considered the following grammar:

$$\begin{aligned}
B &:= \top \mid G \Rightarrow os = H \\
G &:= \top \mid I \{= \mid \neq\} I \\
I &:= x_1 \mid x_2 \\
H &:= \text{emptySet} \mid s \mid \text{add}(H, I) \mid \text{remove}(s, I)
\end{aligned} \tag{A.40}$$

In this case, SPYRO synthesized the algebraic properties

$$\begin{aligned} x_1 = x_2 &\Rightarrow os = \text{remove}(s, x_2) \\ x_1 \neq x_2 &\Rightarrow os = \text{add}(\text{remove}(s, x_2), x_1) \end{aligned} \tag{A.41}$$

A.4 Sensitivity-Analysis Benchmarks

We divide our benchmarks into ones that proved sensitivity properties with respect to the Hamming distance versus the edit distance over input and output lists.

Hamming Distance

In the following, D_h denotes the Hamming distance between two lists. For queries of the form $ol_1 = f(l_1, x_1) \wedge ol_2 = f(l_2, x_2)$, where $f \in \{\text{cons}, \text{deleteFirst}, \text{delete}, \text{snoc}\}$, we considered the following grammar:

$$\begin{aligned} B &:= \top \mid G \wedge \text{len}(l_1) = \text{len}(l_2) \wedge D_h(l_1, l_2) \leq d \Rightarrow D_h(ol_1, ol_2) \leq S \\ G &:= \top \mid x_1 \{ = \mid \neq \} x_2 \\ S &:= C * \text{len}(l_1) + C * d + C \\ C &:= -1 \mid 0 \mid 1 \mid 2 \end{aligned} \tag{A.42}$$

In this case, SPYRO synthesized the following sensitivity properties for cons:

$$\begin{aligned} \text{len}(l_1) = \text{len}(l_2) \wedge D_h(l_1, l_2) \leq d &\Rightarrow D_h(ol_1, ol_2) \leq \text{len}(l_1) + 1 \\ \text{len}(l_1) = \text{len}(l_2) \wedge D_h(l_1, l_2) \leq d &\Rightarrow D_h(ol_1, ol_2) \leq d + 1 \\ x_1 = x_2 \wedge \text{len}(l_1) = \text{len}(l_2) \wedge D_h(l_1, l_2) \leq d &\Rightarrow D_h(ol_1, ol_2) \leq \text{len}(l_1) \\ x_1 = x_2 \wedge \text{len}(l_1) = \text{len}(l_2) \wedge D_h(l_1, l_2) \leq d &\Rightarrow D_h(ol_1, ol_2) \leq d \end{aligned} \tag{A.43}$$

the following properties for deleteFirst,

$$\text{len}(l_1) = \text{len}(l_2) \wedge D_h(l_1, l_2) \leq d \Rightarrow D_h(ol_1, ol_2) \leq \text{len}(l_1) \tag{A.44}$$

the following properties for delete,

$$\text{len}(l_1) = \text{len}(l_2) \wedge D_h(l_1, l_2) \leq d \Rightarrow D_h(ol_1, ol_2) \leq \text{len}(l_1) \quad (\text{A.45})$$

and the following properties for snoc:

$$\begin{aligned} \text{len}(l_1) &= \text{len}(l_2) \wedge D_h(l_1, l_2) \leq d \Rightarrow D_h(ol_1, ol_2) \leq \text{len}(l_1) + 1 \\ \text{len}(l_1) &= \text{len}(l_2) \wedge D_h(l_1, l_2) \leq d \Rightarrow D_h(ol_1, ol_2) \leq d + 1 \\ x_1 = x_2 \wedge \text{len}(l_1) &= \text{len}(l_2) \wedge D_h(l_1, l_2) \leq d \Rightarrow D_h(ol_1, ol_2) \leq \text{len}(l_1) \\ x_1 = x_2 \wedge \text{len}(l_1) &= \text{len}(l_2) \wedge D_h(l_1, l_2) \leq d \Rightarrow D_h(ol_1, ol_2) \leq d \end{aligned} \quad (\text{A.46})$$

For queries of the form $ol_1 = f(l_1) \wedge ol_2 = f(l_2)$, with $f \in \{\text{reverse}, \text{stutter}, \text{tail}\}$, we considered the following grammar:

$$\begin{aligned} B &:= \top \mid \text{len}(l_1) = \text{len}(l_2) \wedge D_h(l_1, l_2) \leq d \Rightarrow D_h(ol_1, ol_2) \leq S \\ S &:= C * \text{len}(l_1) + C * d + C \\ C &:= -1 \mid 0 \mid 1 \mid 2 \end{aligned} \quad (\text{A.47})$$

In this case, SPYRO synthesized the following sensitivity properties for reverse,

$$\begin{aligned} \text{len}(l_1) &= \text{len}(l_2) \wedge D_h(l_1, l_2) \leq d \Rightarrow D_h(ol_1, ol_2) \leq \text{len}(l_1) \\ \text{len}(l_1) &= \text{len}(l_2) \wedge D_h(l_1, l_2) \leq d \Rightarrow D_h(ol_1, ol_2) \leq d \end{aligned} \quad (\text{A.48})$$

the following properties for stutter,

$$\begin{aligned} \text{len}(l_1) &= \text{len}(l_2) \wedge D_h(l_1, l_2) \leq d \Rightarrow D_h(ol_1, ol_2) \leq 2\text{len}(l_1) \\ \text{len}(l_1) &= \text{len}(l_2) \wedge D_h(l_1, l_2) \leq d \Rightarrow D_h(ol_1, ol_2) \leq 2d \end{aligned} \quad (\text{A.49})$$

and the following properties for tail:

$$\begin{aligned} \text{len}(l_1) = \text{len}(l_2) \wedge D_h(l_1, l_2) \leq d &\Rightarrow D_h(ol_1, ol_2) \leq \text{len}(l_1) - 1 \\ \text{len}(l_1) = \text{len}(l_2) \wedge D_h(l_1, l_2) \leq d &\Rightarrow D_h(ol_1, ol_2) \leq d \end{aligned} \quad (\text{A.50})$$

Properties for $ol_1 = \text{deleteFirst}(\text{cons}(l_1, x), x) \wedge ol_2 = \text{deleteFirst}(\text{cons}(l_2, x), x)$ are also expressed in the grammar Eq. (A.42). In this case, SPYRO synthesized the following sensitivity properties:

$$\begin{aligned} \text{len}(l_1) = \text{len}(l_2) \wedge D_h(l_1, l_2) \leq d &\Rightarrow D_h(ol_1, ol_2) \leq \text{len}(l_1) \\ \text{len}(l_1) = \text{len}(l_2) \wedge D_h(l_1, l_2) \leq d &\Rightarrow D_h(ol_1, ol_2) \leq d \end{aligned} \quad (\text{A.51})$$

For queries of the form $ol_1 = \text{append}(l_{11}, l_{12}) \wedge ol_2 = \text{append}(l_{21}, l_{22})$, we considered the following grammar:

$$\begin{aligned} B &:= \top \mid \text{len}(l_{11}) = \text{len}(l_{21}) \wedge \text{len}(l_{12}) = \text{len}(l_{22}) \wedge D_h(l_{11}, l_{21}) \leq d_1 \\ &\quad \wedge D_h(l_{11}, l_{21}) \leq d_2 \Rightarrow D_h(ol_1, ol_2) \leq S \\ S &:= C * \text{len}(l_{11}) + C * \text{len}(l_{12}) + C * d_1 + C * d_2 + C \\ C &:= -1 \mid 0 \mid 1 \mid 2 \end{aligned} \quad (\text{A.52})$$

In this case, SPYRO synthesized the following sensitivity properties:

$$\begin{aligned} \psi \wedge D_h(l_{11}, l_{21}) \leq d_2 &\Rightarrow D_h(ol_1, ol_2) \leq \text{len}(l_{11}) + d_2 \\ \psi \wedge D_h(l_{11}, l_{21}) \leq d_2 &\Rightarrow D_h(ol_1, ol_2) \leq \text{len}(l_{21}) + d_1 \\ \psi \wedge D_h(l_{11}, l_{21}) \leq d_2 &\Rightarrow D_h(ol_1, ol_2) \leq \text{len}(l_{11}) + \text{len}(l_{21}) \\ \psi \wedge D_h(l_{11}, l_{21}) \leq d_2 &\Rightarrow D_h(ol_1, ol_2) \leq d_1 + d_2 \end{aligned} \quad (\text{A.53})$$

where $\psi := \text{len}(l_{11}) = \text{len}(l_{21}) \wedge \text{len}(l_{12}) = \text{len}(l_{22}) \wedge D_h(l_{11}, l_{21}) \leq d_1 \wedge D_h(l_{11}, l_{21}) \leq d_2$.

Edit Distance

For queries of the form $ol_1 = f(l_1, x_1)$ and $ol_2 = f(l_2, x_2)$, with $f \in \{\text{cons}, \text{deleteFirst}, \text{delete}, \text{snoc}\}$, we considered the following grammar:

$$\begin{aligned}
 B &:= \top \mid G \wedge D_e(l_1, l_2) \leq d \Rightarrow D_e(ol_1, ol_2) \leq S \\
 G &:= \top \mid x_1 \{ = \mid \neq \} x_2 \\
 S &:= C * \text{len}(l_1) + C * \text{len}(l_2) + C * d + C \\
 C &:= -1 \mid 0 \mid 1 \mid 2
 \end{aligned} \tag{A.54}$$

In this case, SPYRO synthesized the following sensitivity properties for cons:

$$\top \tag{A.55}$$

the following properties for deleteFirst,

$$\begin{aligned}
 D_e(l_1, l_2) \leq d &\Rightarrow D_e(ol_1, ol_2) \leq \text{len}(l_1) + \text{len}(l_2) \\
 D_e(l_1, l_2) \leq d &\Rightarrow D_e(ol_1, ol_2) \leq \text{len}(l_1) + d \\
 D_e(l_1, l_2) \leq d &\Rightarrow D_e(ol_1, ol_2) \leq \text{len}(l_2) + d \\
 D_e(l_1, l_2) \leq d &\Rightarrow D_e(ol_1, ol_2) \leq d + 2 \\
 x_1 = x_2 \wedge D_e(l_1, l_2) \leq d &\Rightarrow D_e(ol_1, ol_2) \leq \text{len}(l_1) - \text{len}(l_2) + 2d \\
 x_1 = x_2 \wedge D_e(l_1, l_2) \leq d &\Rightarrow D_e(ol_1, ol_2) \leq -\text{len}(l_1) + \text{len}(l_2) + 2d \\
 x_1 = x_2 \wedge D_e(l_1, l_2) \leq d &\Rightarrow D_e(ol_1, ol_2) \leq d + 1
 \end{aligned} \tag{A.56}$$

the following properties for delete,

$$\begin{aligned}
D_e(l_1, l_2) \leq d &\Rightarrow D_e(ol_1, ol_2) \leq \text{len}(l_1) + \text{len}(l_2) \\
D_e(l_1, l_2) \leq d &\Rightarrow D_e(ol_1, ol_2) \leq \text{len}(l_1) + d \\
D_e(l_1, l_2) \leq d &\Rightarrow D_e(ol_1, ol_2) \leq \text{len}(l_2) + d \\
x_1 = x_2 \wedge D_e(l_1, l_2) \leq d &\Rightarrow D_e(ol_1, ol_2) \leq d
\end{aligned} \tag{A.57}$$

and the following properties for snoc:

$$\begin{aligned}
x_1 = x_2 \wedge D_e(l_1, l_2) \leq d &\Rightarrow D_e(ol_1, ol_2) \leq d \\
x_1 = x_2 \wedge D_e(l_1, l_2) \leq d &\Rightarrow D_e(ol_1, ol_2) \leq \text{len}(l_1) + \text{len}(l_2) \\
D_e(l_1, l_2) \leq d &\Rightarrow D_e(ol_1, ol_2) \leq d + 1 \\
D_e(l_1, l_2) \leq d &\Rightarrow D_e(ol_1, ol_2) \leq \text{len}(l_1) + \text{len}(l_2) + 1
\end{aligned} \tag{A.58}$$

For queries of the form $ol_1 = f(l_1) \wedge ol_2 = f(l_2)$, with $f \in \{\text{reverse}, \text{stutter}, \text{tail}\}$, we considered the following grammar:

$$\begin{aligned}
B &:= \top \mid D_e(l_1, l_2) \leq d \Rightarrow D_e(ol_1, ol_2) \leq S \\
S &:= C * \text{len}(l_1) + C * \text{len}(l_2) + C * d + C \\
C &:= -1 \mid 0 \mid 1 \mid 2
\end{aligned} \tag{A.59}$$

In this case, SPYRO synthesized the following sensitivity properties for reverse,

$$\begin{aligned}
D_e(l_1, l_2) \leq d &\Rightarrow D_e(ol_1, ol_2) \leq \text{len}(l_1) + \text{len}(l_2) \\
D_e(l_1, l_2) \leq d &\Rightarrow D_e(ol_1, ol_2) \leq d
\end{aligned} \tag{A.60}$$

the following properties for stutter,

$$\begin{aligned}
D_e(l_1, l_2) \leq d &\Rightarrow D_e(ol_1, ol_2) \leq 2\text{len}(l_1) + 2\text{len}(l_2) \\
D_e(l_1, l_2) \leq d &\Rightarrow D_e(ol_1, ol_2) \leq 2d
\end{aligned} \tag{A.61}$$

and the following properties for tail:

$$\begin{aligned} D_e(l_1, l_2) \leq d &\Rightarrow D_e(ol_1, ol_2) \leq \text{len}(l_1) + \text{len}(l_2) - 1 \\ D_e(l_1, l_2) \leq d &\Rightarrow D_e(ol_1, ol_2) \leq d \end{aligned} \quad (\text{A.62})$$

Properties for $ol_1 = \text{deleteFirst}(\text{cons}(l_1, x), x) \wedge ol_2 = \text{deleteFirst}(\text{cons}(l_2, x), x)$ are also expressed in the grammar Eq. (A.54). In this case, SPYRO synthesized the following sensitivity properties:

$$\begin{aligned} D_e(l_1, l_2) \leq d &\Rightarrow D_e(ol_1, ol_2) \leq \text{len}(l_1) + \text{len}(l_2) \\ x_1 = x_2 \wedge D_e(l_1, l_2) \leq d &\Rightarrow D_e(ol_1, ol_2) \leq d \end{aligned} \quad (\text{A.63})$$

For queries of the form $ol_1 = \text{append}(l_{11}, l_{12}) \wedge ol_2 = \text{append}(l_{21}, l_{22})$, we considered the following grammar:

$$\begin{aligned} B &:= \top \mid D_e(l_{11}, l_{21}) \leq d_1 \wedge D_e(l_{12}, l_{22}) \leq d_2 \Rightarrow D_e(ol_1, ol_2) \leq S \\ S &:= C * \text{len}(l_{11}) + C * \text{len}(l_{12}) + C * \text{len}(l_{21}) + C * \text{len}(l_{22}) + C * d_1 + C * d_2 + C \\ C &:= -1 \mid 0 \mid 1 \mid 2 \end{aligned} \quad (\text{A.64})$$

In this case, SPYRO synthesized the following sensitivity properties:

$$\begin{aligned} D_e(l_{11}, l_{21}) \leq d_1 \wedge D_e(l_{12}, l_{22}) \leq d_2 &\Rightarrow D_e(ol_1, ol_2) \leq d_1 + d_2 \\ D_e(l_{11}, l_{21}) \leq d_1 \wedge D_e(l_{12}, l_{22}) \leq d_2 &\Rightarrow D_e(ol_1, ol_2) \leq \text{len}(l_{12}) + \text{len}(l_{22}) + d_1 \\ D_e(l_{11}, l_{21}) \leq d_1 \wedge D_e(l_{12}, l_{22}) \leq d_2 &\Rightarrow D_e(ol_1, ol_2) \leq \text{len}(l_{11}) + \text{len}(l_{21}) + d_2 \\ D_e(l_{11}, l_{21}) \leq d_1 \wedge D_e(l_{12}, l_{22}) \leq d_2 &\Rightarrow D_e(ol_1, ol_2) \leq \text{len}(l_{11}) + \text{len}(l_{12}) + \text{len}(l_{21}) + \text{len}(l_{22}) \end{aligned} \quad (\text{A.65})$$

A.5 Bit-Vector Polyhedra Benchmarks

We considered the following 9 queries, showcasing a variety of behaviors, when synthesizing $\mathcal{L}$ -conjunctions of linear bit-vector inequalities:

$$\begin{aligned}
\text{square} & : y = x * x \\
\text{cube} & : y = x * x * x \\
\text{half} & : y = x / 2 \\
\text{squareIneq} & : y \leq x * x \\
\text{constNeq} & : y \neq 7 \\
\text{conjunction} & : (x + y + 4 \leq x + 15y + 7) \wedge (2x + 14y + 3 \leq x + 15y + 7) \\
\text{singlePoint} & : (x = 3 \wedge y = 7) \\
\text{fourPoints} & : (x = 3 \wedge y = 7) \vee (x = 9 \wedge y = 10) \vee (x = 6 \wedge y = 1) \vee (x = 0 \wedge y = 15) \\
\text{disjunction} & : (x + y + 4 \leq x + 15y + 7) \vee (2x + 14y + 3 \leq x + 15y + 7)
\end{aligned} \tag{A.66}$$

For each query, we used the following grammar:

$$\begin{aligned}
B & := Cx + Cy + C \leq Cx + Cy + C \\
C & := 0 \mid 1 \mid \dots \mid 15
\end{aligned} \tag{A.67}$$

Additional experiment

Fig. A.1 illustrates the application of join ($\sqcup$) to two formulas in $\mathcal{L}_{BV(\wedge)}$. Let $\psi_a, \psi_b \in \mathcal{L}_{BV(\wedge)}$ be the formulas from Fig. A.1(a) and Fig. A.1(b), respectively. Fig. A.1(c) shows the satisfying solutions of the disjunction $\psi_c =_{df} \psi_a \vee \psi_b$. Fig. A.1(d) shows the $\mathcal{L}_{BV(\wedge)}$ formula ψ_d computed by SPYRO for $\psi_a \sqcup \psi_b$, as well as the satisfying solutions of ψ_d . The join computation was performed by applying SPYRO to ψ_c . In this case, SPYRO showed that the result could be expressed with a single inequality, namely, $1 \leq 7x + 9y + 1$.

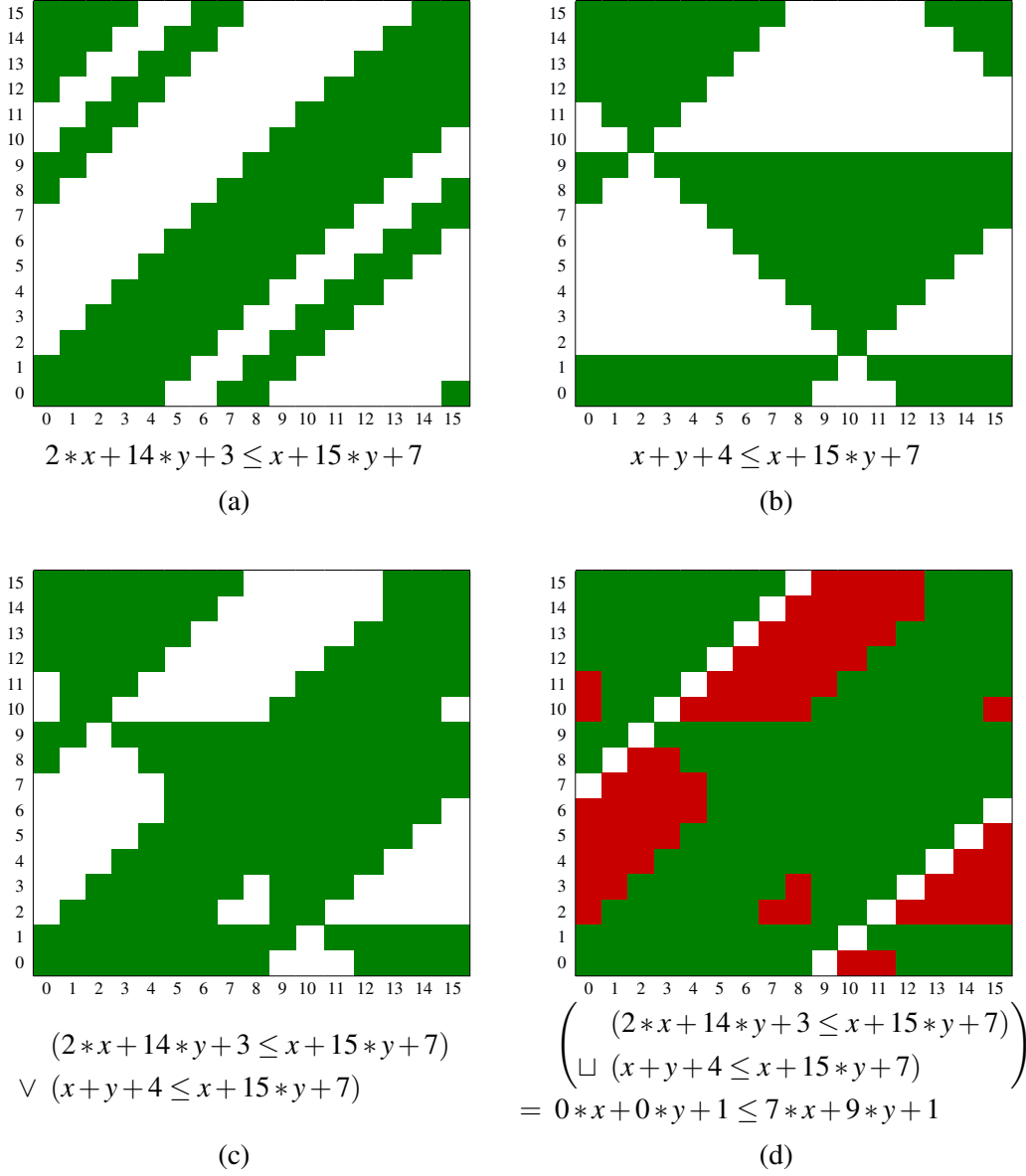


Figure A.1. Each green and red cell represents a solution in 4-bit unsigned modular arithmetic of the indicated formula. In (d), the occurrences of red cells are points that do not satisfy (c), but are needed for a conjunctive formula to over-approximate (c).

A.6 Detailed Evaluation for Each Application

Table A.1 shows how many times each query is performed for each benchmark in our first application in Section 4.5.1 and the total running time spent performing each type of query. Tables A.2-A.4 show the same metrics for the other three applications. “Last iter.” denotes the

Table A.1. Evaluation results of SPYRO for SyGuS and LIA specification-mining. The Enum. column reports the estimated time required to run CHECKSOUNDNESS for all formulas in the DSL $\mathcal{L}$. This estimation is achieved by multiplying the size of the grammar by the average running time of the CHECKSOUNDNESS. A (*) indicates a timeout when attempting to prove precision in the last iteration. In that case, we report as total time the time at which SPYRO timed out.

Problem	$ \mathcal{L} $	SYNTHESIZE		CHECK SOUNDNESS		CHECK PRECISION		Last Iter.	Enum.	Total	
		Num	T(sec)	Num	T(sec)	Num	T(sec)				T(sec)
SYGUS	max2	$1.57 \cdot 10^5$	6	0.13	12	0.06	9	1.37	0.05	787.32	1.55
	max3	$8.85 \cdot 10^5$	19	13.83	48	3.53	36	131.44	0.46	$6.51 \cdot 10^4$	148.79
	max4	$5.06 \cdot 10^8$	-	-	-	-	-	-	-	-	-
	diff	$5.66 \cdot 10^7$	18	19.41	41	1.06	28	236.61	0.48	$1.46 \cdot 10^6$	257.08
	diff,diff	$3.73 \cdot 10^5$	-	-	-	-	-	-	-	-	-
	arrSearch2	$3.37 \cdot 10^6$	16	3.44	41	1.44	31	60.97	0.28	$1.19 \cdot 10^5$	65.84
	arrSearch3	$3.66 \cdot 10^9$	-	-	-	-	-	-	-	-	-
LIA	abs (A.15)	$3.37 \cdot 10^6$	6	1.14	14	0.76	11	10.16	0.38	$1.83 \cdot 10^5$	12.05
	abs (A.16)	$2.90 \cdot 10^{11}$	-	-	-	-	-	-	-	-	-
	abs (A.17)	∞	22	3.98	23	0.15	3	0.67	0.70	∞	4.80

time taken to perform the final call to CHECKPRECISION from the final call on SYNTHPROPERTY at line 5 of Alg. 9. That call on SYNTHPROPERTY is the most difficult one because it implicitly establishes that the synthesized $\mathcal{L}$ -conjunction is indeed a best one.

Table A.2. Evaluation results of SPYRO for data structures and arithmetic specification-mining. The Enum. column reports the estimated time required to run CHECKSOUNDNESS for all formulas in the DSL $\mathcal{L}$. This estimation is achieved by multiplying the size of the grammar by the average running time of the CHECKSOUNDNESS. A (*) indicates a timeout when attempting to prove precision in the last iteration. In that case, we report as total time the time at which SPYRO timed out.

	Problem	$ \mathcal{L} $	SYNTHESIZE		CHECK SOUNDNESS		CHECK PRECISION		Last Iter. T(sec)	Enum. T(sec)	Total T(sec)
			Num	T(sec)	Num	T(sec)	Num	T(sec)			
List	append	$4.97 \cdot 10^8$	29	46.62	68	81.50	43	91.46	57.59	$5.95 \cdot 10^8$	219.58
	deleteFirst	$2.99 \cdot 10^6$	22	33.91	64	72.05	50	100.63	9.10	$3.36 \cdot 10^6$	206.60
	delete	$2.99 \cdot 10^6$	24	39.49	66	87.15	49	107.07	18.41	$3.94 \cdot 10^6$	233.71
	drop	$4.75 \cdot 10^8$	26	38.55	60	66.00	37	71.48	45.89	$5.22 \cdot 10^8$	176.03
	elem	4,096	8	9.05	18	17.36	12	15.71	7.04	3950.36	42.12
	elemIndex	$8.74 \cdot 10^6$	22	28.62	53	52.87	35	53.96	17.22	$8.72 \cdot 10^6$	135.45
	ith	$1.91 \cdot 10^6$	21	28.72	46	45.41	29	47.21	23.40	$1.88 \cdot 10^6$	121.34
	min	$2.38 \cdot 10^5$	5	6.02	14	13.71	11	14.42	2.53	$2.38 \cdot 10^5$	34.15
	replicate	$1.82 \cdot 10^6$	5	6.04	20	19.25	17	22.59	6.59	$1.75 \cdot 10^6$	47.87
	reverse	$1.73 \cdot 10^6$	6	7.28	16	18.11	12	16.84	7.83	$1.96 \cdot 10^6$	42.23
	reverse, reverse	$6.40 \cdot 10^4$	20	26.64	52	96.65	40	76.37	32.21	$1.19 \cdot 10^5$	199.66
	snoc	$2.99 \cdot 10^6$	17	25.03	47	53.42	37	68.81	19.72	$3.39 \cdot 10^6$	147.26
	stutter	$1.73 \cdot 10^6$	6	7.69	23	24.80	20	28.65	2.44	$1.86 \cdot 10^6$	61.14
	take	$8.49 \cdot 10^6$	12	15.1	35	40.08	26	41.35	7.91	$9.72 \cdot 10^6$	96.53
Binary Tree	emptyTree	$1.41 \cdot 10^5$	2	2.1	6	6.41	5	6.21	2.23	$1.50 \cdot 10^5$	14.73
	branch	$5.68 \cdot 10^8$	134	1,487.03	242	340.10	118	1,938.85	*	$8.00 \cdot 10^9$	3,765.98
	elem	$2.62 \cdot 10^5$	12	15.15	27	30.79	17	26.22	13.69	$2.99 \cdot 10^5$	72.17
	branch, left	$2.16 \cdot 10^5$	38	67.7	91	134.30	61	156.89	70.24	$3.19 \cdot 10^5$	358.89
	branch, right	$2.16 \cdot 10^5$	40	69.85	95	134.88	63	149.51	20.75	$3.07 \cdot 10^5$	354.24
	branch, rootval	$2.62 \cdot 10^5$	36	67.19	72	101.16	41	93.45	84.36	$3.68 \cdot 10^5$	261.81
BST	emptyBST	$1.41 \cdot 10^5$	2	2.16	7	6.13	6	6.92	1.92	$1.23 \cdot 10^5$	15.21
	insert	$2.99 \cdot 10^6$	16	24.16	52	60.63	43	118.81	26.52	$3.48 \cdot 10^6$	203.60
	delete	$2.99 \cdot 10^6$	17	24.75	49	68.29	38	109.10	31.50	$4.16 \cdot 10^6$	202.15
	elem	$2.87 \cdot 10^5$	11	13.7	22	21.57	13	19.52	22.10	$2.82 \cdot 10^5$	54.79
Stack	emptyStack	$1.25 \cdot 10^5$	2	2.12	6	5.61	5	5.89	2.05	$1.17 \cdot 10^5$	13.63
	push	$1.73 \cdot 10^6$	4	4.9	10	11.16	7	9.73	2.50	$1.91 \cdot 10^6$	25.79
	pop	$1.73 \cdot 10^6$	4	4.84	11	12.16	8	10.97	12.02	$1.93 \cdot 10^6$	27.98
	push, pop	$2.62 \cdot 10^5$	32	51.12	77	95.72	53	107.13	27.66	$3.26 \cdot 10^5$	253.97
Queue	emptyQueue	64	2	2.18	6	5.86	5	9.81	2.07	62.51	17.85
	enqueue	$5.93 \cdot 10^5$	4	5.4	11	16.82	8	270.26	200.15	$9.06 \cdot 10^5$	292.48
	dequeue	$5.93 \cdot 10^5$	4	5.51	9	12.51	6	290.28	192.02	$8.24 \cdot 10^5$	308.30
Arithmetic	linSum (A.15)	$1.01 \cdot 10^7$	7	6.69	20	14.50	15	15.33	5.33	$7.31 \cdot 10^6$	36.52
	linSum (A.16)	$2.90 \cdot 10^{10}$	15	15.31	99	71.72	88	112.87	2.70	$2.10 \cdot 10^{10}$	199.90
	nonlinSum (A.15)	$1.01 \cdot 10^7$	8	7.95	30	21.58	25	26.48	2.53	$7.25 \cdot 10^6$	56.01
	nonlinSum (A.16)	$1.48 \cdot 10^{13}$	17	16.82	121	102.71	107	734.23	48.21	$1.26 \cdot 10^{13}$	853.77

Table A.3. Evaluation results of SPYRO for modular specification synthesis. The Enum. column reports the estimated time required to run CHECKSOUNDNESS for all formulas in the DSL $\mathcal{L}$. This estimation is achieved by multiplying the size of the grammar by the average running time of the CHECKSOUNDNESS. A (*) indicates a timeout when attempting to prove precision in the last iteration. In that case, we report as total time the time at which SPYRO timed out.

Problem		$ \mathcal{L} $	SYNTHESIZE		CHECK SOUNDNESS		CHECK PRECISION		Last Iter.	Enum.	Total
			Num	T(sec)	Num	T(sec)	Num	T(sec)			
ArrayList	get, add	1,156	13	4.19	23	6.87	13	5.00	5.39	345.29	16.06
	len, emptyArray	3	3	0.55	5	1.34	4	1.23	0.56	0.80	3.12
	len, add	1,303	5	1.26	9	2.54	6	2.07	1.86	367.74	5.88
HashMap	get, emptyMap	2	2	0.67	6	2.53	5	2.40	0.80	0.84	5.60
	get, put	1,860	14	6.22	27	295.04	16	189.98	119.87	$2.03 \cdot 10^4$	491.24
	put, put	103	4	1.58	9	217.01	6	485.72	213.21	2483.56	704.32
HashSet	size, emptySet	3	2	0.62	6	1.80	5	1.62	0.62	0.90	4.04
	size, add	1,315	7	2.39	14	4.56	9	3.40	0.70	428.31	10.36
	contains, emptySet	3	2	0.60	6	1.80	5	1.64	0.64	0.90	4.04
	contains, add	55	5	1.64	12	3.94	9	3.28	0.69	18.06	8.86
	remove, emptySet	5	1	0.34	5	1.05	5	1.78	0.55	1.05	3.17
	remove, add	181	7	3.38	15	6.20	10	13.23	3.86	74.81	22.81

Table A.4. Evaluation results of SPYRO for sensitivity analysis and abstract domains. The Enum. column reports the estimated time required to run CHECKSOUNDNESS for all formulas in the DSL $\mathcal{L}$. This estimation is achieved by multiplying the size of the grammar by the average running time of the CHECKSOUNDNESS. A (*) indicates a timeout when attempting to prove precision in the last iteration. In that case, we report as total time the time at which SPYRO timed out.

	Problem	$ \mathcal{L} $	SYNTHESIZE		CHECK SOUNDNESS		CHECK PRECISION		Last Iter. T(sec)	Enum. T(sec)	Total T(sec)
			Num	T(sec)	Num	T(sec)	Num	T(sec)			
Sensitivity Hamming Dist.	append	1,025	25	11.24	55	25.88	34	19.46	0.98	482.31	56.58
	cons	193	14	5.21	34	12.08	24	10.55	0.80	68.57	27.84
	cons_delete	193	10	3.68	24	9.02	16	7.29	2.44	72.54	20.0
	deleteFirst	193	13	4.97	26	10.03	14	6.48	5.81	74.45	21.48
	delete	193	15	5.99	28	11.69	14	6.56	2.45	161.02	24.24
	reverse	65	7	2.47	15	9.82	10	4.10	0.79	42.55	16.39
	snoc	193	17	6.62	37	14.55	24	11.22	0.81	75.90	32.39
	stutter	65	7	2.47	17	6.38	12	4.95	2.25	24.39	13.79
	tail	65	7	2.39	19	6.61	14	5.52	2.2	22.61	14.52
Sensitivity Edit Dist.	append	16,385	43	29.18	82	118.52	43	55.27	7.95	$2.37 \cdot 10^4$	202.97
	cons	769	6	2.49	10	4.18	4	2.02	8.69	321.44	8.69
	cons_delete	769	19	9.00	36	15.80	19	12.70	11.75	337.51	37.50
	deleteFirst	769	33	17.9	68	115.43	42	31.53	1.31	1385.38	164.86
	delete	769	26	13.42	55	190.48	33	23.55	3.64	2663.26	227.45
	reverse	257	13	6.00	27	61.43	16	10.11	3.32	594.24	77.53
	snoc	769	29	15.44	61	37.21	36	25.72	5.80	700.07	78.38
	stutter	257	14	6.29	26	13.66	14	8.51	3.03	135.02	28.47
	tail	257	14	6.17	29	12.83	17	10.01	9.66	113.70	29.01
Bit-Vector Polyhedra	square	$1.68 \cdot 10^7$	20	14.79	76	24.65	60	136.76	68.99	$5.44 \cdot 10^6$	176.19
	cube	$1.68 \cdot 10^7$	21	14.26	71	22.85	55	195.03	115.98	$6.97 \cdot 10^6$	232.14
	half	$1.68 \cdot 10^7$	19	16.17	56	19.27	40	353.14	*	$5.77 \cdot 10^6$	388.57
	squareIneq	$1.68 \cdot 10^7$	45	105.24	70	20.24	28	95.41	98.74	$4.85 \cdot 10^6$	220.89
	constNeq	$1.68 \cdot 10^7$	46	65.37	63	20.23	18	69.83	75.32	$5.39 \cdot 10^6$	155.43
	conjunction	$1.68 \cdot 10^7$	46	73.08	67	20.49	23	102.32	117.45	$5.13 \cdot 10^6$	195.89
	singlePoint	$1.68 \cdot 10^7$	9	14.61	75	21.64	70	202.59	5.95	$4.84 \cdot 10^6$	238.84
	fourPoints	$1.68 \cdot 10^7$	4	1.32	56	14.75	54	72.47	0.94	$4.42 \cdot 10^6$	88.53
	disjunction	$1.68 \cdot 10^7$	48	178.05	62	22.69	15	102.73	105.71	$6.14 \cdot 10^6$	303.47

Appendix B

Specification Synthesis of Nondeterministic Program Evaluation Details

B.1 Type I Test Set: Mining Under-Approximated Specifications

The goal of this section is to evaluate ASPIRE’s general capability to mine *under-approximated specifications* for deterministic programs. Park et al. [116] mined $\mathcal{L}$ -consequences for programs used in the synthesis literature. Their benchmark set consists of 45 programs paired with corresponding grammars that their tool SPYRO uses to mine $\mathcal{L}$ -consequences. Note that the queries for all the SPYRO benchmarks *do not* contain existential quantifiers.

We successfully recomputed $\mathcal{L}$ -consequences produced by SPYRO for all benchmarks. However, we found that ASPIRE’s implementation was on average 2.2x faster (geometric mean) than SPYRO at computing $\mathcal{L}$ -consequences for deterministic programs. Although ASPIRE and SPYRO implement the same algorithm for computing $\mathcal{L}$ -consequences for deterministic programs, ASPIRE’s implementation of CHECKPRECISION forces the generated example to be negative (using an assertion), whereas SPYRO’s implementation generates an example and then checks later if it is negative. The implementation avoids spurious calls to CHECKPRECISION and is therefore faster.

We focus on the new capability of ASPIRE to compute $\mathcal{L}$ -implicants. To compute $\mathcal{L}$ -implicants for the deterministic benchmarks used to evaluate SPYRO, we modified the top-level

production of each DSL used in the SPYRO evaluation to consider conjunctions instead of disjunctions—i.e., if the top-level production was of the form $P \rightarrow AP \vee AP \vee \dots$, we replaced it with $P \rightarrow AP \wedge AP \wedge \dots$. We denote the grammar before modification as $\mathcal{O}$ and after modification as $\mathcal{U}$.

Of the 45 benchmarks used when evaluating SPYRO, we only consider 35 benchmarks for which the semantics of operations were expressed using SKETCH. The 10 benchmarks we don’t consider are simple ones for which the semantics are expressed directly using SMT formulas instead of SKETCH; ASPIRE does not support SMT semantics yet. The 35 benchmarks consist of: 7 programs from the CLIA track of the SYGUS competition [4], 22 programs manipulating data structures taken from the SYNQUID synthesizer [123], and 6 imperative programs designed by the authors of SPYRO. For the SYGUS problems and imperative programs, the DSLs can express arithmetic comparison between variables—e.g. for the query $o = \text{abs}(x)$, one would get the $\mathcal{O}$ -consequence $-x \geq x \Rightarrow x = -o$. For SYNQUID problems, the DSLs support common functions over the data structures appearing in each specific benchmark —e.g., length for lists. For the query $l_{out} = \text{reverse}(l_{in})$ one would get $\text{len}(l_{out}) = \text{len}(l_{in})$ as one of the $\mathcal{O}$ -consequences.

ASPIRE could synthesize properties for 35/35 benchmarks, and guaranteed that all of them were best $\mathcal{U}$ -implicants. Table B.1 shows the evaluation details of a few selected benchmarks.

Overall, ASPIRE could solve each of the 35 benchmarks in under 7 minutes (the largest grammar contained $1.48 \cdot 10^{13}$ properties). Next, we analyze the results for each subcategory separately.

SYGUS Benchmarks

For the 7 SYGUS benchmarks, we found that the synthesized $\mathcal{U}$ -implicants exactly coincides with the semantics of the given queries. When we replicated the experiment by Park et al. [116], we could only synthesize $\mathcal{O}$ -consequences for 4/7 benchmarks (like SPYRO, ASPIRE failed on max4, max5, and arrSearch3), but for these 4 benchmarks the synthesized properties also coincided with the semantics of the given queries. The result shows that the grammar $\mathcal{U}$

Table B.1. Evaluation of ASPIRE for mining specifications. $|\mathcal{U}|$ is the size of grammar. LoC is the number of lines of SKETCH code used to write the semantics of programs and operators. #P is the number of $\mathcal{L}$ -implicants synthesized by ASPIRE. Num and T(s) are the number of times each primitive was called and the total time it consumed, respectively. For SYNQUID benchmarks, we only show 5 with the longest running time.

	Problem	$ \mathcal{U} $	LoC	#P	CHECKIMPLICATION		CHECKWEAKEST		SYNTHESIZE		Total.
					Num	T(s)	Num	T(s)	Num	T(s)	
SYGUS	max2	$1.60 \cdot 10^5$	7	2	13	0.44	10	0.82	5	0.12	1.51
	max3	$8.94 \cdot 10^5$	15	3	30	0.53	15	1.66	18	0.73	2.92
	max4	$5.10 \cdot 10^8$	31	4	89	2.02	34	18.38	59	10.90	31.32
	max5	$4.72 \cdot 10^1$	63	5	189	5.33	56	136.77	138	150.14	292.26
	diff	$5.68 \cdot 10^7$	7	2	28	0.45	11	7.45	19	0.93	8.84
	arrSearch2	$1.01 \cdot 10^6$	9	3	34	0.50	14	0.99	23	1.21	2.71
	arrSearch3	$7.28 \cdot 10^8$	11	4	86	1.39	27	14.78	63	20.55	36.84
Arithmetic	abs-1	$1.01 \cdot 10^6$	7	2	17	0.27	10	0.35	9	0.20	0.83
	abs-2	$2.90 \cdot 10^{10}$	7	2	29	0.45	11	4.29	20	1.17	5.92
	linSum-1	$3.01 \cdot 10^6$	9	2	7	0.11	4	0.07	4	0.06	0.25
	linSum-2	$2.90 \cdot 10^{10}$	9	2	41	0.66	17	3.64	27	2.87	7.18
	nonlinSum-1	$3.01 \cdot 10^6$	9	2	22	0.36	12	0.67	12	0.35	1.39
	nonlinSum-2	$1.48 \cdot 10^{13}$	9	2	70	9.07	16	378.68	56	95.40	483.17
SYNQUID	append	$4.97 \cdot 10^8$	75	2	82	3.50	22	58.07	62	40.84	102.56
	deleteFirst	$3.01 \cdot 10^6$	83	2	51	1.86	19	254.67	34	20.81	277.39
	delete	$3.01 \cdot 10^6$	82	2	48	2.12	16	39.26	34	19.64	61.17
	drop	$4.75 \cdot 10^8$	75	2	79	2.42	22	37.67	59	39.04	79.16
	enqueue	$6.00 \cdot 10^5$	147	1	8	17.98	5	48.44	4	0.56	67.02

and $\mathcal{O}$ of the benchmarks are expressive enough to synthesize *exact* approximations—i.e., the properties are semantically equivalent to the query. Take $o = \max2(x_1, x_2)$ as an example: the synthesized $\mathcal{O}$ -consequences are $\{o = x_1 \vee o = x_2, x_2 = x_1 \vee o > x_1 \vee o > x_2\}$, and the synthesized $\mathcal{U}$ -implicants are $\{o = x_1 \wedge x_2 < o, o = x_2 \wedge x_1 < o\}$, which can be proved to be equivalent. Due to the equivalence of the $\mathcal{U}$ -implicants and $\mathcal{O}$ -consequences, we further compared the efficiency of computing over-approximations and under-approximations for these benchmarks to assess which formulas were easier to compute. Although the size of grammar for $\mathcal{O}$ -consequences and $\mathcal{U}$ -implicants are the same (since we only replace $\vee$ by $\wedge$ at the top layer), synthesizing $\mathcal{U}$ -implicants was on average 1.6x faster (geometric mean) than synthesizing $\mathcal{O}$ -consequences for the 4 benchmarks on which both synthesis processes terminated. Furthermore, while SPYRO (and also ASPIRE) failed to synthesize $\mathcal{O}$ -consequences for `max4`, `max5`, and `arrSearch3` within the given timeout, ASPIRE successfully synthesized $\mathcal{U}$ -implicants in less than 5 minutes for each of these benchmarks.

We compared the $\mathcal{U}$ -implicants to $\mathcal{O}$ -consequences to see how they differed. For the query $o = \max3(x_1, x_2, x_3)$, for example, the synthesized $\mathcal{O}$ -consequences consist of 5 conjuncts

$$\begin{aligned} x_2 < o \vee x_1 < o \vee x_2 = x_1 & \quad x_3 \leq o & \quad o = x_2 \vee o = x_1 \vee x_1 < x_3 \\ o = x_2 \vee o = x_3 \vee x_3 < x_1 & \quad x_2 < x_3 \vee o = x_2 \vee x_2 < x_1 \end{aligned} \quad (\text{B.1})$$

while the synthesized $\mathcal{U}$ -implicants consist of 3 disjuncts

$$x_1 = o \wedge x_3 \leq x_1 \wedge x_2 \leq x_1 \quad x_2 = o \wedge x_3 \leq x_2 \wedge x_1 \leq x_2 \quad x_3 = o \wedge x_1 \leq x_3 \wedge x_2 \leq x_3 \quad (\text{B.2})$$

The $\mathcal{O}$ -consequences in Eq. B.1 can be thought of as a declarative specification that `max3` function much to satisfy, and in fact similar to the specifications provided in SYGUS benchmarks. Instead, the $\mathcal{U}$ -implicants in Eq. B.2 captures the paths of the program and the output they produce, which one would obtain via symbolic execution.

To summarize, under-approximation computed semantically equivalent specifications for SYGUS benchmarks, but it was faster than over-approximation. This improvement could be attributed to the fact that fewer $\mathcal{U}$ -implicants could capture the semantics of a program when compared to $\mathcal{O}$ -consequences—e.g. 3 $\mathcal{U}$ -implicants vs 5 $\mathcal{O}$ -consequences for `max3`.

Imperative Program Benchmarks

These 6 benchmarks ask ASPIRE to find linear or nonlinear relations to capture the semantics of simple imperative programs. ASPIRE synthesized $\mathcal{U}$ -implicants that exactly capture the program semantics except for `nonlinSum1`, for which the DSL can only describe linear relations, but the program semantics can only be captured by a quadratic relation.

SYNQUID Benchmarks

Many of the $\mathcal{L}$ -implicants synthesized by ASPIRE for this benchmark category were not particularly useful. Specifically, the DSLs are not expressive enough for computing useful $\mathcal{L}$ -implicants, mostly yielding trivial properties involving cases in which the data structure is empty. For example, for the query $l_{out} = \text{reverse}(l_{in})$, ASPIRE only synthesized the $\mathcal{U}$ -implicant $\{l_{out} = l_{in} \wedge \text{len}(l) \leq 1\}$, which only describes the behavior of the function `reverse` on lists of length less than 1. While the original DSLs were useful for computing consequences (e.g., $\text{len}(l_{in}) = \text{len}(l_{out})$), their dual versions are too weak to reason about implicants—one would instead need to talk about more specific position information of elements in lists. The same problem holds for other data structures.

Finding: ASPIRE can mine $\mathcal{L}$ -consequences and $\mathcal{L}$ -implicants that help understand program behaviors. If the DSL is not expressive enough, ASPIRE cannot produce useful $\mathcal{L}$ -implicants.

B.2 Type II Test Set: Simple nondeterministic programs

In this section, we evaluate ASPIRE’s ability to reason about nondeterministic programs. Nondeterminism can be modeled using existential quantifiers, a key innovation of the LOUD

framework. Consider a program that takes as input an integer x and then adds a nondeterministically chosen positive number to it. Such a program can be modeled using the query “ $\exists h. o = x + h$ ” where h is a positive number. A consequence of this query is the property “ $o \geq x$ ”, which holds for all possible values of h . An implicant of the same query is the property “ $o = x + 17$ ”, which holds when $h = 17$.

From these two example properties (which can be synthesized by ASPIRE when given proper DSLs), we observe that for nondeterministic programs, consequences hold for *every possible* nondeterministic choice (the demonic perspective of nondeterminism), whereas implicants hold for *at least one* nondeterministic choice (the angelic perspective). To model a nondeterministic program, we introduce an array of nondeterministic values H into the query as an existentially quantified variable. Whenever the program execution reaches a nondeterministic command (e.g. `nondet()` is called), the command takes the next value of the array H .

Benchmark Selection and Quantitative Results

We collected 12 benchmarks involving nondeterminism: (i) we created 4 nondeterministic sorting algorithms where the goal is to synthesize properties that characterize when the algorithm works or does not work as intended; (ii) we collected the 4 nondeterministic recursive programs for which the goal is to synthesize a polynomial invariant by Chatterjee et al. [27] (all other benchmarks by Chatterjee et al. are deterministic), and (iii) we collected 4 SV-COMP [17] benchmarks in the bitvector category where nondeterministic values are used to model unknown inputs and parameters that determine control flow in the program (all others SV-COMP benchmarks are deterministic or unsuitable for specification synthesis).

For each category of benchmarks, we acted as the users of LOUD and designed the DSLs following such a methodology: we included constants, free variables that appear in the query, basic arithmetic operations between them, and also several atomic functions and predicates that were related to the problem or we expected the resulting properties to contain. We describe the details of the DSLs in Section B.2.

Table B.2. Evaluation results of Type II test set. $|\exists|$ is the size of the domain of the existentially quantified variables. #P and T(s) are the number of properties and synthesis time for both $\mathcal{L}$ -consequences and $\mathcal{L}$ -implicants (- denotes timeouts).

	Problem	LoC	$ \exists $	$\mathcal{L}$ -cons.		$\mathcal{L}$ -impl.	
				#P	T(s)	#P	T(s)
Nondeterminism	shuffle3	33	4	1	0.14	1	1.11
	shuffle4	33	27	1	0.30	1	1.17
	shuffle5	33	256	1	33.19	1	15.26
	rsum	28	$\sim 10^5$	1	5.51	1	10.29
	rsqsum	28	$\sim 10^5$	1	11.25	2	4.53
	rcubsum	28	$\sim 10^5$	1	4.07	2	4.80
	mergesort	31	$\sim 10^7$	-	-	-	-
	jain1	27	$\sim 10^{12}$	1	0.91	1	0.63
	jain2	29	$\sim 10^{12}$	2	26.94	1	27.19
	jain4	34	$\sim 10^{12}$	3	17.34	1	37.10
	jain6	34	$\sim 10^{12}$	3	161.80	1	131.91
	bubble3	33	64	6	9.63	8	7.34
	bubble4	33	6,561	17	392.00	22	136.56
	swap3	33	$\sim 10^7$	7	9.30	6	7.27
	swap4	33	$\sim 10^9$	24	164.61	19	69.77

ASPIRE synthesizes $\mathcal{O}$ -consequences and $\mathcal{U}$ -implicants for 11/12 benchmarks. For those solved benchmarks, ASPIRE takes less than 400 seconds to synthesize $\mathcal{O}$ -consequences and less than 140 seconds to synthesize $\mathcal{U}$ -implicants. Evaluation details are shown in Table B.2.

Because these benchmarks involve existential quantifiers, they require Algorithms 14 and 15, the CEGQI algorithms presented in Section 5.5. Algorithms 14 and 15 avoided exploring a large portion of the space of values for existentially quantified variables—e.g., for the benchmark `bubble4` each call to Algorithms 14 and 15 terminated with at most 13 instances h in H , instead of considering all $3^8 = 6561$ nondeterministic instances. When we reuse the instance h generated across all calls to `CHECKCANDIDATENEGEX`, the total number was only 17. This result inspired us to create a version of our algorithm that caches instances and reuses them across different calls to `CHECKCANDIDATENEGEX`. Caching and reusing instances results, on average, in a 1.4x speedup.

Qualitative Evaluation

We discuss each benchmark in detail.

Random Shuffle

The `shufflen` ($n \in \{3, 4, 5\}$) benchmarks model a random shuffle function that, given an array $l = [a_1, \dots, a_n]$, for each element a_i , nondeterministically picks a value $j < i$ and swaps a_i and a_j . When encoding the problem as query $\exists h. l_{out} = \text{shuffle}(l_{in}, h)$ and providing the DSL $\mathcal{L}$ in Equation (B.3), ASPIRE synthesized “ $\text{sort}(l_{in}) = \text{sort}(l_{out})$ ” as both $\mathcal{L}$ -consequence and $\mathcal{L}$ -implicant, which essentially states that the output should be a permutation of the input, and any permutation can be obtained as the output.

$$\begin{aligned} S &:= L = L \\ L &:= l_{in} \mid l_{out} \mid \text{sort}(l_{in}) \mid \text{sort}(l_{out}) \end{aligned} \tag{B.3}$$

Nondeterministic Sorting

The `bubble $l$` ($l \in \{3, 4\}$) benchmarks model a program that, given an array $[a_1, \dots, a_l]$, repeatedly nondeterministically swaps two neighboring elements at most n time and returns whether the final array is sorted (stored in Boolean variable ok). When encoding the problem as the query $\exists H. ok = \text{bubble}(a_1, \dots, a_l, H)$ and supplying the DSLs $\mathcal{O}$ (rooted at $B_{\mathcal{O}}$) and $\mathcal{U}$ (rooted at $B_{\mathcal{U}}$) in Equation (B.4), for `bubble3` ASPIRE computes 7 $\mathcal{U}$ -implicants, including “ $n \geq 1 \wedge a_1 \leq a_3 \wedge ok$ ”, which states that if the first and third elements are already in the right order, we can make the entire array sorted using one swap. ASPIRE also synthesizes 5 $\mathcal{O}$ -consequences. For example, the $\mathcal{O}$ -consequence “ $(n < 3 \wedge a_1 > a_2 \wedge a_2 > a_3) \Rightarrow \neg ok$ ” tells us that, if the array is descending, we cannot make it sorted using fewer than 3 swaps.

$$\begin{aligned}
 B_{\mathcal{O}} &:= G \Rightarrow D & B_{\mathcal{U}} &:= G \wedge D \\
 G &:= \top \mid AP \mid AP \wedge AP \mid \dots \mid AP \wedge AP \wedge AP \wedge AP \wedge AP \\
 AP &:= I \{ \leq \mid < \mid = \mid \neq \} I \mid n \{ < \mid \geq \} \{ 0 \mid 1 \mid \dots \mid l^2 \} \\
 I &:= a_1 \mid \dots \mid a_l \\
 D &:= ok \mid \neg ok
 \end{aligned} \tag{B.4}$$

The benchmarks `swap3` and `swap4` consider a similar problem to `bubble $l$` , but allow swaps of arbitrary elements in the array instead of only neighboring ones.

Polynomial invariants [27]

This category contains 4 benchmarks: `rsum`, `rsqsum`, `rcubsum`, and `mergesort`. The DSLs contain polynomials of a certain degree.

The `rsum` benchmark models a program that nondeterministically adds a subset of the numbers from 1 to n , i.e., $s = \sum_{k=1}^n \text{nondet}() \cdot k$, which is encoded as a query “ $\exists H. s = \text{rsum}(n, H)$ ”. When given a DSL that can describe quadratic functions over n , ASPIRE produces $n^2 + n \geq 2s \geq 0$ as both the only $\mathcal{O}$ -consequence and the only $\mathcal{U}$ -implicant. The synthesized formula tells us the summation is not greater than $(n^2 + n)/2$ and all natural numbers that are

not greater than $(n^2 + n)/2$ can be obtained as the result. Similarly, the benchmarks `rsqsum` and `rcubsum` model $\sum_{k=1}^n \text{nondet}() \cdot k^2$ and $\sum_{k=1}^n \text{nondet}() \cdot k^3$. ASPIRE synthesizes “ $2n^3 + 3n^2 + n + 2 \geq 6s \geq 0$ ” and “ $2n^4 + 2n + 4 > 4s \geq 0$ ” as the only $\mathcal{O}$ -consequence for `rsqsum` and `rcubsum`, respectively. Unlike for `rsum`, not all natural numbers in the range induced by the synthesized $\mathcal{O}$ -consequence can be obtained, so ASPIRE will not yield the same $\mathcal{U}$ -implicant. Instead, for each benchmark ASPIRE synthesizes two $\mathcal{U}$ -implicants that describe particular nondeterministic choices, e.g., $\{s = 0, s = n^2\}$ for `rsqsum`.

`mergesort` takes an array A with nondeterministic values as input and computes the number of inverse pairs of the subarray $A_{s..e}$ using merge sort. ASPIRE should synthesize an $\mathcal{O}$ -consequence like $n \leq (e - s)(e - s + 1)/2$ that explains the maximal number of inverse pairs, but due to the nested recursion/loops, ASPIRE fails to return any properties within the time limit.

Benchmarks from SV-COMP [17]

These 4 benchmarks model programs that add a constant to each input variable repeatedly. E.g., `jain2` models the following program:

```
x = 1; y = 1;
while(nondet()) { x = x + 2 * nondet(); y = y + 2 * nondet(); }
```

The remaining `jain` benchmarks differ from `jain2` in the number of variables, the initial values, and the values that are added. We designed DSLs with basic arithmetic operators including modulo, from which ASPIRE can discover properties about the final values of the variables. For `jain2`, ASPIRE synthesizes $\{y \bmod 2 = 1, (x + y) \bmod 2 = 0\}$ as $\mathcal{O}$ -consequences and $\{x \bmod 2 = 1 \wedge y \bmod 2 = 1\}$ as $\mathcal{U}$ -implicants. Note that the $\mathcal{O}$ -consequences consists of two properties and $\mathcal{U}$ -implicants only has a single one; however, these properties are equivalent. For the `jain` benchmarks, synthesis of $\mathcal{U}$ -implicants takes on average 1.38x longer than synthesis of $\mathcal{O}$ -consequences; for the former, ASPIRE has to discover the entire property at once.

B.3 Application 1: Incorrectness Reasoning

Expressible amounts in two kinds of coins

The coin benchmarks nondeterministically model the possible dollar amounts that can be represented using coins of two values.

O’Hearn [110] provides syntax-direct rules to find the weakest under-approximate postcondition and the weakest possible precondition, but this approach has to explicitly unroll loops and introduce existential quantifiers to deal with assignments and nondeterminism, thus resulting in predicates that might be hard to dispatch to a constraint solver. Using LOUD, one can instead use the DSL $\mathcal{L}$ to customize what properties they are interested in obtaining in the $\mathcal{L}$ -weakest under-approximate postcondition and $\mathcal{L}$ -weakest possible precondition.

For example, the following function `coin` takes two integers a and b that are co-prime (shown in `presumes` part) as input, then nondeterministically chooses two non-negative integers x and y , and finally returns $ax + by$. The more intuitive interpretation of the program is that it represents all amounts that can be expressed using only coins of value a and b .

```
int coin(int a, int b)
/*  presumes: [gcd(a,b)==1]
    achieve1: [gcd(a,b)==1 /\ exists x>=0, y>=0. r==a*x+b*y]
    achieve2: [gcd(a,b)==1 /\ r==a]
    achieve3: [gcd(a,b)==1 /\ r>a*b-a-b]   */
{
    int x = nondet(); assume(x >= 0);
    int y = nondet(); assume(y >= 0);
    return a * x + b * y;
}
```

The predicates `achieve1`, `achieve2`, and `achieve3` are all valid under-approximation postconditions. `achieve1` is the one obtained using the derivation rules by O’Hearn [110]: It is precise but has an existential quantifier and multiplication, which make it hard to check in later reasoning. `achieve2` could be obtained by a dynamic symbolic execution approach [23] that concretizes

$x = 1$ and $y = 0$; this postcondition is valid but less precise than `achieve1`.

The flexibility of `LOUD` allows us to modify the DSL and not be tied to any specific rule derivation technique. Consider for example a situation in which we are not interested in the actual relation captured by the program, but are just interested in identifying a lower bound (or upper bound) above (or below) which all program outcomes can be effectively produced—i.e., an under-approximation of the output range of the function. For example, one may care about after some nondeterministic perturbation of the initial state, which states within a certain distance could all be possible results.

In terms of `coin`, one can look for such a lower bound using `ASPIRE`, by supplying the following DSL that has the bounding predicate $r > N_0$ at the top level and such that nonterminal N_0 can derive a quadratic expression containing a and b :

$$\begin{aligned}
C &:= r > N_0 \\
N_0 &:= N_1 \mid N_1 + N_1 \mid N_1 - N_1 \\
N_1 &:= I \mid I + I \mid I - I \\
I &:= a \mid b \mid ab \mid a^2 \mid b^2 \mid 1 \mid 0
\end{aligned} \tag{B.5}$$

For the DSL in Equation (B.5), `ASPIRE` will synthesize the $\mathcal{L}$ -weakest under-approximate postcondition `achieve3`, i.e., “ $\text{gcd}(a, b) = 1 \wedge r > ab - a - b$ ”, which states that all numbers greater than $ab - a - b$ can be produced by `coin` assuming $\text{gcd}(a, b) = 1$. Compared to `achieve1` and `achieve2`, `achieve3` guarantees a certain degree of precision and meanwhile meets the needs to obtain a lower bound.

Conditions lead to a hash collision

The `hashcoll` benchmark models the condition of hash collision after applying a parametric hash function to a set of integers. The parametric hash function is defined as $f[a](x) = ax \bmod M$, where $a \in \{1, \dots, M - 1\}$ is the parameter we can instantiate the function with. We also have a set of integers S to which we want to apply the hash function.

To identify what kind of set S will possibly lead to a hash collision, we compute the $\mathcal{L}$ -weakest possible precondition $wpp_{\mathcal{L}}(S_o = \text{map}(f[a], S), \text{size}(S) > \text{size}(S_o))$. The following DSL $\mathcal{L}$ is intended to capture the relation between S and M in the $\mathcal{L}$ -weakest possible precondition:

$$\begin{aligned} D &:= \perp \mid AP \mid AP \wedge AP \mid AP \wedge AP \wedge AP \\ AP &:= \text{isPrime}(M) \mid \neg \text{isPrime}(M) \mid N\{\leq \mid < \mid = \mid \neq\}N \\ N &:= \text{size}(S) \mid \text{modsize}(S, M) \mid M \mid 0 \mid 1 \end{aligned} \tag{B.6}$$

The function $\text{modsize}(S, M)$ computes the size of the set obtained by taking the M -modulus for each number in S .

Using the DSL from Equation (B.6), ASPIRE synthesizes the following $\mathcal{L}$ -weakest possible precondition:

$$\text{size}(S) > \text{modsize}(S, M) \quad \text{size}(S) \geq M \wedge \neg \text{isPrime}(M) \tag{B.7}$$

The first $\mathcal{L}$ -implicant is a valid possible precondition since it implies that there are at least two integers in S that are congruent w.r.t. M , which will be hashed to the same value.

The second $\mathcal{L}$ -implicant is also a valid one as twofold: (i) if the size of S is larger than M , there exists two integers in S are congruent and thus collide. (ii) if the size of S is equal to M , and meanwhile M is not prime, there always exists a bad parameter a that is not coprime with M and therefore can cause collision.

By following the syntax-directed rules proposed by Hoare [71] to compute the weakest possible precondition, we would get the predicates “ $\exists a. \text{size}(S) = \text{size}(\text{map}(f[a], S))$ ”, which still contains a quantifier and is effectively the same as the original query we were asking—i.e., it does not help us understand the program behavior.

Furthermore, if we are interested in what set S will possibly *not* lead to a hash collision, we could set the postcondition as $\text{size}(S) = \text{size}(S_o)$, and ASPIRE will synthesize the $\mathcal{L}$ -weakest

possible precondition in same DSL from Equation (B.6) as follows:

$$\text{size}(S) = \text{modsize}(S, M) \quad (\text{B.8})$$

which means no two integers in S are congruent w.r.t M . This is a valid possible precondition since when it is satisfied, there is always *some* parameter a (e.g., 1) such that $\text{gcd}(a, M) = 1$, and thus one can prevent hash collision.

B.4 Application 2: Reasoning about Concurrent Programs

Describing Sources of Deadlock

The `philo` benchmark encodes the dining-philosophers problem where N “philosophers” are sitting around a table, and between each pair of philosophers is a single fork (and thus, N total). Each philosopher alternatively thinks and eats. To eat, a philosopher needs two forks, both the one on the left and the one on the right. When finishing eating and back to thinking, they will put down both forks. We say there is a deadlock when all philosophers want to eat but cannot get both forks because every philosopher is holding a single fork.

In this example, the so-called *circular wait* is a necessary condition for deadlock, in which there exists a circular chain of threads such that each thread holds resources that are being requested by the next thread in the chain. Circular wait can be prevented by adjusting the order in which each thread requests resources. In terms of `philo`, each philosopher could either take the left fork first or the right fork first.

We show `ASPIRE` can be used to understand what execution orders affect whether a deadlock happens. To do so, we model this problem as query $\exists h. dl = \text{schedule}(o_1, \dots, o_N, h)$, where $o_i \in \{L, R\}$ indicates which fork the philosopher i always takes first. We then supply the following two DSLs (the one rooted at nonterminal $B_{\mathcal{O}}$ is for $\mathcal{O}$ -consequences, whereas the one

rooted at $B_{\mathcal{U}}$ is for $\mathcal{U}$ -implicants):

$$\begin{aligned}
B_{\mathcal{O}} &:= G \Rightarrow D & B_{\mathcal{U}} &:= G \wedge D \\
G &:= \top \mid AP \mid AP \wedge AP \mid \cdots \mid AP \wedge AP \wedge AP \wedge AP \wedge AP \\
AP &:= O = L \mid O = R \\
O &:= o_1 \mid \cdots \mid o_N \\
R &:= dl \mid \neg dl
\end{aligned} \tag{B.9}$$

For the case involving three threads/philosophers ($N = 3$), ASPIRE synthesizes the following $\mathcal{O}$ -consequences, which informally state that deadlock can be prevented by having two of the threads disagree on their fork choice:

$$(o_0 = L \wedge o_2 = R) \Rightarrow \neg dl \quad (o_2 = L \wedge o_1 = R) \Rightarrow \neg dl \quad (o_1 = L \wedge o_0 = R) \Rightarrow \neg dl \tag{B.10}$$

For the same N , ASPIRE also synthesizes the following $\mathcal{U}$ -implicants, which exactly characterize the two cases in which a deadlock can happen (first two properties) and also capture that there exists an execution that does not lead to a deadlock (last property).

$$o_0 = L \wedge o_1 = L \wedge o_2 = L \wedge dl \quad o_0 = R \wedge o_1 = R \wedge o_2 = R \wedge dl \quad \neg dl \tag{B.11}$$

The 4 `rsrc` benchmarks focus on how the amount of resources affects the deadlock. Consider a simple resource allocator that contains M types of resources $R_1, \dots, R_M$, and initially has n_i units of resource R_i . The allocator receives T threads, each containing a list of resources the thread needs and in what order, and at each step, it needs to decide which next resource of each thread should be allocated. Once all the resources in the list are allocated, the thread completes its job and releases them altogether. However, if a request cannot be fulfilled due to the lack of resources of that type, the thread waits. We say the allocator is in a deadlock when multiple threads are waiting and no progress can be made.

We show how ASPIRE tells us what resource amounts never lead to a deadlock and what resource amounts possibly cause a deadlock. The benchmark `rsrc2` is a case where $T = 2$ and $M = 2$, and where thread-1 requests resources $[R_1, R_2, R_1, R_2]$ and thread-2 requests resources $[R_2, R_1, R_2, R_1]$. In LOUD, a completely nondeterministic allocator that could allocate any resource to any waiting thread could be modeled as the query “ $\exists h. dl = \text{schedule}(n_1, n_2, h)$ ”, where dl is a Boolean variable that indicates whether deadlock happens, and the h is an array used to model the sequence of nondeterministic choices during scheduling. Figure B.1 shows the actual SKETCH program used to define the semantics of the scheduler.

We supply ASPIRE with a similar DSL to Equation (B.9), with replace the production rule for AP by “ $AP := N\{< | \leq | =\}N$ ”, where N can derive every n_i and integer constants. ASPIRE synthesizes the $\mathcal{O}$ -consequences in Equation (B.12) that tells us about resource amounts for which a deadlock must or must not happen. For example, the third $\mathcal{O}$ -consequence states that “if there are more than 2 units of R_1 and more than 3 units of R_2 , *no scheduling order* can lead to a deadlock.”

$$n_1 \leq 1 \Rightarrow dl \quad n_2 \leq 1 \Rightarrow dl \quad (n_1 \geq 2 \wedge n_2 \geq 3) \Rightarrow \neg dl \quad (n_1 \geq 3 \wedge n_2 \geq 2) \Rightarrow \neg dl \quad (\text{B.12})$$

For the same problem, ASPIRE synthesizes the $\mathcal{U}$ -implicants in B.12 that tells us about resource amounts for which a deadlock may or may not happen. For example, the third $\mathcal{U}$ -implicant states that “if both types of resources are available in a quantity no more than 3, *there exists a scheduling order* that leads to a deadlock.”

$$n_1 \leq 2 \wedge dl \quad n_2 \leq 2 \wedge dl \quad n_1 \leq 3 \wedge n_2 \leq 3 \wedge dl \quad n_1 \geq 2 \wedge n_2 \geq 2 \wedge \neg dl \quad (\text{B.13})$$

All `rsrc` benchmarks are instances of the problem above, of which $(T, M, \text{length of the request list})$ are $(2, 2, 2)$, $(2, 2, 4)$, $(3, 2, 4)$, and $(2, 3, 8)$. Even for the hardest instance of $(2, 3, 8)$, ASPIRE synthesized the best $\mathcal{O}$ -consequences within 160 seconds (from $4.37 \cdot 10^{12}$ properties),

and the best $\mathcal{U}$ -implicants within 60 seconds (from $2.75 \cdot 10^{11}$ properties)

```
1  int[] th1, th2;
2  bool schedule(int n1, int n2, int[] h) {
3      int pc1 = 0, pc2 = 0;
4      for(int i = 0; i < len(th1) + len(thread2); i++) {
5          bool ready1 = pc1 < len(th1) && resource of type th1[pc1] available
6          bool ready2 = pc2 < len(th2) && resource of type th2[pc2] available
7          int allocto;
8          if(h[i] == 1) { // prioritize thread 1
9              if(ready1) allocto = 1;
10             else if(ready2) allocto = 2;
11             else return true;
12             // the next resource of both threads is not enough, deadlock.
13         }
14         else if(h[i] == 2) { // prioritize thread 2
15             if(ready2) allocto = 2;
16             else if(ready1) allocto = 1;
17             else return true;
18             // the next resources of both threads are not enough, deadlock.
19         }
20         allocate the resource to th_allocto;
21         pc_allocto++;
22         if (pc_allocto == len(th_allocto))
23             release the resources it occupied;
24     }
25     return false; // both threads finished, no deadlock.
26 }
```

Figure B.1. Resource allocator for two threads

Preventing Race Conditions

The 3 race benchmarks are about describing possible race conditions in concurrent programs. In each of them, there are 2 threads that access and modify a shared variable using the methods `set()` and `get()`. For example, Figure B.2 shows the code of 2 threads in benchmark `race2`.

When there is no possible context switching between the 2 threads in `race2`, if the initial value of the variable `t` is 0, its final value should be -1 (which we call the expected result). However, context switching can cause different interleaving of the threads to produce values different than -1 —i.e., there exists a data race. Such a data race is typically prevented by introducing critical sections, in which instructions must be executed atomically.

<pre>// Thread 1 0: t ← get() 1: t ← t + 1 2: t ← get() 3: t ← t + 1 4: set(t)</pre>	<pre>// Thread 2 0: t ← get() 1: t ← t - 1 2: set(t) 3: t ← t - 1 4: set(t)</pre>
--	---

Figure B.2. Two threads in race2

We show how ASPIRE can be used to identify the minimum part of the code that should be made atomic for the code to be race free. To model the problem in ASPIRE, we introduce two variables ac_1 and ac_2 , which will be used to capture what lines in each thread should be executed atomically. The predicate $atom(ac_i, l, r)$ holds true if the instructions from line l to line r of thread- i should be executed atomically.

Now we can model whether a race happens as a query $\exists h. race = schedule(ac_1, ac_2, h)$, where $race$ captures whether a race condition can happen. We provide the following two DSLs (the one rooted at nonterminal $B_{\mathcal{O}}$ is for $\mathcal{O}$ -consequences, whereas the one rooted at $B_{\mathcal{U}}$ is for $\mathcal{U}$ -implicants):

$$\begin{aligned}
B_{\mathcal{O}} &:= G \Rightarrow D & B_{\mathcal{U}} &:= G \wedge D \\
G &:= \top \mid AP \mid AP \wedge AP \mid \cdots \mid AP \wedge AP \wedge AP \wedge AP \wedge AP \\
AP &:= atom(AC, I, I) \mid \neg atom(AC, I, I) \\
I &:= \text{all line numbers} \\
AC &:= ac_1 \mid ac_2 \\
D &:= race \mid \neg race
\end{aligned} \tag{B.14}$$

The predicate $atom(ac_i, l, r)$ is implemented as a conjunction $\bigwedge_{k=l}^{r-1} noSwitch(ac_i, k)$, where the predicate $noSwitch(ac_i, k)$ is true if in thread-1, the instruction $k+1$ must be immediately executed after instruction k . This implementation makes it so that the predicate $atom(ac_i, l_1, r_1)$ implies $atom(ac_i, l_2, r_2)$ when $l_1 \leq l_2$ and $r_1 \geq r_2$. Since ASPIRE looks for a tightest properties, such an implementation lets ASPIRE reason about what is the smallest needed atomic execution.

For `race2`, ASPIRE synthesizes the following $\mathcal{O}$ -consequences, which informally states that setting line 2 to line 4 of thread-1 and line 0 to line 4 of thread-2 as critical sections can prevent data race:

$$(atom(ac_1, 2, 4) \wedge atom(ac_2, 0, 4)) \Rightarrow \neg race \quad (B.15)$$

We can observe that the effect of the first two instructions of thread-1 is overwritten by the third instruction, so it is unnecessary to include them in a critical section.

For `race2`, ASPIRE synthesizes the following $\mathcal{U}$ -implicants, which states that setting line 2 to line 4 of thread-1 and whole thread-2 as critical sections is in fact *necessary*!

$$\neg atom(ac_1, 2, 4) \wedge race \quad \neg atom(ac_2, 0, 4) \wedge race \quad \neg race \quad (B.16)$$

This example shows how over- and under-approximated reasoning can be cleverly combined in LOUD to understand necessary and sufficient interventions in preventing data races.

B.5 Application 3: Solving two-player games

Definition of safety games

A safety game consists of a game graph $G = \langle (S, E), (S_1, S_2) \rangle$ and a safety objective F . In the graph, S is a set of states partitioned into player-1 states S_1 and player-2 states S_2 , $E \subseteq S \times S$ is a set of edges in which each edge connects a state in S_1 and a state in S_2 . The safety objective $F \subseteq S$ is a set of safe states. player-1's goal is to remain in safe states, while player-2's goal is to visit unsafe states at least once.

We assume both players play the game according to a finite-state *memoryless* strategy that is independent of the action history and depends only on the current states. For safety games, there always exists a memoryless winning strategy.

More benchmarks

The nim2 game is played with 2 heaps of pebbles with number n_1 and n_2 . On each turn, a player must remove at least one pebble and may remove any number of pebbles if they all come from the same heap. The goal of the game is to be the player to remove the last pebble. One synthesized must strategy is Equation (B.17).

$$\begin{aligned} & \forall i, j. (i < j \Rightarrow \alpha[i, j].\text{heap} = 2) \wedge \forall i, j. (i < j \Rightarrow \alpha[i, j].\text{num} = j - i) \\ & \wedge \forall i, j. (i > j \Rightarrow \alpha[i, j].\text{heap} = 1) \wedge \forall i, j. (i > j \Rightarrow \alpha[i, j].\text{num} = i - j) \wedge n_1 \neq n_2 \Rightarrow w = T \end{aligned} \quad (\text{B.17})$$

where heap denotes the heap from which pebbles are taken when the numbers of pebbles in two heaps are i and j , and num denotes the number of pebbles taken. Equation (B.17) essentially states that one can win if the initial two heaps of stones are different, and always keep them the same after taking.

```
temp = 20.5; isOn = 0;
while(*) {
    assert(20 <= temp <= 25);
    isOn ++;
    if (isOn == ??) {
        temp += ??; isOn = 0;
    }
    temp -= (temp - 19) / 10;
}
```

The temp game models a controller for a thermostat shown in the right. We consider strategies that set isOn as 1 at the k -th of every n times. ASPIRE synthesizes 9 must strategies (and also 9 equivalent may strategies since there is no adversary in temp game), e.g., “ $2 \leq \alpha.n \leq 7 \wedge \alpha.k = 1 \Rightarrow w$ ”, which states that the thermostat can keep the temperature in $[20, 25]$ by increasing two degree in the first second of every $n(2 \leq n \leq 7)$ seconds.

The num benchmarks consider games played over a one-dimensional grid—i.e., an integer.

Each game is a 4-tuple (v, A_1, A_2, S) , where v is an initial integer value, $A_1 = \{f_1, f_2, \dots, f_n\}$ and $A_2 = \{g_1, g_2, \dots, g_m\}$ (such that $f_i, g_i \in \mathbb{Z} \rightarrow \mathbb{Z}$) are the actions set that player-1 and player-2 use to manipulate the integer (e.g., increments, decrements, etc.), and $S \subseteq \mathbb{Z}$ is a the set of integers player-1 wants to stay in to win the game.

The problems $\text{num1} = (v, \{nop, -1\}, \{\times 2, +1\}, [0, 4])$ and $\text{num2} = (\{nop, -1\}, \{nop, +1\}, [0, 4])$ are two instances of the game above, where the initial integer value v is left unspecified.

We want to use ASPIRE to understand the relationship between the value of v and winning strategies, and thus supply ASPIRE with the following DSL:

$$\begin{aligned}
B_{must} &:= G \Rightarrow R & B_{may} &:= G \wedge R \\
G &:= \top \mid AP \mid AP \wedge AP \mid \dots \mid AP \wedge AP \wedge AP \wedge AP \wedge AP \\
AP &:= \text{decision}(\alpha, S_1, A_1) \mid v = \{0 \mid 1 \mid 2 \mid 3 \mid 4\} \\
S_1 &:= 0 \mid 1 \mid 2 \mid 3 \mid 4 \\
A_1 &:= nop \mid -1 \\
R &:= w = \{T \mid F\}
\end{aligned} \tag{B.18}$$

Using the DSL rooted at B_{must} , for num1 ASPIRE synthesized the $\mathcal{O}$ -consequences that consists of

$$(v = 0 \wedge \alpha[0] = nop \wedge \alpha[2] = -1) \Rightarrow w = T \tag{B.19}$$

as well as other 9 $\mathcal{O}$ -consequences in 15 seconds. The property states that if the initial value is 0, player-1 can remain in the range $[0, 5]$ by performing the action nop at state 0 and the action -1 at state 2. Note that the strategy does not need to be defined at any of the other infinitely many states.

Using the DSL rooted at B_{may} , for num1 ASPIRE synthesized the $\mathcal{U}$ -implicants that consists of

$$\alpha[1] = nop \wedge \alpha[2] = nop \wedge w = F \tag{B.20}$$

as well as other 16 $\mathcal{U}$ -implicants in 10 seconds. The property states that by performing the action *nop* at states 0 and 2, there exists a player-2 strategy that makes player-1 lose the game.

Bibliography

- [1] Lakshya A Agrawal, Aditya Kanade, Navin Goyal, Shuvendu K. Lahiri, and Sriram K. Rajamani. Monitor-guided decoding of code lms with static analysis of repository context. In Alice Oh, Tristan Naumann, Amir Globerson, Kate Saenko, Moritz Hardt, and Sergey Levine, editors, *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*, 2023. URL http://papers.nips.cc/paper_files/paper/2023/hash/662b1774ba8845fc1fa3d1fc0177ceeb-Abstract-Conference.html.
- [2] Cyril Allauzen and Michael Riley. A pushdown transducer extension for the openfst library. In *Implementation and Application of Automata: 17th International Conference, CIAA 2012, Porto, Portugal, July 17-20, 2012. Proceedings 17*, pages 66–77. Springer, 2012.
- [3] Rajeev Alur, Pavol Cerný, and Arjun Radhakrishna. Synthesis through unification. In Daniel Kroening and Corina S. Pasareanu, editors, *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part II*, volume 9207 of *Lecture Notes in Computer Science*, pages 163–179. Springer, 2015. doi: 10.1007/978-3-319-21668-3_10. URL https://doi.org/10.1007/978-3-319-21668-3_10.
- [4] Rajeev Alur, Dana Fisman, Saswat Padhi, Rishabh Singh, and Abhishek Udupa. Sygus-comp 2018: Results and analysis, 2019. URL <https://arxiv.org/abs/1904.07146>.
- [5] Afra Amini, Li Du, and Ryan Cotterell. Structured voronoi sampling. *Advances in Neural Information Processing Systems*, 36, 2024.
- [6] Emmanuel Anaya Gonzalez, Sairam Vaidya, Kanghee Park, Ruyi Ji, Taylor Berg-Kirkpatrick, and Loris D’Antoni. Constrained sampling for language models should be easy: An mcmc perspective. *Advances in Neural Information Processing Systems*, 38: 59951–59981, 2026.
- [7] Peter Anderson, Basura Fernando, Mark Johnson, and Stephen Gould. Guided open vocabulary image captioning with constrained beam search. *arXiv preprint arXiv:1612.00576*, 2016.

- [8] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. Arpaci-Dusseau Books, 1.10 edition, November 2023.
- [9] 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. URL <https://doi.org/10.1145/3485481>.
- [10] Roberto Bagnara, Patricia M. Hill, and Enea Zaffanella. The parma polyhedra library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. *Sci. Comput. Program.*, 72(1-2):3–21, 2008. doi: 10.1016/j.scico.2007.08.001. URL <https://doi.org/10.1016/j.scico.2007.08.001>.
- [11] Debangshu Banerjee, Tarun Suresh, Shubham Ugare, Sasa Misailovic, and Gagandeep Singh. Crane: reasoning with constrained llm generation. In *Proceedings of the 42nd International Conference on Machine Learning, ICML’25*. JMLR.org, 2025.
- [12] 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 Dana Fisman and Grigore Rosu, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I*, volume 13243 of *Lecture Notes in Computer Science*, pages 415–442. Springer, 2022. doi: 10.1007/978-3-030-99524-9_24. URL https://doi.org/10.1007/978-3-030-99524-9_24.
- [13] Nels E. Beckman, Aditya V. Nori, Sriram K. Rajamani, Robert J. Simmons, SaiDeep Tetali, and Aditya V. Thakur. Proofs from tests. *IEEE Trans. Software Eng.*, 36(4): 495–508, 2010. doi: 10.1109/TSE.2010.49. URL <https://doi.org/10.1109/TSE.2010.49>.
- [14] Jan Bessai, Andrej Dudenhefner, Boris Döder, Moritz Martens, and Jakob Rehof. Combinatory logic synthesizer. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change - 6th International Symposium, ISoLA 2014, Imperial, Corfu, Greece, October 8-11, 2014, Proceedings, Part I*, volume 8802 of *Lecture Notes in Computer Science*, pages 26–40. Springer, 2014. doi: 10.1007/978-3-662-45234-9_3. URL https://doi.org/10.1007/978-3-662-45234-9_3.
- [15] Luca Beurer-Kellner, Marc Fischer, and Martin Vechev. Prompting is programming: A query language for large language models. *Proc. ACM Program. Lang.*, 7(PLDI), jun 2023. doi: 10.1145/3591300. URL <https://doi.org/10.1145/3591300>.

- [16] Luca Beurer-Kellner, Marc Fischer, and Martin Vechev. Guiding llms the right way: fast, non-invasive constrained generation. 2024.
- [17] D. Beyer. State of the art in software verification and witness validation: SV-COMP 2024. In *Proc. TACAS (3)*, LNCS 14572, pages 299–329. Springer, 2024. doi: 10.1007/978-3-031-57256-2_15.
- [18] Nikolaj Bjorner and Mikolas Janota. Playing with quantified satisfaction. In Ansgar Fehnker, Annabelle McIver, Geoff Sutcliffe, and Andrei Voronkov, editors, *LPAR-20. 20th International Conferences on Logic for Programming, Artificial Intelligence and Reasoning - Short Presentations*, volume 35 of *EPiC Series in Computing*, pages 15–27. EasyChair, 2015. doi: 10.29007/vv21. URL /publications/paper/jmM.
- [19] Tim Blazytko, Moritz Contag, Cornelius Aschermann, and Thorsten Holz. Syntia: Synthesizing the semantics of obfuscated code. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 643–659, Vancouver, BC, August 2017. USENIX Association. ISBN 978-1-931971-40-9. URL <https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/blazytko>.
- [20] Roderick Bloem, Krishnendu Chatterjee, and Barbara Jobstmann. *Graph Games and Reactive Synthesis*, pages 921–962. Springer International Publishing, Cham, 2018. ISBN 978-3-319-10575-8. doi: 10.1007/978-3-319-10575-8_27. URL https://doi.org/10.1007/978-3-319-10575-8_27.
- [21] Matthew Bowers, Theo X Olausson, Lionel Wong, Gabriel Grand, Joshua B Tenenbaum, Kevin Ellis, and Armando Solar-Lezama. Top-down synthesis for library learning. *Proceedings of the ACM on Programming Languages*, 7(POPL):1182–1213, 2023.
- [22] Roberto Bruni, Roberto Giacobazzi, Roberta Gori, and Francesco Ranzato. Abstract interpretation repair. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2022, page 426–441, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450392655. doi: 10.1145/3519939.3523453. URL <https://doi.org/10.1145/3519939.3523453>.
- [23] Cristian Cadar and Koushik Sen. Symbolic execution for software testing: three decades later. *Commun. ACM*, 56(2):82–90, feb 2013. ISSN 0001-0782. doi: 10.1145/2408776.2408795. URL <https://doi.org/10.1145/2408776.2408795>.
- [24] Cristiano Calcagno, Dino Distefano, Peter W. O’Hearn, and Hongseok Yang. Compositional shape analysis by means of bi-abduction. *Journal of the ACM*, 58(6):26:1–26:66, 2011. doi: 10.1145/2049697.2049700.
- [25] Diego Calvanese, Silvio Ghilardi, Alessandro Gianola, Marco Montali, and Andrey Rivkin. Combined covers and beth definability. In Nicolas Peltier and Viorica Sofronie-

- Stokkermans, editors, *Automated Reasoning*, pages 181–200, Cham, 2020. Springer International Publishing. ISBN 978-3-030-51074-9.
- [26] David Cao, Rose Kunkel, Chandrakana Nandi, Max Willsey, Zachary Tatlock, and Nadia Polikarpova. babble: Learning better abstractions with e-graphs and anti-unification. *Proceedings of the ACM on Programming Languages*, 7(POPL):396–424, 2023.
 - [27] Krishnendu Chatterjee, Hongfei Fu, Amir Kafshdar Goharshady, and Ehsan Kafshdar Goharshady. Polynomial invariant generation for non-deterministic recursive programs. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, page 672–687, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450376136. doi: 10.1145/3385412.3385969. URL <https://doi.org/10.1145/3385412.3385969>.
 - [28] Hubert Comon, Max Dauchet, Rémi Gilleron, Florent Jacquemard, Denis Lugiez, Christof Löding, Sophie Tison, and Marc Tommasi. *Tree Automata Techniques and Applications*. 2008. URL <https://hal.inria.fr/hal-03367725>.
 - [29] Thierry Coquand and Gérard Huet. *The calculus of constructions*. PhD thesis, INRIA, 1986.
 - [30] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In Robert M. Graham, Michael A. Harrison, and Ravi Sethi, editors, *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*, pages 238–252. ACM, 1977. doi: 10.1145/512950.512973. URL <https://doi.org/10.1145/512950.512973>.
 - [31] Patrick Cousot and Radhia Cousot. Static determination of dynamic properties of recursive procedures. In *Formal Descriptions of Programming Concepts*. North-Holland, 1978.
 - [32] Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In Alfred V. Aho, Stephen N. Zilles, and Thomas G. Szymanski, editors, *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages, Tucson, Arizona, USA, January 1978*, pages 84–96. ACM Press, 1978. doi: 10.1145/512760.512770. URL <https://doi.org/10.1145/512760.512770>.
 - [33] Loris D’Antoni, Marco Gaboardi, Emilio Jesús Gallego Arias, Andreas Haeberlen, and Benjamin C. Pierce. Sensitivity analysis using type-based constraints. In Richard Lazarus, Assaf J. Kfoury, and Jacob Beal, editors, *Proceedings of the 1st annual workshop on Functional programming concepts in domain-specific languages, FPCDSL@ICFP 2013, Boston, Massachusetts, USA, September 22, 2013*, pages 43–50. ACM, 2013. doi: 10.1145/2505351.2505353. URL <https://doi.org/10.1145/2505351.2505353>.

- [34] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.
- [35] Edsko de Vries and Vasileios Koutavas. Reverse hoare logic. In Gilles Barthe, Alberto Pardo, and Gerardo Schneider, editors, *Software Engineering and Formal Methods*, pages 155–171, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg. ISBN 978-3-642-24690-6.
- [36] Daniel Deutsch, Shyam Upadhyay, and Dan Roth. A general-purpose algorithm for constrained sequential inference. In *Proceedings of the 23rd Conference on Computational Natural Language Learning (CoNLL)*, pages 482–492, 2019.
- [37] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8):453–457, aug 1975. ISSN 0001-0782. doi: 10.1145/360933.360975. URL <https://doi.org/10.1145/360933.360975>.
- [38] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice-Hall series in automatic computation. Prentice-Hall, 1976. ISBN 9780132158718. URL <https://books.google.com/books?id=MsUmAAAAMAAJ>.
- [39] Edsger W Dijkstra and Carel S Scholten. *Predicate calculus and program semantics*. Springer-Verlag, 1990.
- [40] Isil Dillig, Thomas Dillig, and Alex Aiken. Precise reasoning for programs using containers. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL ’11, page 187–200, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450304900. doi: 10.1145/1926385.1926407. URL <https://doi.org/10.1145/1926385.1926407>.
- [41] Isil Dillig, Thomas Dillig, and Alex Aiken. Automated error diagnosis using abductive inference. *ACM SIGPLAN Notices*, 47(6):181–192, 2012.
- [42] Isil Dillig, Thomas Dillig, Boyang Li, and Ken McMillan. Inductive invariant generation via abductive inference. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA ’13, page 443–456, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450323741. doi: 10.1145/2509136.2509511. URL <https://doi.org/10.1145/2509136.2509511>.
- [43] Yihong Dong, Xue Jiang, Yuchen Liu, Ge Li, and Zhi Jin. Codepad: Sequence-based code generation with pushdown automaton. *arXiv preprint arXiv:2211.00818*, 2022.
- [44] Yixin Dong, Charlie F Ruan, Yaxing Cai, Ruihang Lai, Ziyi Xu, Yilong Zhao, and Tianqi Chen. Xgrammar: Flexible and efficient structured generation engine for large language

models. *arXiv preprint arXiv:2411.15100*, 2024.

- [45] Kevin Ellis, Catherine Wong, Maxwell Nye, Mathias Sablé-Meyer, Lucas Morales, Luke Hewitt, Luc Cary, Armando Solar-Lezama, and Joshua B Tenenbaum. Dreamcoder: Bootstrapping inductive program synthesis with wake-sleep library learning. In *Proceedings of the 42nd acm sigplan international conference on programming language design and implementation*, pages 835–850, 2021.
- [46] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Trans. Software Eng.*, 27(2):99–123, 2001. doi: 10.1109/32.908957. URL <https://doi.org/10.1109/32.908957>.
- [47] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. The daikon system for dynamic detection of likely invariants. *Sci. Comput. Program.*, 69(1-3):35–45, 2007. doi: 10.1016/j.scico.2007.01.015. URL <https://doi.org/10.1016/j.scico.2007.01.015>.
- [48] Azadeh Farzan and Zachary Kincaid. Strategy synthesis for linear arithmetic games. *Proc. ACM Program. Lang.*, 2(POPL), dec 2017. doi: 10.1145/3158149. URL <https://doi.org/10.1145/3158149>.
- [49] Yu Feng, Ruben Martins, Yuepeng Wang, Isil Dillig, and Thomas W. Reps. Component-based synthesis for complex APIs. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 599–612, 2017. doi: 10.1145/3009837.3009851. URL <https://doi.org/10.1145/3009837.3009851>.
- [50] Cormac Flanagan and K. Rustan M. Leino. Houdini, an annotation assistant for ESC/Java. In José Nuno Oliveira and Pamela Zave, editors, *FME 2001: Formal Methods for Increasing Software Productivity, International Symposium of Formal Methods Europe, Berlin, Germany, March 12-16, 2001, Proceedings*, volume 2021 of *Lecture Notes in Computer Science*, pages 500–517. Springer, 2001. doi: 10.1007/3-540-45251-6_29. URL https://doi.org/10.1007/3-540-45251-6_29.
- [51] Robert W. Floyd. Algorithm 97: Shortest path. *Commun. ACM*, 5(6):345, June 1962. ISSN 0001-0782. doi: 10.1145/367766.368168. URL <https://doi.org/10.1145/367766.368168>.
- [52] Pranav Garg, Christof Löding, P. Madhusudan, and Daniel Neider. ICE: A robust framework for learning invariants. In Armin Biere and Roderick Bloem, editors, *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, volume 8559 of *Lecture Notes in Computer Science*, pages 69–87. Springer, 2014. doi: 10.1007/978-3-319-08867-9_5. URL https://doi.org/10.1007/978-3-319-08867-9_5.

- [53] Thibault Gauthier, Cezary Kaliszyk, Josef Urban, Ramana Kumar, and Michael Norrish. TacticToe: Learning to prove with tactics. *Journal of Automated Reasoning*, 65(2): 257–286, 2021. doi: 10.1007/s10817-020-09580-x.
- [54] Saibo Geng, Martin Josifoski, Maxime Peyrard, and Robert West. Transformers-CFG. <https://github.com/epfl-dlab/transformers-CFG>, 2023.
- [55] Saibo Geng, Martin Josifoski, Maxime Peyrard, and Robert West. Grammar-constrained decoding for structured NLP tasks without finetuning. In Houda Bouamor, Juan Pino, and Kalika Bali, editors, *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, Singapore, December 2023. Association for Computational Linguistics. URL <https://aclanthology.org/2023.emnlp-main.674>.
- [56] Saibo Geng, Hudson Cooper, Michał Moskal, Samuel Jenkins, Julian Berman, Nathan Ranchin, Robert West, Eric Horvitz, and Harsha Nori. Jonschemabench: A rigorous benchmark of structured outputs for language models, 2025. URL <https://arxiv.org/abs/2501.10868>.
- [57] Georgi Gerganov. llama.cpp, 2024. URL <https://github.com/ggerganov/llama.cpp>.
- [58] Patrice Godefroid. Compositional dynamic test generation. *SIGPLAN Not.*, 42(1):47–54, jan 2007. ISSN 0362-1340. doi: 10.1145/1190215.1190226. URL <https://doi.org/10.1145/1190215.1190226>.
- [59] J.A. Goguen, J.W. Thatcher, E.G. Wagner, and J.G. Wright. Abstract data-types as initial algebras and correctness of data representations. In *Proceedings Conference on Computer Graphics, Pattern Recognition and Data Structure*, May 1975.
- [60] Warren D. Goldfarb. The undecidability of the second-order unification problem. *Theoretical Computer Science*, 13(2):225–230, 1981. ISSN 0304-3975. doi: [https://doi.org/10.1016/0304-3975\(81\)90040-2](https://doi.org/10.1016/0304-3975(81)90040-2). URL <https://www.sciencedirect.com/science/article/pii/0304397581900402>.
- [61] Denis Gopan and Thomas W. Reps. Low-level library analysis and summarization. In Werner Damm and Holger Hermanns, editors, *Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings*, volume 4590 of *Lecture Notes in Computer Science*, pages 68–81. Springer, 2007. doi: 10.1007/978-3-540-73368-3_10. URL https://doi.org/10.1007/978-3-540-73368-3_10.
- [62] Orna Grumberg, Flavio Lerda, Ofer Strichman, and Michael Theobald. Proof-guided underapproximation-widening for multi-process systems. *SIGPLAN Not.*, 40(1):122–131, jan 2005. ISSN 0362-1340. doi: 10.1145/1047659.1040316. URL <https://doi.org/10.1145/1047659.1040316>.

- [63] Sumit Gulwani and Madan Musuvathi. Cover algorithms and their combination. In Sophia Drossopoulou, editor, *Programming Languages and Systems*, pages 193–207, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. ISBN 978-3-540-78739-6.
- [64] Zheng Guo, David Cao, Davin Tjong, Jean Yang, Cole Schlesinger, and Nadia Polikarpova. Type-directed program synthesis for restful apis. In Ranjit Jhala and Isil Dillig, editors, *PLDI '22: 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, San Diego, CA, USA, June 13 - 17, 2022*, pages 122–136. ACM, 2022. doi: 10.1145/3519939.3523450. URL <https://doi.org/10.1145/3519939.3523450>.
- [65] John V. Guttag. *The Specification and Application to Programming of Abstract Data Types*. PhD thesis, Computer Systems Research Group, Univ. of Toronto, Toronto, Canada, September 1975. URL <https://archive.org/details/technicalreportc59univ/mode/2up>.
- [66] John V. Guttag and James J. Horning. The algebraic specification of abstract data types. *Acta Informatica*, 10:27–52, 1978. doi: 10.1007/BF00260922. URL <https://doi.org/10.1007/BF00260922>.
- [67] Kodai Hashimoto and Hiroshi Unno. Refinement type inference via horn constraint optimization. In *SAS*, 2015.
- [68] Johannes Henkel, Christoph Reichenbach, and Amer Diwan. Discovering documentation for Java container classes. *IEEE Trans. Software Eng.*, 33(8):526–543, 2007. doi: 10.1109/TSE.2007.70705. URL <https://doi.org/10.1109/TSE.2007.70705>.
- [69] Roger Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the american mathematical society*, 146:29–60, 1969.
- [70] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, oct 1969. ISSN 0001-0782. doi: 10.1145/363235.363259. URL <https://doi.org/10.1145/363235.363259>.
- [71] C. A. R. Hoare. Some properties of predicate transformers. *J. ACM*, 25(3):461–480, jul 1978. ISSN 0004-5411. doi: 10.1145/322077.322088. URL <https://doi.org/10.1145/322077.322088>.
- [72] Chris Hokamp and Qun Liu. Lexically constrained decoding for sequence generation using grid beam search. *arXiv preprint arXiv:1704.07138*, 2017.
- [73] Robert Hood and Robert Melville. Real-time queue operation in pure LISP. *Inf. Process. Lett.*, 13(2):50–54, 1981. doi: 10.1016/0020-0190(81)90030-2. URL [https://doi.org/10.1016/0020-0190\(81\)90030-2](https://doi.org/10.1016/0020-0190(81)90030-2).

- [74] William A Howard. The formulae-as-types notion of construction. 1995.
- [75] J Edward Hu, Huda Khayrallah, Ryan Culkin, Patrick Xia, Tongfei Chen, Matt Post, and Benjamin Van Durme. Improved lexically constrained decoding for translation and monolingual rewriting. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 839–850, 2019.
- [76] Qinheping Hu, John Cyphert, Loris D’Antoni, and Thomas W. Reps. Exact and approximate methods for proving unrealizability of syntax-guided synthesis problems. In Alastair F. Donaldson and Emina Torlak, editors, *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, pages 1128–1142. ACM, 2020. doi: 10.1145/3385412.3385979. URL <https://doi.org/10.1145/3385412.3385979>.
- [77] Linghan Huang, Peizhou Zhao, Huaming Chen, and Lei Ma. Large language models based fuzzing techniques: A survey, 2024.
- [78] Gerard P. Huet. A unification algorithm for typed λ -calculus. *Theoretical Computer Science*, 1(1):27–57, 1975.
- [79] Shachar Itzhaky. *Automatic Reasoning for Pointer Programs Using Decidable Logics*. PhD thesis, Tel Aviv University, 2014.
- [80] Albert Q. Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, L  lio Renard Lavaud, Marie-Anne Lachaux, Pierre Stock, Teven Le Scao, Thibaut Lavril, Thomas Wang, Timoth  e Lacroix, and William El Sayed. Mistral 7b, 2023. URL <https://arxiv.org/abs/2310.06825>.
- [81] Pankaj Kumar Kalita, Sujit Kumar Muduli, Loris D’Antoni, Thomas W. Reps, and Subhajit Roy. Synthesizing abstract transformers. *Proc. ACM Program. Lang.*, (OOPSLA), 2022. doi: 10.1145/3563334. URL <https://doi.org/10.1145/3563334>.
- [82] Daniela Kaufmann, Paul Beame, Armin Biere, and Jakob Nordstrom. Adding dual variables to algebraic reasoning for gate-level multiplier verification. In *Proceedings of the 2022 Conference on Design, Automation Test in Europe, DATE ’22*, page 1431–1436, Leuven, BEL, 2022. European Design and Automation Association. ISBN 9783981926361.
- [83] Jinwoo Kim. Messy-release. <https://github.com/kjw227/Messy-Release>, 2022.
- [84] Jinwoo Kim, Qinheping Hu, Loris D’Antoni, and Thomas W. Reps. Semantics-guided synthesis. *Proc. ACM Program. Lang.*, 5(POPL):1–32, 2021. doi: 10.1145/3434311. URL <https://doi.org/10.1145/3434311>.

- [85] Donald E Knuth. On the translation of languages from left to right. *Information and control*, 8(6):607–639, 1965.
- [86] Anvesh Komuravelli, Arie Gurfinkel, and Sagar Chaki. Smt-based model checking for recursive programs. In Armin Biere and Roderick Bloem, editors, *Computer Aided Verification*, pages 17–34, Cham, 2014. Springer International Publishing. ISBN 978-3-319-08867-9.
- [87] Terry Koo, Frederick Liu, and Luheng He. Automata-based constraints for language model decoding. 2024. URL <https://openreview.net/forum?id=BDBdblmzyY>.
- [88] Sachin Kumar, Biswajit Paria, and Yulia Tsvetkov. Gradient-based constrained sampling from language models. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pages 2251–2277, 2022.
- [89] Sandeep Kumar. Specification mining in concurrent and distributed systems. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE ’11, page 1086–1089, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450304450. doi: 10.1145/1985793.1986002. URL <https://doi.org/10.1145/1985793.1986002>.
- [90] Andrea Lattuada, Travis Hance, Chanhee Cho, Matthias Brun, Isitha Subasinghe, Yi Zhou, Jon Howell, Bryan Parno, and Chris Hawblitzel. Verus: Verifying rust programs using linear ghost types. *Proc. ACM Program. Lang.*, 7(OOPSLA1), April 2023. doi: 10.1145/3586037. URL <https://doi.org/10.1145/3586037>.
- [91] Mark Law, Alessandra Russo, and Krysia Broda. *Logic-Based Learning of Answer Set Programs*, pages 196–231. Springer International Publishing, Cham, 2019. ISBN 978-3-030-31423-1. doi: 10.1007/978-3-030-31423-1_6. URL https://doi.org/10.1007/978-3-030-31423-1_6.
- [92] Ivan Yee Lee, Loris D’Antoni, and Taylor Berg-Kirkpatrick. The format tax. *arXiv preprint arXiv:2604.03616*, 2026.
- [93] K. Rustan M. Leino and Valentin Wüstholtz. The dafny integrated development environment. In Catherine Dubois, Dimitra Giannakopoulou, and Dominique Méry, editors, *Proceedings 1st Workshop on Formal Integrated Development Environment, F-IDE 2014, Grenoble, France, April 6, 2014*, volume 149 of *EPTCS*, pages 3–15, 2014. doi: 10.4204/EPTCS.149.2. URL <https://doi.org/10.4204/EPTCS.149.2>.
- [94] Xiang Li, John Thickstun, Ishaan Gulrajani, Percy S Liang, and Tatsunori B Hashimoto. Diffusion-lm improves controllable text generation. *Advances in Neural Information Processing Systems*, 35:4328–4343, 2022.

- [95] Yixuan Li, Julian Parsert, and Elizabeth Polgreen. Guiding enumerative program synthesis with large language models. In *International Conference on Computer Aided Verification*, pages 280–301. Springer, 2024.
- [96] Barbara H. Liskov and Stephen N. Zilles. Specification techniques for data abstractions. *IEEE Trans. Software Eng.*, 1(1):7–19, 1975. doi: 10.1109/TSE.1975.6312816. URL <https://doi.org/10.1109/TSE.1975.6312816>.
- [97] David Lo, Siau-Cheng Khoo, Jiawei Han, and Chao Liu, editors. *Mining Software Specifications: Methodologies and Applications*. Chapman & Hall, 2017.
- [98] Ximing Lu, Peter West, Rowan Zellers, Ronan Le Bras, Chandra Bhagavatula, and Yejin Choi. NeuroLogic decoding: (un)supervised neural text generation with predicate logic constraints. In Kristina Toutanova, Anna Rumshisky, Luke Zettlemoyer, Dilek Hakkani-Tur, Iz Beltagy, Steven Bethard, Ryan Cotterell, Tanmoy Chakraborty, and Yichao Zhou, editors, *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 4288–4299, Online, June 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.naacl-main.339. URL <https://aclanthology.org/2021.naacl-main.339>.
- [99] Ximing Lu, Sean Welleck, Peter West, Liwei Jiang, Jungo Kasai, Daniel Khashabi, Ronan Le Bras, Lianhui Qin, Youngjae Yu, Rowan Zellers, Noah A. Smith, and Yejin Choi. NeuroLogic a*esque decoding: Constrained text generation with lookahead heuristics. In Marine Carpuat, Marie-Catherine de Marneffe, and Ivan Vladimir Meza Ruiz, editors, *Proceedings of the 2022 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 780–799, Seattle, United States, July 2022. Association for Computational Linguistics. doi: 10.18653/v1/2022.naacl-main.57. URL <https://aclanthology.org/2022.naacl-main.57>.
- [100] Scott Lundberg, Marco Tulio Correia Ribeiro, Richard Edgar, Harsha-Nori, Hudson Cooper, Paul Koch, Nicholas King, Markus Marawan, Michał Moskal, S Jenkins, cpdc, h-k nyosu, Mihai Chirculescu, David Viggiano, Holden X, João Rafael, Adam Gordon Bell, Shawn Zivontsis, sam rodriguez, Riya Amemiya, SimFG, Ryan Peach, Nick Heiner, JC1DA, Flavius Popan, Andrew Minh Nguyen, AnilTuraga, Ryan Delaney, and Saibocreator. *guidance-ai/guidance*. URL <https://github.com/guidance-ai/guidance>.
- [101] Benjamin Mariano, Josh Reese, Siyuan Xu, ThanhVu Nguyen, Xiaokang Qiu, Jeffrey S. Foster, and Armando Solar-Lezama. Program synthesis with algebraic library specifications. *Proc. ACM Program. Lang.*, 3(OOPSLA), October 2019. doi: 10.1145/3360558. URL <https://doi.org/10.1145/3360558>.
- [102] Robert McNaughton and Hisao Yamada. Regular expressions and state graphs for automata. *IRE transactions on Electronic Computers*, (1):39–47, 1960.

- [103] Daniel Melcer, Nathan Fulton, Sanjay Krishna Gouda, and Haifeng Qian. Constrained decoding for code language models via efficient left and right quotienting of context-sensitive grammars. *arXiv preprint arXiv:2402.17988*, 2024.
- [104] Dale Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *Journal of logic and computation*, 1(4):497–536, 1991.
- [105] Robin Milner. A theory of type polymorphism in programming. *Journal of computer and system sciences*, 17(3):348–375, 1978.
- [106] Anders Miltner, Saswat Padhi, Todd Millstein, and David Walker. Data-driven inference of representation invariants. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, page 1–15, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450376136. doi: 10.1145/3385412.3385967. URL <https://doi.org/10.1145/3385412.3385967>.
- [107] Tom M. Mitchell. *Machine Learning*. McGraw-Hill, 1997.
- [108] Michał Moskal, Hudson Cooper, Aaron Pham, Devise Lucato, Steph Wolski, and Ying Xiong. *guidance-ai/llguidance*. URL <https://github.com/guidance-ai/llguidance>.
- [109] Leonardo de Moura and Sebastian Ullrich. The lean 4 theorem prover and programming language. In *Automated Deduction – CADE 28: 28th International Conference on Automated Deduction, Virtual Event, July 12–15, 2021, Proceedings*, page 625–635, Berlin, Heidelberg, 2021. Springer-Verlag. ISBN 978-3-030-79875-8. doi: 10.1007/978-3-030-79876-5_37. URL https://doi.org/10.1007/978-3-030-79876-5_37.
- [110] Peter W. O’Hearn. Incorrectness logic. *Proc. ACM Program. Lang.*, 4(POPL), dec 2019. doi: 10.1145/3371078. URL <https://doi.org/10.1145/3371078>.
- [111] Or Ozeri, Oded Padon, Noam Rinetzk, and Mooly Sagiv. Conjunctive abstract interpretation using paramodulation. In Ahmed Bouajjani and David Monniaux, editors, *Verification, Model Checking, and Abstract Interpretation - 18th International Conference, VMCAI 2017, Paris, France, January 15-17, 2017, Proceedings*, volume 10145 of *Lecture Notes in Computer Science*, pages 442–461. Springer, 2017. doi: 10.1007/978-3-319-52234-0_24. URL https://doi.org/10.1007/978-3-319-52234-0_24.
- [112] Saswat Padhi, Rahul Sharma, and Todd D. Millstein. Data-driven precondition inference with learned features. In Chandra Krintz and Emery D. Berger, editors, *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, pages 42–56. ACM, 2016. doi: 10.1145/2908080.2908099. URL <https://doi.org/10.1145/2908080.2908099>.
- [113] Oded Padon. *Deductive Verification of Distributed Protocols in First-Order Logic*. PhD

thesis, Tel Aviv University, 2018.

- [114] Oded Padon, James R. Wilcox, Jason R. Koenig, Kenneth L. McMillan, and Alex Aiken. Induction duality: primal-dual search for invariants. *Proc. ACM Program. Lang.*, 6(POPL): 1–29, 2022. doi: 10.1145/3498712. URL <https://doi.org/10.1145/3498712>.
- [115] Kanghee Park, Loris D’Antoni, and Thomas Reps. Synthesizing specifications. *Proc. ACM Program. Lang.*, 7(OOPSLA2), oct 2023. doi: 10.1145/3622861. URL <https://doi.org/10.1145/3622861>.
- [116] Kanghee Park, Loris D’Antoni, and Thomas Reps. Synthesizing specifications. 7 (OOPSLA2), oct 2023. doi: 10.1145/3622861. URL <https://doi.org/10.1145/3622861>.
- [117] Kanghee Park, Jiayu Wang, Taylor Berg-Kirkpatrick, Nadia Polikarpova, and Loris D’Antoni. Grammar-aligned decoding. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024. URL <https://openreview.net/forum?id=5G7ve8E1Lu>.
- [118] David L. Parnas. On the criteria to be used in decomposing systems into modules. *Comm. ACM*, 15(12):1053–1058, 1972.
- [119] Paweł Parys, Sairam Vaidya, Taylor Berg-Kirkpatrick, and Loris D’Antoni. Constrained adaptive rejection sampling. In *Forty-third International Conference on Machine Learning*, 2025.
- [120] Frank Pfenning. Unification and anti-unification in the calculus of constructions. In *LICS*, volume 91, pages 74–85, 1991.
- [121] Gabriel Poesia and Noah D Goodman. Peano: learning formal mathematical reasoning. *Philosophical Transactions of the Royal Society A*, 381(2251):20220044, 2023.
- [122] Gabriel Poesia, Alex Polozov, Vu Le, Ashish Tiwari, Gustavo Soares, Christopher Meek, and Sumit Gulwani. Synchromesh: Reliable code generation from pre-trained language models. In *International Conference on Learning Representations*, 2022. URL <https://openreview.net/forum?id=KmtVD97J43e>.
- [123] Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. Program synthesis from polymorphic refinement types. *ACM SIGPLAN Notices*, 51(6):522–538, 2016.
- [124] Matt Post and David Vilar. Fast lexically constrained decoding with dynamic beam allocation for neural machine translation. *arXiv preprint arXiv:1804.06609*, 2018.
- [125] Lianhui Qin, Sean Welleck, Daniel Khashabi, and Yejin Choi. Cold decoding: Energy-based constrained text generation with langevin dynamics. In *Advances in Neural Infor-*

mation Processing Systems, 2022.

- [126] Mohammad Raza, Sumit Gulwani, and Natasa Milic-Frayling. Compositional program synthesis from natural language and examples. In *Proceedings of the 24th International Conference on Artificial Intelligence, IJCAI'15*, page 792–800. AAAI Press, 2015. ISBN 9781577357384.
- [127] Alex Renda, Aspen K. Hopkins, and Michael Carbin. Can LLMs generate random numbers? evaluating LLM sampling in controlled domains. In *ICML 2023 Workshop: Sampling and Optimization in Discrete Space*, 2023. URL <http://people.csail.mit.edu/renda/llm-sampling-paper>.
- [128] Thomas W. Reps and Aditya V. Thakur. Automating abstract interpretation. In Barbara Jobstmann and K. Rustan M. Leino, editors, *Verification, Model Checking, and Abstract Interpretation - 17th International Conference, VMCAI 2016, St. Petersburg, FL, USA, January 17-19, 2016. Proceedings*, volume 9583 of *Lecture Notes in Computer Science*, pages 3–40. Springer, 2016. doi: 10.1007/978-3-662-49122-5_1. URL https://doi.org/10.1007/978-3-662-49122-5_1.
- [129] Thomas W. Reps, Shmuel Sagiv, and Greta Yorsh. Symbolic implementation of the best transformer. In *Verification, Model Checking, and Abstract Interpretation, 5th International Conference, VMCAI 2004, Venice, Italy, January 11-13, 2004, Proceedings*, pages 252–266, 2004. doi: 10.1007/978-3-540-24622-0_21. URL https://doi.org/10.1007/978-3-540-24622-0_21.
- [130] Andrew Reynolds, Morgan Deters, Viktor Kuncak, Cesare Tinelli, and Clark Barrett. Counterexample-guided quantifier instantiation for synthesis in smt. In Daniel Kroening and Corina S. Păsăreanu, editors, *Computer Aided Verification*, pages 198–216, Cham, 2015. Springer International Publishing. ISBN 978-3-319-21668-3.
- [131] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings*, pages 55–74. IEEE Computer Society, 2002. doi: 10.1109/LICS.2002.1029817. URL <https://doi.org/10.1109/LICS.2002.1029817>.
- [132] J Alan Robinson. Computational logic: The unification computation. *Machine intelligence*, 6:63–72, 1971.
- [133] Patrick Maxim Rondon, Ming Kawaguchi, and Ranjit Jhala. Liquid types. In Rajiv Gupta and Saman P. Amarasinghe, editors, *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*, pages 159–169. ACM, 2008. doi: 10.1145/1375581.1375602. URL <https://doi.org/10.1145/1375581.1375602>.

- [134] David A. Schmidt. A calculus of logical relations for over- and underapproximating static analyses. *Sci. Comput. Program.*, 64(1):29–53, jan 2007. ISSN 0167-6423. doi: 10.1016/j.scico.2006.03.008. URL <https://doi.org/10.1016/j.scico.2006.03.008>.
- [135] Torsten Scholak, Nathan Schucher, and Dzmitry Bahdanau. PICARD: Parsing incrementally for constrained auto-regressive decoding from language models. In Marie-Francine Moens, Xuanjing Huang, Lucia Specia, and Scott Wen-tau Yih, editors, *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 9895–9901, Online and Punta Cana, Dominican Republic, November 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.emnlp-main.779. URL <https://aclanthology.org/2021.emnlp-main.779>.
- [136] Micha Sharir and Amir Pnueli. Two approaches to interprocedural data flow analysis. In *Program Flow Analysis: Theory and Applications*. Prentice-Hall, 1981.
- [137] Kensen Shi, Jacob Steinhardt, and Percy Liang. FrAngel: Component-based synthesis with control structures. *Proc. ACM Program. Lang.*, 3(POPL):73:1–73:29, 2019. doi: 10.1145/3290386. URL <https://doi.org/10.1145/3290386>.
- [138] Richard Shin, Christopher H Lin, Sam Thomson, Charles Chen, Subhro Roy, Emmanouil Antonios Platanios, Adam Pauls, Dan Klein, Jason Eisner, and Benjamin Van Durme. Constrained language models yield few-shot semantic parsers. *arXiv preprint arXiv:2104.08768*, 2021.
- [139] Erez Shinan, MegaIng, chanicpanic, Julien Malard-Adam, Sasank Chilamkurthy, Patrick Lannigan, Kaspar Emanuel, konstantin, evandrocoan, Yuan Chang, Henry Schreiner, ThatXliner, Nicholas Chammas, ornariece, Rogdham, Michael Makukha, Ken Lauer, Michael K., Robert Rose, decorator factory, orcharddweller, Dmitry Gritsenko, Jan Rydzewski, starwarswii, Jonah Y-M, Raekye, Peter Wienemann, PJCampi, Idbo, and Kevin Latimer. *lark-parser/lark*. URL <https://github.com/lark-parser/lark>.
- [140] Paul Simon. One man’s ceiling is another man’s floor, May 1973. T-700.050.850-1 BMI, ISWC, JASRAC.
- [141] Rohit Singh, Rishabh Singh, Zhilei Xu, Rebecca Krosnick, and Armando Solar-Lezama. Modular synthesis of sketches using models. In Kenneth L. McMillan and Xavier Rival, editors, *Verification, Model Checking, and Abstract Interpretation - 15th International Conference, VMCAI 2014, San Diego, CA, USA, January 19-21, 2014, Proceedings*, volume 8318 of *Lecture Notes in Computer Science*, pages 395–414. Springer, 2014. doi: 10.1007/978-3-642-54013-4_22. URL https://doi.org/10.1007/978-3-642-54013-4_22.
- [142] John L. Singleton, Gary T. Leavens, Hriday Rajan, and David R. Cok. Inferring concise specifications of apis. *CoRR*, abs/1905.06847, 2019. URL <http://arxiv.org/abs/1905.06847>.

- [143] Nicholas Smallbone, Moa Johansson, Koen Claessen, and Maximilian Alghed. Quick specifications for the busy programmer. *Journal of Functional Programming*, 27:e18, 2017.
- [144] Armando Solar-Lezama. *Program synthesis by sketching*. PhD thesis, USA, 2008. AAI3353225.
- [145] Armando Solar-Lezama. Program sketching. *Int. J. Softw. Tools Technol. Transf.*, 15 (5-6):475–495, 2013. doi: 10.1007/s10009-012-0249-7. URL <https://doi.org/10.1007/s10009-012-0249-7>.
- [146] Jay M. Spitz and Ben Wegbreit. The verification and synthesis of data structures. *Acta Informatica*, 4:127–144, 1974. doi: 10.1007/BF00288745. URL <https://doi.org/10.1007/BF00288745>.
- [147] Elias Stengel-Eskin, Kyle Rawlins, and Benjamin Van Durme. Zero and few-shot semantic parsing with ambiguous inputs. *arXiv preprint arXiv:2306.00824*, 2023.
- [148] Zhi Rui Tam, Cheng-Kuang Wu, Yi-Lin Tsai, Chieh-Yen Lin, Hung-yi Lee, and Yun-Nung Chen. Let me speak freely? a study on the impact of format restrictions on large language model performance. In Franck Dernoncourt, Daniel Preotiuc-Pietro, and Anastasia Shimorina, editors, *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing: Industry Track*, pages 1218–1236, Miami, Florida, US, November 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.emnlp-industry.91. URL <https://aclanthology.org/2024.emnlp-industry.91/>.
- [149] The Rocq Development Team. The rocq prover, September 2025. URL <https://doi.org/10.5281/zenodo.17473943>.
- [150] Aditya V. Thakur and Thomas W. Reps. A method for symbolic computation of abstract operations. In *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings*, pages 174–192, 2012. doi: 10.1007/978-3-642-31424-7_17. URL https://doi.org/10.1007/978-3-642-31424-7_17.
- [151] Aditya V. Thakur, Matt Elder, and Thomas W. Reps. Bilateral algorithms for symbolic abstraction. In *Static Analysis - 19th International Symposium, SAS 2012, Deauville, France, September 11-13, 2012. Proceedings*, pages 111–128, 2012. doi: 10.1007/978-3-642-33125-1_10. URL https://doi.org/10.1007/978-3-642-33125-1_10.
- [152] Shubham Ugare, Tarun Suresh, Hangoo Kang, Sasa Misailovic, and Gagandeep Singh. Syncode: Llm generation with grammar augmentation, 2024. URL <https://arxiv.org/abs/2403.01632>.
- [153] Niki Vazou, Eric L Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton-Jones.

- Refinement types for haskell. In *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming*, pages 269–282, 2014.
- [154] Bailin Wang, Zi Wang, Xuezhi Wang, Yuan Cao, Rif A Saurous, and Yoon Kim. Grammar prompting for domain-specific language generation with large language models. *Advances in Neural Information Processing Systems*, 36, 2024.
 - [155] Rong Wang, Zuohua Ding, Ning Gui, and Yang Liu. Detecting bugs of concurrent programs with program invariants. *IEEE Transactions on Reliability*, 66(2):425–439, 2017. doi: 10.1109/TR.2017.2681107.
 - [156] Yu-Xiang Wang, Jing Lei, and Stephen E. Fienberg. Learning with differential privacy: Stability, learnability and the sufficiency and necessity of erm principle. *J. Mach. Learn. Res.*, 17(1):6353–6392, jan 2016. ISSN 1532-4435.
 - [157] Brandon T Willard and Rémi Louf. Efficient guided generation for large language models. *arXiv e-prints*, pages arXiv–2307, 2023.
 - [158] Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. egg: Fast and extensible equality saturation. *Proc. ACM Program. Lang.*, 5(POPL):1–29, 2021. doi: 10.1145/3434304. URL <https://doi.org/10.1145/3434304>.
 - [159] Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. egg: Fast and extensible equality saturation. *Proc. ACM Program. Lang.*, 5(POPL), January 2021. doi: 10.1145/3434304. URL <https://doi.org/10.1145/3434304>.
 - [160] Yutong Xin, Jimmy Xin, Gabriel Poesia, Noah D. Goodman, Qiaochu Chen, and Işıl Dillig. Automated discovery of tactic libraries for interactive theorem proving. *Proc. ACM Program. Lang.*, 9(OOPSLA2), October 2025. doi: 10.1145/3763121. URL <https://doi.org/10.1145/3763121>.
 - [161] Peisen Yao, Qingkai Shi, Heqing Huang, and Charles Zhang. Program analysis via efficient symbolic abstraction. *Proc. ACM Program. Lang.*, 5(OOPSLA):1–32, 2021. doi: 10.1145/3485495. URL <https://doi.org/10.1145/3485495>.
 - [162] S. Zdancewic and A.C. Myers. Observational determinism for concurrent program security. In *16th IEEE Computer Security Foundations Workshop, 2003. Proceedings.*, pages 29–43, 2003. doi: 10.1109/CSFW.2003.1212703.
 - [163] Zhe Zhou, Robert Dickerson, Benjamin Delaware, and Suresh Jagannathan. Data-driven abductive inference of library specifications. *Proc. ACM Program. Lang.*, 5(OOPSLA):1–29, 2021. doi: 10.1145/3485493. URL <https://doi.org/10.1145/3485493>.
 - [164] He Zhu, Stephen Magill, and Suresh Jagannathan. A data-driven CHC solver. In Jeffrey S.

Foster and Dan Grossman, editors, *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, pages 707–721. ACM, 2018. doi: 10.1145/3192366.3192416. URL <https://doi.org/10.1145/3192366.3192416>.