Program Synthesis for Data Analysis: Scalability and Privacy

by

Calvin Smith

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN–MADISON

23 July 2020

Date of final oral examination: 05/15/2020

The dissertation is approved by the following members of the Final Oral Committee:
Aws Albarghouthi, Assistant Professor, Computer Sciences
Kassem Fawaz, Assistant Professor, Electrical and Computer Engineering
Justin Hsu, Assistant Professor, Computer Sciences
Somesh Jha, Professor, Computer Sciences
Thomas Reps, Professor, Computer Sciences

© Copyright by Calvin Smith 23 July 2020
All Rights Reserved

To Maggie, Tammy, and Lyndell.

ACKNOWLEDGMENTS

The arrival of COVID-19 has highlighted for me the importance of community; scholarship cannot exist in a vacuum. I am grateful to have had the patience and direction of my advisor, Aws Albarghouthi, who is single-handedly responsible for sparking my interest in PL research. It is the mentorship of Aditya Nori and Mayur Naik that gave context to that interest, and the brilliant faculty—Loris D'Antoni, Justin Hsu, Somesh Jha, Ben Liblit, Joe Miller, and Thomas Reps—in the MadPL group and elsewhere that provided the bedrock of knowledge for a successful thesis.

My doctoral career would not be the same without my cohort. Samuel Drews proved the best office-mate one could ask for, and John Cyphert and Jason Breck reliable travel companions. Jialu Bao, David Bingham Brown, Zhicheng Cai, Jordan Henkel, Qinheping Hu, Jinwoo Kim, Zachary Susang, Michael Vaughn, Yuhao Zhang, and Jinman Zhao all served as valued colleagues I am grateful to know. I've relied heavily on the continued friendship, across years and miles, of Jesse, Dillon, Ben C., Andrew, and many others who have provided succor.

I owe a great deal to my family. My parents, Tammy and Lyndell, and my sister Maggie have backed me unwaveringly throughout my academic career. I have learned much about the efficacy of hard work from them, and admire them all greatly. My aunt and uncle, Lazane and Ron, provided their considerable wisdom and experience, and at one point their finished basement, and the Kellers, the Smiths, and my grand-parents have all shown me their love and encouragement through the years.

For all the support, I am eternally grateful.

CONTENTS

Contents iii

List of Tables vi

List of Figures vii

Abstract ix

1 Introduction 1
- *1.1 Challenges in Enabling Effective Data Access* 2
- *1.2 Contributions* 4
- *1.3 Structure of this Dissertation* 5

2 Preliminary Notions 7
- *2.1 Differential Privacy in a Nutshell* 7
- *2.2 The Terminology of Terms* 11
- *2.3 Concretizing Program Synthesis* 19

3 Synthesizing Data-Processing Programs 23
- *3.1 Data-Parallel Programming Frameworks* 23
- *3.2 Higher-Order Sketches and the Synthesis Domain* 28
- *3.3 Compositional Synthesis Algorithm* 32
- *3.4 Commutative Semigroup Reducers* 39
- *3.5 Implementation and Evaluation* 43
- *3.6 Related Work* 50

4 Privacy-Aware Synthesis 53
- *4.1 Illustrating Privacy-Aware Synthesis* 54
- *4.2 The Synthesis Problem* 58
- *4.3 Synthesis with Linear Dependent Types* 66

4.4 *Applications of Privacy-Aware Synthesis* 81
4.5 *Implementation Details* 89
4.6 *Evaluation* 91
4.7 *Related Work* 99

5 Automating Proofs of High-Probability Guarantees 102
5.1 *Overview and Illustration* 103
5.2 *Programs, Automata, and Properties* 112
5.3 *Trace Abstraction Modulo Probability* 118
5.4 *Constructing Trace Abstractions* 121
5.5 *Labeling Individual Traces* 127
5.6 *Implementation and Case Studies* 136
5.7 *Related Work* 148

6 Exploiting Symmetries in Term Algebras 153
6.1 *Introduction to Equivalence Reduction* 153
6.2 *Overview of Synthesis Modulo Equations* 156
6.3 *Defining Synthesis Modulo Equations* 161
6.4 *Term Rewriting and Completion* 163
6.5 *Synthesis Modulo Equations* 166
6.6 *Properties of Equivalence Reduction* 169
6.7 *Implementation and Evaluation* 171
6.8 *Related Work* 183

7 Conclusion 185
7.1 *Future Directions and Extensions* 185
7.2 *Closing Remarks, or: a Promise* 187

A Properties of Data-Parallel Synthesis 188

B Properties of Privacy-Aware Synthesis 191

C Proofs for Probabilistic Trace Abstraction 205

D Equivalence Reduction Benchmarks 232

Colophon 236

References 237

LIST OF TABLES

3.1 Set of data-parallel components C_{DP} from Apache Spark 28
3.2 A sample of components used by BIGλ 44
3.3 BIGλ synthesis task results 45

4.1 Deterministic alternatives to probabilistic functions 65
4.2 Examples of functions in our synthesis domain and privacy mechanisms 81
4.3 Datasets ZINC was tested over 92
4.4 Results of evaluating ZINC 93

5.1 Distribution expressions and their semantics 113
5.2 Example families of distribution axioms 129
5.3 Results on private algorithms 139

6.1 Example synthesis domain over integers and strings 157
6.2 Partial list of equations for integer and string components . . . 157
6.3 Experimental equivalence reduction results 177

D.1 Descriptions of benchmarks from Table 6.3 233
D.2 Synthesis domain used in used in Section 6.7 234
D.3 All equations used in Section 6.7 235

LIST OF FIGURES

3.1 High-level view of a MapReduce computation on a simple example . . . 25
3.2 Non-associative/commutative binary reduce functions . . . 27
3.3 High-level illustration of Bigλ synthesis algorithm . . . 33
3.4 Inference rules defining the Bigλ synthesis algorithm . . . 37
3.5 Scalability experimental results for Bigλ . . . 50

4.1 Overview of privacy-aware synthesis technique and setting . . . 54
4.2 Applying the function f to adjacent databases m_1 and m_2 . . . 55
4.3 Implementation sketch for k-means, adapted from Gaboardi et al. (2013) . . . 58
4.4 Implementation sketch for idc, adapted from Gaboardi et al. (2013). . . . 58
4.5 Grammars of DFuzz types and programs . . . 62
4.6 Basic synthesis rules . . . 68
4.7 Advanced synthesis rules extending the procedure in Figure 4.6 69
4.8 Inference rules defining abduction . . . 75
4.9 Performance comparison between Zinc and baseline . . . 96
4.10 Partial synthesis problem for cdf (Gaboardi et al., 2013) . . . 97

5.1 Main loop of verification algorithm . . . 103
5.2 Examples of probabilistic programs . . . 105
5.3 A simple probabilistic program and possible trace annotations 105
5.4 A looping illustrative example . . . 110
5.5 Statement and trace semantics . . . 115
5.6 Overall abstract algorithm for implementing Theorem 5.5 . . . 122
5.7 Examples of merging automata . . . 125
5.8 Logical encoding of statement semantics . . . 131
5.9 Randomized Response (randResp) algorithm . . . 139

5.10 Noisy Sum (noisySum) algorithm 140
5.11 Report Noisy Max (noisyMax) algorithm 141
5.12 Discrete Exponential mechanism (expMech) 142
5.13 Above Threshold (aboveT) algorithm 144
5.14 Sparse Vector (sparseVec) algorithm 145
5.15 Reliable computing example (Carbin et al., 2013) 147

6.1 Overview of synthesis with equivalence reduction 155
6.2 Number of terms vs. term size with varying levels of equivalence reduction 160
6.3 Algorithms for synthesis with equivalence reduction 167
6.4 Building a perfect discrimination tree from left-hand side of rules 173
6.5 Addition of two complex numbers of the form $a + bi$, where a and b are represented as a `pair` 174
6.6 Equivalence reduction overhead 179
6.7 Average performance of $\textsc{Norm}(\cdot)$ on `sum-to-first` 180
6.8 Performance per number of rules sampled 181

C.1 Simplified logical encoding of statement semantics for feasible traces 216
C.2 The union bound logic, core rules (Barthe et al., 2016c) 220
C.3 Implementation of nondeterministic algorithm in Figure 5.6 . 227
C.4 Main loop of verification algorithm 231

ABSTRACT

This dissertation expands the applicability of program synthesis to data analysis, with a focus on scalability and privacy. Data is abundant and empowering, but barriers-to-access such as specialized technical knowledge or privacy requirements limit access. Program synthesis—the automatic generation of programs satisfying user intent—enables non-technical users to circumvent these barriers.

To provide access to large-scale data analysis techniques, we introduce Bigλ, a synthesis tool that utilizes *higher-order sketches* to generate scalable MapReduce programs from input-output examples. Higher-order sketches divide the work between mappers and reducers, and we introduce a verification technique that proves if a reducer is associative and commutative: a sufficient condition for the pipeline to be robust in the face of network-induced non-determinism. We demonstrate the efficacy of Bigλ by synthesizing a host of data analysis benchmarks on real-word data sets from a small number of examples.

We then study the problem of automatically synthesizing privacy-respecting programs using Zinc, a tool that automatically synthesizes probabilistic and provably-private programs. We base our technique on an effective inversion of the *linear dependent type system* DFuzz that tracks the resources consumed by a program, and hence its privacy cost. Zinc directs the synthesis towards programs satisfying a given privacy budget by using *symbolic context constraints* and *subtyping constraint abduction* to reason about the privacy-preserving behavior of partial programs. We then show Zinc's ability to automatically synthesize privacy-preserving data analysis queries, as well as recursive differential privacy mechanisms from the literature.

To reason about the properties of probabilistic programs, such as those produced by Zinc, we introduce *trace abstraction modulo probability*, a proof

technique for verifying high-probability guarantees. Our proofs over-approximate the set of program traces using *failure automata*, finite-state automata that upper-bound the probability of failing to satisfy a specification. We automate proof construction by a synthesis-enabled reduction of probabilistic reasoning to logical reasoning, which allows us to apply classic interpolation-based proof techniques. We evaluate our proof technique by proving properties of probabilistic programs drawn from the differential privacy literature. Our evaluation is the first instance of automatically-established accuracy properties—which contain symbolic inputs, parameterized distributions, and infinite state spaces—for these algorithms.

Finally, we discuss program synthesis with *equivalence reduction*, a methodology that utilizes equational specifications over a given synthesis domain to reduce the search space and improve performance. By leveraging classic and modern techniques from term rewriting, we use equations to induce a canonical representative per equivalence class of programs. We show how to design synthesis procedures that only consider canonical representatives, thus pruning the search space. We conclude by illustrating how to implement equivalence reduction using efficient data structures, and demonstrating the significant reductions it can achieve in overall synthesis time.

1 INTRODUCTION

The future is already here — it's just not evenly distributed.

— William Gibson (The Economist, December 4, 2003)

This thesis is about program synthesis for effective data analysis, with a focus on scalabilty and privacy. Modern society collects information on nearly every imaginable facet of life via mechanisms such as e-commerce, social media, electronic medical records, and the ever-present smartphones in the pockets of 3.5 billion people. Access to this digital panopticon is highly valued: data, and the interpretation thereof, is empowering. Given the importance of data to our society, it is imperative that we strive for *responsible data democratization*, both to magnify the benefits of every byte, but also to limit data access asymmetries that can lead to manipulation and oppression.

Merely making information public is insufficient. The complexity of data storage and access mechanisms—clusters and databases and distributed computing platforms—are a large burden-of-knowledge standing in the way of *effective* data use. To complicate matters, much sought-after data is about people, and we have a responsibility to ensure open data does not invite abuse or intrude upon their rights, chief among which is the right to privacy. On this front, recent advancements in *differential privacy* provide a way forward: by carefully adding noise to queries, differential privacy mechanisms allow for the provably secure release of sensitive information.

Unfortunately, differential privacy is hard to get right. Industry techniques are dependent on complicated one-off inference procedures (Erlingsson et al., 2014), and even privacy experts occasionally implement incorrect variants of common mechanisms (Lyu et al., 2016). Even framing a query requires error-prone analysis of possible information leakages. For

all the benefits, differential privacy represents another *significant* burden-of-knowledge in the way of effective data use.

Advances in automation stand ready to help: program synthesis—the automatic construction of programs from easily-provided user intent—provides non-programmers with an interface for computation. By construcing scalable and privacy-aware queries from simple examples, and automating analyses and proofs of said queries, program synthesis lets users frame questions and analyze the results without having to know a thing about protocols or backends. This dissertation, in lowering the burden-of-knowledge neessary for effective access of data, advances the state-of-the-art in program synthesis by constructing synthesis tools that address the challenges in scalably accessing and understanding data in a privacy-preserving way.

1.1 Challenges in Enabling Effective Data Access

Program synthesis, by converting easily-specified user intent into non-trivial programs, promises to be a computational interface for those with limited programming experience. Developments in program synthesis have brought powerful automation techniques across a variety of domains within reach of the average computer user (Gulwani, 2011; Barowy et al., 2015; Le and Gulwani, 2014; Osera and Zdancewic, 2015; Feser et al., 2015; Perelman et al., 2014; Yaghmazadeh et al., 2017; Wang et al., 2017a; Feng et al., 2017). However, each new program synthesis domain requires specialization in order to raise the level of abstraction to the point of usability. In the context of effective access of data, this means an end-user should not have to worry about the details of data access mechanisms and privacy enforcement strategies, leaving their focus solely on the asking the questions and understanding the answers

Framing the Question

The goal of program synthesis is to convert an easily specified user intent—e.g., input-output examples—into a full-fledged, executable program. Depending on the use-case, however, the quality of a synthesized program is dependent on more than just consistency with user intent. To meet our goal, we wish to enable individuals to (*i*) effectively interact with large-scale data processing tools, and (*ii*) frame questions in privacy-preserving ways, *without* requiring expertise in data-parallel computation and differential privacy.

A data-parallel computation tool like MapReduce (Dean and Ghemawat, 2004) hides fiddly details like load balancing, but still requires the user to craft a program in terms of higher-order data-parallel operators like `map` and `reduce`, and choosing the wrong operators or division-of-labor between them can introduce bottlenecks or network-induced non-determinism. Other tools, such as Airavat (Roy et al., 2010) and PINQ (McSherry, 2009), provide privacy-preserving data access by letting developers provide queries in the form of programs. However, not all privacy-preserving functions are equivalent: a program that answers the desired question, but also releases unnecessary information, is punished by having more noise added. Users are expected to understand how function composition and application impacts the final privacy guarantee; a difficult task for privacy experts (Lyu et al., 2016), let alone the average non-programmer.

Interpreting Privacy-Preserving Answers

For well-known privacy mechanisms, such as the Laplace and exponential mechanisms (Dwork and Roth, 2014), expert scrutiny has precisely quantified the impact the addition of noise has on the utility of the result. This quantification is framed as a *high-probability guarantee*: the output satisfies

some accuracy condition ϕ, which might, e.g., bound the difference between the returned result and the optimal value, *except* with some failure probability p. But proving high-probability guarantees over probabilistic programs is tedious work, and unfortunately beyond the ability of the average non-technical user. Automated tools often focus on restricted, tractable models of probabilistic programs (see the surveys by Baier et al. (2018) and Katoen (2016)), or rely on the fact that programs are *closed* and have finite state space (Kwiatkowska et al., 2011; Dehnert et al., 2017), neither of which is true in the case of differential privacy, where distributions often rely on symbolic inputs and are difficult to integrate.

1.2 Contributions

In advancing the state-of-the-art of program synthesis for effective, privacy-aware data access, this dissertation makes the following contributions:

C1 We present a *compositional program synthesis algorithm* that automatically constructs data-parallel programs in the MapReduce framework (Chapter 3). Our implementation—Bɪɢλ—constructs distributed programs that scale well with data and available computational power. To ensure programs are resistant to network-induced non-determinism, we apply *hyperproperty verification techniques* to prove that reducers form commutative semigroups Section 3.4). Lastly, we empirically demonstrate Bɪɢλ's efficiency and applicability by synthesizing a range of distributed data analysis tasks on real-world datasets (Section 3.5).

C2 We extend type-directed synthesis to a privacy-tracking linear dependent type system to construct a *sensitivity-directed synthesis* algorithm (Chapter 4). The algorithm employs *symbolic context constraints* and *subtyping constraint abduction* to direct the search towards programs satisfying a provided privacy budget (Section 4.3). We describe how sensitivity-directed

synthesis can: (*i*) synthesize privacy-preserving data-analysis queries using higher-order combinators, and (*ii*) synthesize recursive differential privacy mechanisms from the literature (Section 4.4). We implement our algorithm in a tool called ZINC and demonstrate the importance of sensitivity-direction on synthesis performance (Section 4.6).

C3 We present an automated proof technique for probabilistic accuracy properties based on *trace abstraction* (Chapter 5). The technique uses *Craig interpolation* to construct *failure automata* (Section 5.4) from proofs of correctness of probabilistic traces, which it generates via a reduction to a non-deterministic *constraint-based synthesis* problem (Section 5.5). The proof technique is implemented and used to verify differentially private programs, where accuracy properties include symbolic distribution parameters and parametric inputs (Section 5.6).

C4 We incorporate domain knowledge in the form of *equational specifications* into program synthesis via *equivalence reduction* (Chapter 6). To do so, we utilize modern techniques from theorem proving to impose normal forms on programs, and incorporate normality-checking into bottom-up and top-down synthesis algorithms (Sections 6.3 and 6.5). Lastly, we empirically evaluate the impacts of equivalence reduction to (*i*) demonstrate the necessity of fast normality checking algorithms and efficient data structures such as *perfect discrimination trees* (McCune, 1992), and (*ii*) understand the impacts of equivalence reduction on program synthesis across different domains and algorithms (Section 6.7).

1.3 Structure of this Dissertation

In Chapter 2, we introduce preliminary definitions and notation. The meat of the contributions are in Chapters 3 to 6. Based on the reader's interest, we recommend the following two sub-dissertations:

SD1 For those primarily interested in *program synthesis*, we recommend Sections 2.2 and 2.3 for preliminary remarks, and then Chapters 3, 4 and 6.

SD2 For a focus on *differential privacy*, read instead Section 2.1 for an overview, followed by Chapters 4 and 5.

Finally, we provide closing remarks and future directions in Chapter 7.

The contents of this thesis are based on four papers (Smith and Albarghouthi, 2016, 2019b; Smith et al., 2019; Smith and Albarghouthi, 2019a), indicated at the beginning of a section when relevant.

2 PRELIMINARY NOTIONS

In this chapter, we present some preliminary notions to frame the work to follow. In Section 2.1, we give a brief overview of differential privacy. Section 2.2 provides an introduction to the construction and use of term algebras. Finally, Section 2.3 concretizes the program synthesis problem.

2.1 Differential Privacy in a Nutshell

To better understand the challenges inherent in data access via privacy-preserving mechanisms, we must understand what differential privacy promises and how it upholds those promises. What follows in this chapter is a high-level view of differential privacy; for a more in-depth presentation, we recommend the work of Dwork and Roth (2014).

Differential privacy is framed as a *robustness* property of probabilistic programs: the same query on similar inputs should produce similar outputs.

Definition 2.1 (Differential Privacy)**.** *Let* A *be a space with metric* d_A*, and let* B *be a space. For* $\epsilon, \delta \in \mathbb{R}^{\geqslant 0}$*, a probabilistic function* $f : A \to B$ *is* (ϵ, δ)*-differentially private (*(ϵ, δ)*-DP) if and only if, for all* $x, y \in A$ *with* $d_A(x, y) \leqslant 1$ *and for all* $S \subseteq B$*, we have*

$$Pr\left[f(x) \in S\right] \leqslant e^{\epsilon} \cdot Pr\left[f(y) \in S\right] + \delta$$

When δ *is* 0*, we say* f *is* ϵ*-differentially private.*

The strictness of the privacy-preservation is determined by the parameters ϵ and δ—the smaller they are, the more privacy is preserved. In the rest of this work, we focus on the more restrictive case where $\delta = 0$.

The subtlety of Definition 2.1 comes, in part, from determining the appropriate metric d_A. While metrics over some domains are natural —

e.g., $\mathbb{R}$ and $\mathbb{N}$ with the Euclidean metric — we care about enforcing privacy over *data sets*, which require some care to model mathematically.

Definition 2.2 (Data Sets). *We will represent data sets as* multisets *of rows, and denote the space of all data sets as* $\mathcal{D}$.

$\mathcal{D}$ *has the following metric: for* $x, y \in \mathcal{D}$, $d_{\mathcal{D}}(x, y) = |x \triangle y|$, *the cardinality of the* symmetric difference *of* x *and* y.

There are many metrics over multisets. The one chosen encodes the assumption that each row in a data set represents an individual. To protect *individual privacy*, we want queries to be robust to the presence of an individual's data.

Benefits of Differential Privacy

Differential privacy is an attractive notion of privacy. In the face of linkage attacks and de-anonymization, the fact that differential privacy is *immune to post-processing* means that the results of a query, once set loose into the public, cannot yield more information than what was originally released, even with the most stubborn of attacks.

Theorem 2.3 (DP Immune to Post-Processing). *Let* A *and* B *be spaces, with* d_A *a metric for* A, *and let* $f : A \to B$ *be an* ϵ-DP *probabilistic function.*

Let $g : B \to C$ *be any deterministic function. Then* $g \circ f$ *is* ϵ-DP.

To best utilize such a guarantee, it is important to understand what impact the information originally released can have. In traditional discussions on privacy, impacts are often framed as a strong dichotomy: either an individual is protected, or their data is compromised. Such discussions don't offer much of a path forward when it comes to regulatory action, as the responsible response to the all-or-nothing framing of impacts is to cloister data and prevent any unauthorized access.

Fortunately, differential privacy admits a notion of *quantitative privacy loss* parameterized by ϵ. We will illustrate this with an example: suppose Alex is an individual who is contacted by the Fictional research group, who would like to use Alex's medical data for a study. Fictional is federally funded, and consequently mandated to ensure their analysis is ϵ-dp—that is, Alex knows there is some ϵ-dp function $f : \mathcal{D} \to R$ representing the study, where R is the space of all possible outcomes of the research. Some of these outcomes are more desirable than others. Alex represents this fact with a *utility function* $u : R \to \mathbb{R}^+$, which assigns high (resp., low) weights to desirable (resp., undesirable) outcomes.

To decide what to do, Alex considers two possible outcomes: (*i*) Alex gives Fictional their data, and they run their analysis on a, which includes Alex's data, or (*ii*) Alex withholds their data, and Fictional runs their analysis on b, which is identical to a sans Alex's data. Surely, $d_{\mathcal{D}}(a, b) = 1$, and so Alex is confident the privacy guarantee will hold. Alex then computes:

$$\begin{aligned}
\mathbb{E}_{r \sim f(a)}[u(r)] &= \sum_{r \in R} u(r) \cdot \Pr[f(a)]\, r \\
&\leqslant \sum_{r \in r} u(r) \cdot e^{\epsilon} \cdot \Pr[f(b))]\, r \\
&= e^{\epsilon} \cdot \mathbb{E}_{r \sim f(b)}[u(r)]
\end{aligned}$$

Regardless of the outcome of the study, Alex's two choices result in expected utility differing at most by a multiplicative factor of e^{ϵ}. If the published ϵ is small enough, Alex may decide that the personal risk is worth it and give their data to Fictional.

Understanding Functions through Sensitivity

Differential privacy is a property of probabilistic programs, but most analyses one wants to perform are deterministic. To make a deterministic

function private, we must strategically add noise in order to enforce the guarantee of Definition 2.1. Privacy mechanisms that convert deterministic functions typically require some information about the function, usually in a form of robustness called *sensitivity*:

Definition 2.4 (Sensitivity)**.** *Let* A *and* B *be spaces with metrics* d_A *and* d_B*, respectively. For* $c \in \mathbb{R}^{\geqslant 0}$*, a* deterministic *function* $f : A \to B$ *is* c-sensitive *if and only if:*

$$\forall x, y \in A.\, d_B(f(x), f(y)) \leqslant c \cdot d_A(x, y)$$

Note that sensitivity is an upper-bound: a c-sensitive function is *also* c'-sensitive, for every $c' \geqslant c$. Another important property of sensitivity is that it composes *multiplicatively*: given a c_1-sensitive function g and a c_2-sensitive function f, the composition $g \circ f$ is $(c_1 \cdot c_2)$-sensitive.

Privacy Enforcing Mechanisms

The most natural way to enforce differential privacy is to add randomly-sampled noise to a real-valued function. When that noise is drawn from the Laplace distribution, we have the Laplace mechanism:

Definition 2.5 (Laplace Mechanism)**.** *Let* A *be a space with metric* d_A*. Let* $c, \epsilon \in \mathbb{R}^{\geqslant 0}$*, and let* $f : A \to \mathbb{R}$ *be a* c*-sensitive function.*

The Laplace mechanism $\mathit{LapMech}(f, a, \epsilon)$ *selects and returns an output from the distribution* $\mathit{Lap}(f(a), c/\epsilon)$*, where* $\mathit{Lap}(m, s)$ *is the* Laplace *distribution with mean* m *and scale* d*.*

The mechanism $\mathit{LapMech}(f, a, \epsilon)$ *is* ϵ-DP*.*

Of course, the Laplace mechanism is only effective with real-valued functions whose *utility* is relatively continuous: if varying an answer by a small amount destroys the value of the information (such as in auction pricing), or the answer is categorical (how do we add noise to categories?),

we need a different mechanism. For these conditions, the exponential mechanism is more useful:

Definition 2.6 (Exponential Mechanism)**.** *Let* A *and* B *be spaces with metrics* d_A *and* d_B*, respectively. Let* $c, \epsilon \in \mathbb{R}^{\geqslant 0}$*, and let* $u : A \to B \to \mathbb{R}$ *be a* utility function c*-sensitive in the second argument.*

The exponential mechanism $\texttt{ExpMech}(B, u, a, \epsilon)$ *selects and returns an output* $b \in B$ *with probability proportional to*

$$\exp\left(\frac{\epsilon \cdot u(a, b)}{2c}\right)$$

The mechanism $\texttt{ExpMech}(B, u, a, \epsilon)$ *is* ϵ*-DP.*

Intuitively, the exponential mechanism weights each possible output by bow useful it is, before smoothing the weights and sampling outputs from the resulting proportional distribution. There is a slight semantic gap between the computation performed by $\texttt{ExpMech}(\dots)$ and the operation of u; as such, the exponential mechanism is used primarily by algorithm designers, and not end users attempting to access data. We will see in Section 4.4, however, how program synthesis can bridge this gap.

Other mechanisms focus on optimizing particular workloads or enforcing different notions of privacy. We will discuss them as they become relevant.

2.2 The Terminology of Terms

Based on the use case, the form of the artifacts generated by program synthesis can vary wildly. Here, we will introduce a framework for the work that follows built out of the formalisms of *term algebras*.

Term algebras contain and construct *terms*. Pinning down programs is fiddly, as one needs to simultaneously consider syntax, semantics, and

implementation; in contrast, terms are purely syntactic objects. They are, in fact, *the most syntactic object one can have* in a sense that will be made precise in Section 2.2. We think of terms as trees whose nodes are labeled by variables, constants, and operations *without any associated interpretation*.

Signatures and Function Symbols

Constants and operations are grouped together into a *signature*, which defines something similar to an interface: each constant and operation is named, and annotated with the *sorts* (or types) of their inputs and their output.

Definition 2.7 (Signature). *A* signature *is a pair* (S, Σ) *comprising:*

1. *A set of* sorts S*, and*

2. *An* $(S^* \times S)$*-indexed set of* function symbols $\Sigma = \bigsqcup_{w \in S^*, s \in S} \Sigma_{w,s}$*, where each* $\Sigma_{w,s}$ *is disjoint.*

When clear from context, we will refer to signature (S, Σ) *simply as* Σ*.*

We will define some additional notation for function symbols: for $f \in \Sigma_{w,s}$, we define the *arity* of f (written $\mathrm{ar}(f)$) as $|w|$. A function symbol $c \in \Sigma_{\epsilon,s}$ (that is, with $\mathrm{ar}(c) = 0$) is a *constant*. Lastly, we will use the alternate notation $f : w \to s$ to indicate (*i*) the existence of a set $\Sigma_{w,s}$ and (*ii*) f's inclusion in that set.

For an example, consider a simple arithmetic framework which only has 0, a successor function, and a binary addition operator:

$$\Sigma^A \coloneqq (\{n\}, \{0 : \epsilon \to n, \mathtt{Succ} : n \to n, + : nn \to n\})$$

where n is a sort representing the natural numbers. It is important to stress that the sort is not *inhabited* by anything, and the function symbols

0, `Succ`, and $+$ do not have any operational interpretation beyond their input and output sorts. That will come shortly.

A more interesting construction can be had via *context-free grammars* (CFGs): non-terminals become sorts, and the function symbols are derived from the production rules. In our application of signatures, this construction is equivalent to interpreting a CFG as a *regular tree language*, where the language defined by the grammar is over derivation trees instead of strings.

Definition 2.8 (Signature of a CFG)**.** *Let* G *be a context-free grammar with non-terminals* N*, a disjoint set of terminals* T*, and production rules of the form* $n \to w$*, where* $n \in N$ *and* $w \in (N \cup T)^*$*.*

The signature of G*, denoted* Σ^G*, is the pair* (N, Σ^G)*, where* Σ^G *is populated as follows: let* $n \to w$ *be a production rule, and let* w' *be the string constructed by dropping all terminal symbols from* w*. Then* $r_{n \to w} \in \Sigma^G_{w',n}$*, where* $r_{n \to w}$ *is a fresh function symbol.*

Algebras

If a signature is an interface, an algebra is an implementation. By defining how the function symbols operate over some target domain, called the *carrier*, an algebra represents a model of a signature and provides a computational interpretation.

Definition 2.9 (Algebra)**.** *Let* (S, Σ) *be a signature. A* Σ*-algebra* $(A, [\![\cdot]\!])$ *is a pair comprising:*

1. *An* S*-indexed* carrier set $A = \bigcup_{s \in S} A(s)$*, and*
2. *An* interpretation function $[\![\cdot]\!]$ *that converts every function symbol* $f \in \Sigma_{w,s}$ *into a function* $[\![f]\!] : A^w \to A(s)$*, where* $A^w = A(s_1) \times A(s_2) \times \ldots A(s_n)$ *if* $w = s_1 s_2 \ldots s_n$*.*

When clear from context, we will refer to the Σ*-algebra* $(A, [\![\cdot]\!])$ *simply by* A*.*

Consider the signature Σ^A: to provide the usual interpretation to the function symbols, we could construct the Σ^A-algebra $(A, [\![\cdot]\!])$ where (*i*) the carrier set A has a single index, n, and $A(n) = \mathbb{N}$, and (*ii*) $[\![\cdot]\!]$ converts the function symbols as follows:

$$[\![0]\!] = 0$$
$$[\![\texttt{Succ}]\!](n) = n + 1$$
$$[\![+]\!](x, y) = x + y$$

While our algebra interprets the function symbols in Σ^A as expected, there is nothing in the structure of Σ^A that restricts the semantics to coincide, e.g., with the Peano axioms. That is to say, we could instead define $[\![+]\!](x, y) = x$, which is not commutative. Either interpretation is a valid model of Σ^A.

The Term Algebra

In a term algebra, the computational interpretation of a signature is the *construction of a term*: the carrier objects are syntactic terms, and the function symbols are interpreted as term constructors.

Definition 2.10 (Term Algebra)**.** *Let* (S, Σ) *be a signature. The* term algebra $(T_\Sigma, [\![\cdot]\!])$ *is the* Σ*-algebra defined as follows:*

1. Carrier Set*: The carrier set* T_Σ *is defined* recursively*, as follows:*
    1. *For each sort* $s \in S$*, and each constant* $c \in \Sigma_{\epsilon,s}$*,* $c \in T_\Sigma(s)$
    2. *For each function symbol* $f \in \Sigma_{w,s}$*, where* $w = s_1 s_2 \ldots s_n$*, and for each* $t_i \in T_\Sigma(s_i)$*,* $f(t_1, t_2, \ldots, t_n) \in T_\Sigma(s)$
2. Interpretation*: For each function symbol* $f \in \Sigma_{w,s}$*, where* $w = s_1 s_2 \ldots s_n$*, and for each* $t_i \in T_\Sigma(s_i)$*, we define* $[\![f]\!](t_1, t_2, \ldots, t_n) = f(t_1, t_2, \ldots, t_n)$.

So the term algebra for our arithmetic signature — T_{Σ^A} — contains terms built only out of the function symbols 0, $\texttt{Succ}$, and $+$:

$$0 \qquad \texttt{Succ}(0) \qquad 0+0 \qquad 0+\texttt{Succ}(0) \qquad \texttt{Succ}(0+0) \qquad \ldots$$

and when we interpret a function symbol, we treat it as a term constructor, e.g.:

$$[\![+]\!]\,(0, 0+\texttt{Succ}(0)) = 0 + (0+\texttt{Succ}(0))$$

Why Terms?

Our interest in terms stems primarily from their relation to every other Σ-algebra. To frame this relation precisely, we will view the class of Σ-algebras as a *category*:

Definition 2.11 (Category of Σ-Algebras)**.** *Let Σ be a signature, and let A and B be two Σ-algebras. We say a function $h : A \to B$ between the carrier sets of A and B is a* homomorphism *if, for every $w \in S^*$, $s \in S$, and $f \in \Sigma_{w,s}$, the following diagram commutes:*

$$\begin{array}{ccc} A & \xrightarrow{h} & B \\ \downarrow{\scriptstyle [\![f]\!]_A} & & \downarrow{\scriptstyle [\![f]\!]_B} \\ A & \xrightarrow{h} & B \end{array}$$

Denote by $\textsc{Alg}\,(\Sigma)$ the category whose objects are Σ-algebras and whose arrows are homomorphisms.

Algebra homomorphisms are maps between algebras that commute with function application. While many pairs of algebras will have meaningful homomorphisms between them, the homomorphisms between the term algebra and all other algebras are special, because (*i*) they are guaranteed to exist, and (*ii*) they are unique. More precisely, term algebras are initial:

Theorem 2.12 (Term Algebras are Initial)**.** *Let* Σ *be a signature. The term algebra* T_Σ *is* initial *in* $\textsc{Alg}(\Sigma)$*: for every* Σ*-algebra* A*, there is a* unique *homomorphism* $h_{eval} : T_\Sigma \to A$*, which we refer to as the evaluation homomorphism.*

As a consequence of being initial, term algebras are the *only* algebras that have unique homomorphisms to every other algebra. Interestingly, the evaluation homomorphism's primary computational power comes solely from its ability to commute over function applications. As an example, consider the natural Σ^A-algebra A and a term $0 + \texttt{Succ}(0)$ to be evaluated:

$$h_{\text{eval}}(0 + \texttt{Succ}(0)) = h_{\text{eval}}(\llbracket + \rrbracket_{T_{\Sigma A}} (\llbracket 0 \rrbracket_{T_{\Sigma A}}, \llbracket \texttt{Succ} \rrbracket_{T_{\Sigma A}} (\llbracket 0 \rrbracket_{T_{\Sigma A}}))) \tag{2.1}$$

$$= \llbracket + \rrbracket_A (\llbracket 0 \rrbracket_A, \llbracket \texttt{Succ} \rrbracket_A (\llbracket 0 \rrbracket_A)) \tag{2.2}$$

$$= 1 \tag{2.3}$$

Equation (2.1) is justified by the construction of T_{Σ^A}—the only way terms are generated is by application of the term constructors. Equation (2.2) follows directly from the fact that h_{eval} is a homomorphism, and Equation (2.3) is simple evaluation in the algebra A.

The fact that term algebras are initial is the reason why we care about terms. The ability to unambiguously convert a term into any other algebra via the evaluation homomorphism means that, as we require different computational interpretations of the terms being synthesized—the default semantics, a logical encoding, or even a weight function—we simply construct the relevant algebra and get the ability to evaluate terms into objects in that algebra for free.

Incomplete Terms

With a term algebra, our only mechanism for generating terms is *bottom-up*: we take an n-arity function symbol f and n appropriately-sorted terms

$t_1, \ldots, t_n$ before constructing the newer, larger term $[\![f]\!](t_1, \ldots, t_n)$. There are times, however, when constructing terms is better done *top-down*. Instead of only gluing existing terms together, we would like to construct terms with *holes* in them, to be filled in later. We need to extend our signature with variables.

Definition 2.13 (Free Term Algebra)**.** *Let* (S, Σ) *be a signature, and let* $X = \bigsqcup_{s \in S} X(s)$ *be an* S*-sorted set of* variable symbols *disjoint from all symbols in* Σ*. The* free term algebra *generated by* Σ *and* X *(denoted* $(T_\Sigma(X), [\![\cdot]\!])$*) is constructed as follows:*

1. Carrier Set*: The carrier set* $T_\Sigma(X)$ *is defined recursively:*
   1. *For each sort* $s \in S$ *and variable* $x \in X(s)$*,* $x \in T_\Sigma(X)(s)$
   2. *For each sort* $s \in S$ *and constant* $c \in \Sigma_{\epsilon,s}$*,* $c \in T_\Sigma(X)(s)$
   3. *For each function symbol* $f \in \Sigma_{w,s}$*, where* $w = s_1 s_2 \ldots s_n$*, and for each* $t_i \in T_\Sigma(X)(s_i)$*,* $f(t_1, t_2, \ldots, t_n) \in T_\Sigma(X)(s)$
2. Interpretation*: The interpretation* $[\![\cdot]\!]$ *is defined as in Definition 2.10.*

Do note that our term algebras T_Σ introduced in Definition 2.10 are equivalent to the free term algebra $T_\Sigma(\emptyset)$. We refer to this variable-laden construction as a *free* term algebra precisely because it is the free Σ-algebra generated by X:

Theorem 2.14 (Free Term Algebras are Free)**.** *Let* Σ *be a signature, and* X *an* S*-sorted set of variables.*

$T_\Sigma(X)$ *is the free* Σ*-algebra generated by* X*, by which we mean: let* A *be a* Σ*-algebra, and let* $\nu : X \to A$ *be an assignment function mapping variables to the carrier set of* A*. Then* ν *uniquely extends to a homomorphism* $\nu^* : T_\Sigma(X) \to A$ *such that the following diagram commutes:*

$$\begin{array}{ccc} X & \xrightarrow{\text{id}} & T_\Sigma(X) \\ & \searrow^{\nu} & \downarrow \nu^* \\ & & A \end{array}$$

In other words, we can introduce variables into terms (by using $T_\Sigma(X)$), assign those variables to other terms using $\nu : X \to T_\Sigma(X)$, and acquire (for free) the *substitution* $\nu^* : T_\Sigma(X) \to T_\Sigma(X)$. Using variables, we can formalize some notions that will aid us in synthesizing terms later.

Contexts and Boxes

Suppose we have done the work to build a term t bottom-up, and now we would like to refer to a *sub-term* of t (call it s) in the context of t. More precisely, we would like to *decompose* t into $t = C[s]$, where C is a *context*.

Definition 2.15 (Term Contexts)**.** *Let Σ be a signature, and let $\square = \bigsqcup_{s \in S} \{\square_s\}$ be an S-sorted set of variables.*

A context C *is a term in $T_\Sigma(\square)$ that contains exactly one instance of a variable, which we will simply refer to as $\square$.*

Let s is a term in T_Σ, and denote by $\nu : \square \to T_\Sigma$ the assignment mapping the variable $\square$ that occurs in C to s. We will write $C[s]$ to denote the term $\nu^*(C) \in T_\Sigma$, where ν^* is the unique homomorphism induced by applying Theorem 2.14 to ν.

We will liberally make use of the context notation to select and replace sub-terms.

Wildcard

When constructing terms top-down, we need a mechanism for representing an *incomplete* term. We will mark holes in the term with specially-labeled variables called *wildcards* (written $\bullet$). Although our terminology may be unique, the idea is not: representing incomplete programs as terms

with wildcards is similar to the refinement trees of Osera and Zdancewic (2015).

Definition 2.16 (Wildcards and Incomplete Programs)**.** *Let Σ be a signature, and let W be an infinite, S-sorted set of* wildcards *(denoted $\bullet$).*

A term in $T_\Sigma(\bullet)$ is said to be complete *if it contains no wildcards (and* incomplete *otherwise). A complete term is uniquely identifiable with a term in T_Σ, via an extension of the empty assignment using Theorem 2.14. When clear from context, we will refer to complete and incomplete programs without making reference to the free term algebra.*

Unless otherwise stated, all wildcards in a term are distinct from all other variables.

If required by an application, wildcards can be *annotated* to carry information (see, for example, the wildcards in Chapter 4). Such annotations are a convenient mechanism to pass relevant context to synthesis subproblems, and can make algorithms and implementations much cleaner.

2.3 Concretizing Program Synthesis

We can now formalize program synthesis tasks and their solutions:

Definition 2.17 (Program Synthesis)**.** *A* program synthesis problem *is a tuple (Σ, ϕ, h) given by:*

1. *A* signature Σ *characterizing a search space,*
2. *A* correctness constraint ϕ *characterizing the solution space, and*
3. *A* qualitative objective h *to be minimized.*

A term $t \in T_\Sigma$ is a solution *to (Σ, ϕ, h) if and only if t satisfies the correctness constraint (written $t \models \phi$) and minimizes $h(t)$ compared to other solutions.*

That is, t *is a solution if and only if:*

$$t \in \operatorname{argmin}_{t \in T_\Sigma, t \models \phi} h(t)$$

While program synthesis problems are framed as optimization tasks, existing optimization techniques are rarely applicable, due to the highly symbolic and discrete nature of the search space T_Σ. Instead, we often rely on clever enumeration to generate candidate solutions in a smart order. Starting in Chapter 3 we will see how one can approach efficiently enumerating T_Σ—in the rest of this section, we will examine various forms of (*i*) the correctness constraint and (*ii*) the qualitative objective.

Correctness Constraints

Unlike many optimization tasks, in program synthesis we have the ability to check solutions against a correctness constraint that is often provided as a logical formula acting over some encoding of programs. While issues of undecidability do creep in, most synthesis tools will only provide a solution to the user if there is *proof* that the program satisfies the constraint.

Correctness constraints cannot usually be stated and left as-is: one must also provide a mechanism for deciding the judgement $t \models \phi$, which depends on the form of ϕ. Fortunately, our search space is the initial object in $\textsc{Alg}(\Sigma)$, and so we can make use of the immediate translation to any Σ-algebra that may be relevant.

Examples Arguably the most common form of correctness constraint—Gulwani (2011); Albarghouthi et al. (2013); Gulwani et al. (2011, 2012); Harris and Gulwani (2011); Osera and Zdancewic (2015); Smith and Albarghouthi (2016, 2019b); Osera and Zdancewic (2015); Polozov and Gulwani (2015); Feng et al. (2017) are just a few instances—examples serve as a user-friendly and low-cost way of defining intent. In their most straight-

forward presentation, examples are pairs $\langle i, o \rangle$ that require a solution p satisfy $p(i) = o$. In most cases, checking examples requires the ability to evaluate terms.

Types When function symbols represent operations in a system imbued with types, we should reasonably expect that the desired solution is well-typed. In simple cases, type-checking is easy (Feser et al., 2015; Smith and Albarghouthi, 2016) and serves to prune nonsense terms from the search, but more sophisticated type systems encode information about the problem domain (Osera and Zdancewic, 2015; Frankle et al., 2016; Smith and Albarghouthi, 2019b; Polikarpova et al., 2016), in which case type-checking often needs to be woven tightly into the enumeration.

Logical Constraints Instead of defining the behavior of a solution on a few points, some synthesis tools allow users to encode desirable properties into a logical constraint (Solar-Lezama et al., 2006; Wang et al., 2017b; Alur et al., 2013; Polikarpova and Sergey, 2019). Many are based on proof search techniques, or use a counterexample-guided loop to iteratively refine solutions.

The Qualitative Objective

Because we already have our correctness constraint in ϕ, in many synthesis applications the qualitative objective is closer to a regularizer than it is a loss function. In practice, most synthesis techniques integrate h into the enumeration of candidates, so that programs are *produced* in close to h-increasing order. Others, however, view optimization as the primary goal and structure the search to guarantee optimization of the qualitative objective (Hu and D'Antoni, 2018; Bornholt et al., 2016).

Program Size Folk wisdom suggests that small programs generalize better than large programs, and so are more desirable. Nearly every synthesis tool will consider solutions from smallest to largest, although some are less strict about the order, and others will jump the queue to peer deeper into the search space (Alur et al., 2015).

Performance Constraints Superoptimization is often framed as a synthesis task, where solutions are code snippets that (*i*) match the semantics of the original code and (*ii*) have higher performance than the original (Phothilimthana et al., 2016). Other tools apply statistical techniques like Monte Carlo Markov Chain methods to heuristically direct the search towards high performance solutions (Schkufza et al., 2013).

3 SYNTHESIZING DATA-PROCESSING PROGRAMS

Current data production—estimated to be about 1.7MB per person per second—necessitates a focus on scalability. Developments in distributed data access mechanisms, such as Google's original MapReduce (Dean and Ghemawat, 2004) and related techniques (White, 2015; Zaharia et al., 2012; Yu et al., 2008), do much to let developers construct data processing pipeline that scale well in the cloud while abstracting away issues like network topology and node failures. Yet, they still require users to construct programs using higher-order operators like `map` and `reduce`. We expect this interface to be beyond the scope of the average non-developer: choosing the wrong operators, or the wrong workload balance between them, can easily result in inefficiencies, bottlenecks, and even network-induced non-determinism. In this chapter, we address the challenge of synthesizing efficiently scalable data processing programs from user-provided examples. The contents of this chapter are based on the work of Smith and Albarghouthi (2016).

3.1 Data-Parallel Programming Frameworks

Since the introduction of Google's MapReduce system in Dean and Ghemawat's seminal paper (Dean and Ghemawat, 2004), a number of powerful systems that implement and extend the MapReduce paradigm have been proposed, e.g., Hadoop (White, 2015), Spark (Zaharia et al., 2012), and Dryad (Yu et al., 2008), amongst others (Flink; Halperin et al., 2014; Alsubaiee et al., 2014). For the purposes of our work here, we present a generic view of data-parallel programs as functional programs.

In its simplest form, a MapReduce program contains an application of `map` followed by an application of `reduceByKey`:

```
map m ∘ reduceByKey r
```

where `m` has type $\tau \to (k, v)$ and `r` has type $v \to v \to v$. Given a list of elements of type τ, `map` applies `m` in parallel to each element to produce a list of key-value pairs (k, v). Then, for each key produced by m, `reduceByKey` receives a list of elements of type v—all values associated with the key—and *aggregates* (or *folds*) each such list using `r`. The result of this computation is a list of key-value pairs of type (k, v), where each key appears once.

Suppose we are given a list of words and we would like to count the number of occurrences of each word in the list. We can do this with the following function:

```
let count = map m ∘ reduceByKey r
where
   m w = (w, 1)
   r a b = a + b
```

For each input word `w`, the mapper emits the key-value pair $(w, 1)$; the reducer then sums the values associated with each word, producing a list $[(w_1, v_1), \ldots, (w_n, v_n)]$ containing each unique word w_i and its corresponding count v_i. So far, this is good old functional programming; in a distributed environment, however, execution and data are partitioned amongst many nodes. This is illustrated and described in Figure 3.1, where `count` is applied to a list of words $[w_1, \ldots, w_n]$. Notice how the *shuffle phase* routes key-value pairs, of the form $(w_i, 1)$, to their respective reducers. In this process, values of a given key w_i may arrive out of order. In a sense, the reducer views the list as an *unordered collection*, and therefore may produce different results depending on the order in which it applies the binary reduce function `r`.

To ensure that the reducer produces the same value regardless of the shuffle phase, we must ensure that the binary function passed to the reducer—in this example, `r a b = a + b`—is associative, commutative, and closed on the type the reducer operates on. Indeed, this is what the APIs for Apache Spark (Spark) and Twitter Summingbird (Boykin et al., 2014) expect. Commutativity and associativity ensure determinism *despite* the

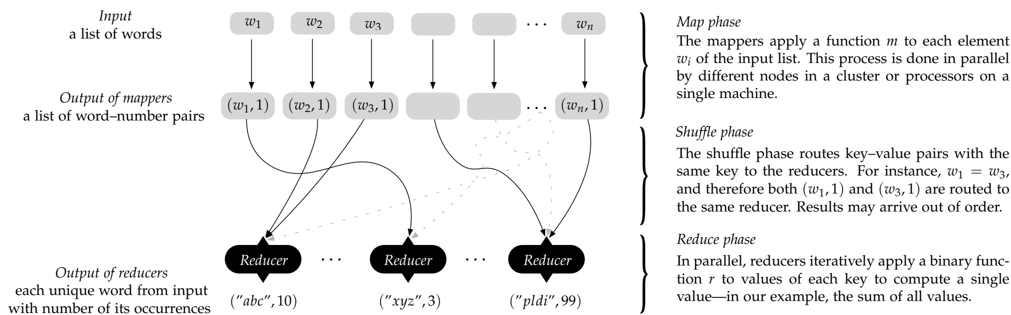


Figure 3.1: High-level view of a MapReduce computation on a simple example

shuffle phase. They also allow the runtime environment to apply `r` in parallel and at the mappers before transferring results to the reducers, in order to reduce the transferred data bottleneck.

We presented a simple data-parallel program: a mapper followed by a reducer. In many modern frameworks, e.g., Spark and Dryad, we can have more sophisticated combinations of mappers and reducers (e.g., iterative MapReduce) and various forms of data-parallel operations. Here, we will focus on programs made of arbitrary compositions of data-parallel operations presented as higher-order sketches.

Wordcount: The Fibonacci of MapReduce

Suppose you want to compute the number of times each word appears in Wikipedia. With many gigabytes of articles, the only way to do this efficiently is via distribution. To synthesize this task, you can supply our algorithm with a fairly simple example describing word counting, e.g.:

```
["hello bucky", "hello badgers"]
    ↪ [("hello",2),("bucky",1),("badgers",1)]
```

where the left side of $\hookrightarrow$ is the input (two strings representing simple documents) and the right side is the intended output.

The fascinating bit is that, even with a very simple example that can fit in one line, we can synthesize a word-counting program that can easily scale to *gigabytes of documents*. Specifically, our technique, when instantiated with appropriate constants, synthesizes the following:

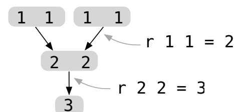
```
let wc = flatMap ms ∘ map mp ∘ reduceByKey r
where
    ms doc = split doc " "
    mp word = (word,1)
    r c1 c2 = c1 + c2
```

The synthesized program is a composition of three data-parallel operations: (*i*) a `flatMap` that *maps* each document into the list of words appearing in it (using `split`), and *flattens* (concatenates) all resulting lists into a single list, (*ii*) a `map` that transforms each word `w` into the string-integer pair `(w,1)`, and (*iii*) a `reduceByKey` that computes a count of occurrences of each word.

Note that for our input-output example, the following argument to `reduceByKey` would suffice:

```
r c1 c2 = c1 + 1
```

However, this will be rejected by our algorithm, since this reduce function does not form a commutative semigroup over integers. Specifically, using this function results in a non-deterministic program that may produce incorrect results for larger inputs. Suppose, for instance, that our input has four occurrences of `"hello"`. Then, for the key `"hello"`, the reducer would receive the list $[1, 1, 1, 1]$. Applying the binary function `r` in parallel (or as a combiner) could yield the wrong results, e.g., by applying `r` as follows:



Our algorithm ensures that synthesized programs are deterministic, despite the shuffle phase and parallel applications of binary reduce functions

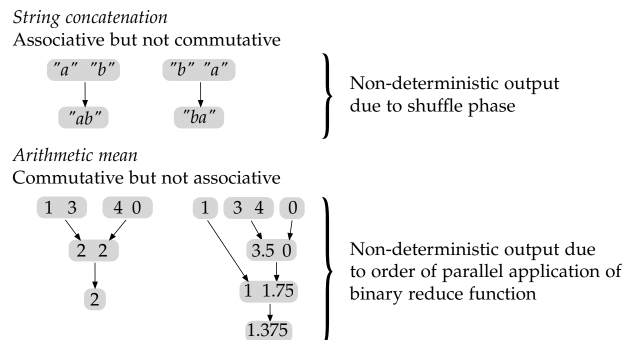


Figure 3.2: Non-associative/commutative binary reduce functions

(see Section 3.4). Figure 3.2 provides two additional examples to illustrate the effects of non-commutative or non-associative reduce functions.

Histograms

Now, suppose that you would like to plot a histogram of the page views of Wikipedia articles (available in Wikipedia log dumps (Wikipedia)) using three bins: less than 100 views, 100-10,000 views, and greater than 10,000 views. To construct a histogram, we need a procedure that computes the number of articles in each bin. We can supply the following example:

```
[("pg1", 99),("pg2",20000),("pg3",200),("pg4",300)]
   ↪ [(bin1,1), (bin2,2), (bin3,1)]
```

The inputs specify a set of pages and their views; the outputs specify each of the three bins in the histogram (< 100, 100-10,000, and $> 10{,}000$) as `bin1`, `bin2`, and `bin3`.

Here, our technique would synthesize the following:

```
let hist = map m ∘ reduceByKey r
where
   m p = if (snd p) < 100 then (bin1, 1)
          else if (snd p) < 10000 then (bin2, 1)
          else (bin3, 1)
   r c1 c2 = c1 + c2
```

Component name : type description
`map` : $(\alpha \to \beta) \to \texttt{mset}\,[\alpha] \to \texttt{mset}\,[\beta]$ *Applies a function* `f` *in parallel to each element in a multiset, producing a new multiset.*
`flatMap` : $(\alpha \to \texttt{mset}\,[\beta]) \to \texttt{mset}\,[\alpha] \to \texttt{mset}\,[\beta]$ *Applies a function* `f` *(that produces a multiset) to each element in a multiset and returns union of all multisets.*
`reduce` : $(\alpha \to \alpha \to \alpha) \to \texttt{mset}\,[\alpha] \to \alpha$ *Continuously applies a binary function* `f` *in parallel to pairs of elements in a multiset, producing a single element as a result.*
`reduceByKey` : $(\alpha \to \alpha \to \alpha) \to \texttt{mset}\,[(\beta, \alpha)] \to \texttt{mset}\,[(\beta, \alpha)]$ *Similar to* `reduce`*, but applies the binary function* `f` *to the multiset of values of a given key, resulting in a multiset of key–value pairs, with one value per key.*
`filter` : $(\alpha \to \texttt{bool}) \to \texttt{mset}\,[\alpha] \to \texttt{mset}\,[\alpha]$ *In parallel, removes elements of multiset that do not satisfy a Boolean predicate.*

Table 3.1: Set of data-parallel components C_{DP} from Apache Spark (variables α and β are implicitly universally quantified)

where `snd` returns the second element of a pair, and `bin1`, `bin2`, and `bin3` are values of an enumerated type. Observe that `map` places each page in the appropriate bin and `reduceByKey` counts the number of page in each bin.

3.2 Higher-Order Sketches and the Synthesis Domain

The language in which we synthesize programs is a restricted, typed λ-calculus parameterized by a set of components, or predefined functions. We first fix an ML-like type system. Let $\iota_1, \iota_2, \ldots$ be countably-many *base types*, and let $\alpha_1, \alpha_2, \ldots$ be countably-many *type variables*. Then a type can

be either a *monotype* or a *polytype*:

$$
\begin{aligned}
\text{monotype } \tau &:= \iota && \text{(base type)}\\
&\mid \alpha && \text{(type variable)}\\
&\mid \tau_1 \to \tau_2 && \text{(function construction)}\\
&\mid \tau_1 \times \tau_2 && \text{(product construction)}\\
&\mid \texttt{mset}\,[\tau] && \text{(multiset construction)}\\
\text{polytype } \sigma &:= \forall \vec{\alpha}.\, \tau && \text{(polymorphic construction)}
\end{aligned}
$$

We use C to denote a set of *components*—user-provided functions annotated with monotypes—and X to denote a countable set of *program variables*. A *program* is a p-sorted object in the Σ^C-term algebra, where (S, Σ^C) is the signature (Definition 2.7) given by:

$$
\begin{aligned}
S &:= \{p\} && \text{(the program-sort)}\\
\Sigma^C &:= \{x : \epsilon \to p \mid x \in X\} && \text{(variables as programs)}\\
&\sqcup \{c : \epsilon \to p \mid c \in C\} && \text{(components)}\\
&\sqcup \{\texttt{abs}_x : p \to p \mid x \in X\} && \text{(abstraction)}\\
&\sqcup \{\texttt{app} : pp \to p\} && \text{(application)}
\end{aligned}
$$

Note the treatment of abstraction and application as *function symbols*. When clear, we will write $\lambda x.\, p$ instead of $\texttt{abs}_x(p)$ and $p(p)$ (or even $p(p, p, \ldots, p)$ instead of $\texttt{app}(p, p)$ (and $\texttt{app}(p, \texttt{app}(p, \texttt{app}(\ldots, p))))$. Lastly, as is common for higher-order data-parallel operators, we will use $\circ$ to denote *reverse composition*: $(g \circ f)(x) \equiv f(g(x))$.

Typing Rules

As our type system is the classic Hindley-Milner type system with function, product, and multiset type constructors, we will elide the usual inference

rules. For typing components from C, we defer to the type annotations provided by the user. The usual judgement $\Gamma \vdash p : \tau$ has the expected meaning: given the assumptions in type context Γ, program p can be given type τ. We will also use:

Definition 3.1. *Let $p \in T_{\Sigma^C}(W)$. We will use the judgement $p \in \tau^\Gamma$ to indicate that there exists a substitution σ from type variables to variable-free monotypes such that $\sigma\Gamma \vdash p : \sigma\tau$.*

Canonical Algebra

Let $HM(C)$ be the Hindley-Milner λ-calculus with (*i*) encodings for all base types $\iota_1, \iota_2, \ldots$, (*ii*) function, product, and multiset type constructors, and (*iii*) λ-term implementations for each component $c \in C$ consistent with the base type encodings. Then $HM(C)$ is a Σ^C-algebra, with the carrier set containing all λ-terms, and the interpretation that maps each function symbol in Σ^C to the appropriate λ-term constructor.

Now, let P be *any* programming language capable of emulating $HM(C)$, and treat P as a *super-algebra* of Σ^C. By emulation, there exists an algebra homomorphism that embeds λ-terms into P. We will treat P as our *canonical algebra* for the rest of this work: by Theorem 2.12, there is a unique homomorphism that embeds T_{Σ^C} terms into P, which must exactly capture the semantics of $HM(C)$.

Higher-Order Sketches and Data-Parallel Components

A *higher-order sketch* (HOS) is an *incomplete* (Definition 2.16) and well-typed program. For practical purposes, a HOS will typically be a composition of *data-parallel components*, such as `map` and `reduceByKey`. To make this formal, we restrict HOSs to be an incomplete program built from components in C_{DP}, where C_{DP} is a set of data-parallel components.

We curated C_{DP} using data-parallel components that mimic the primary operations offered by Apache Spark (Spark). C_{DP} components are described and exemplified in Table 3.1. Note that our restricted language does not exploit advanced cluster-programming features needed to maximize performance for complex workloads.

An important point to make here is that Spark operates over *Resilient Distributed Datasets* (RDDs) (Zaharia et al., 2012), a data abstraction that represents a collection of elements partitioned and replicated amongst various nodes in a cluster. This representation is incredibly important for the scalability of systems like Spark; however, for the purpose of program synthesis, it suffices to model an RDD of elements of a given type τ as a multiset of τ, written $\texttt{mset}[\tau]$.

Data-Parallel Synthesis Tasks

A data-parallel synthesis task S is given as a triple (E, C, H), where:

1. $E = \{(i_i, i_i)\}_{i=1}^{n}$ is a finite set of *input-output examples*: pairs of values in the canonical algebra such that each i_i (respectively, each o_i) has the same type.

2. C is a set of components. We assume all functions $f \in C$ are provided with an interpretation that is terminating and referentially transparent.

3. H is a set of HOSs over C_{DP}.

Each synthesis task S defines a *correctness constraint* (Definition 2.17), denoted ϕ_S, as follows:

Definition 3.2 (Solutions to Correctness Constraint). *Let* $S = (E, C, H)$ *be a synthesis task, and let* ϕ_S *be the induced correctness constraint. A program* p *(e.g., a* p*-sorted term in the* $\Sigma^{C \cup C_{DP}}$*-term algebra) is a* solution *to* ϕ_S*, denoted* $p \models \phi_S$*, if and only if:*

1. *There is an* $h \in H$ *such that* p *is a* completion *of* h—*that is, there exists an assignment from wildcards to terms* $\nu : W \to T_{\Sigma^C}$ *such that* $\nu^*(h) = p$.

2. $\forall (i, o) \in E, [\![p]\!]\,(i) \to^* o$, *where* $[\![\cdot]\!]$ *is the interpretation function of the canonical algebra induced by* C, *and* $\to^*$ *denotes finitely-many evaluation steps.*

3. *The program* p *is deterministic, regardless of how* `reduce` *and* `reduceByKey` *operate (see Section 3.4).*

Intuitively, a solution to a synthesis task is a deterministic program p that, when applied to any input example i_i, produces the corresponding output example o_i. Further, p is a completion of one of the HOSs in H.

3.3 Compositional Synthesis Algorithm

Given a synthesis task $S = (E, C, H)$, our goal is to *complete* on of the HOSs in H such that the result is a solution of S. For practical purposes, we assume input-output examples in E are monotyped and variable-free.

To compute a solution of S, our algorithm employs two cooperating phases—synthesis and composition—that act as producers and consumers, respectively.

1. *Synthesis Phase (Producers)*: Initially, the algorithm infers the type of terms that may appear for each wildcard in H. For instance, it may infer that $\bullet$ needs to be replaced by a term of type `int` $\to$ `int`. Thus, for each inferred type τ, the synthesis phase will *produce* terms of type τ.

2. *Composition Phase (Consumers)*: For each $h \in H$, the composition phase attempts to find an assignment $\nu : W \to T_{\Sigma^C}$ from wildcards to complete programs such that $\nu^*(h)$ is a solution of S. To construct the map ν, this phase *consumes* results produced by the synthesis phase.

To implement the two phases, the algorithm maintains two data structures: (*i*) M, a map from *types and typing contexts* to sets of (potentially

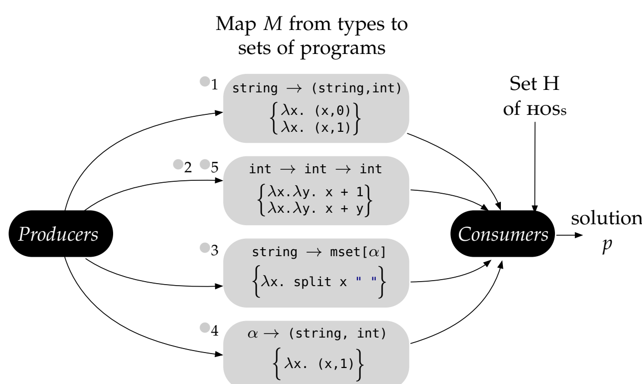


Figure 3.3: High-level illustration of BIGλ synthesis algorithm

incomplete) programs of the given type, and (*ii*) C, a set of complete, well-typed programs that are *candidate* solutions to the synthesis task. Informally, the synthesis phase populates M with programs of inferred types, while the composition phase scavenges M to construct candidate solutions and insert them in C. This algorithm is best illustrated through an example.

A Walkthrough of Compositional Synthesis

Suppose our goal is to synthesize the wordcount example from Section 3.1, and that we have the following two HOSs, written as program terms:

```
let h_1 i = (map ●1 ∘ reduceByKey ●2) i
let h_2 j = (flatMap ●3 ∘ map ●4 ∘ reduceByKey ●5)
```

Assume we've been given a set of examples E, where the inputs have type `mset` [`int`] and the outputs have type `mset` [(`string`, `int`)]. The synthesis algorithm determines the type of programs that need to be synthesized

for the various wildcards $\bullet_i$. Specifically, it will determine that:

$\bullet_1 : \texttt{string} \to (\texttt{string}, \texttt{int})$

$\bullet_2 : \texttt{int} \to \texttt{int} \to \texttt{int}$

$\bullet_3 : \texttt{string} \to \texttt{mset}\,[\alpha]$

$\bullet_4 : \alpha \to (\texttt{string}, \texttt{int})$

$\bullet_5 : \texttt{int} \to \texttt{int} \to \texttt{int}$

Note that, for $\bullet_4$, we are looking for programs of type $\tau \to (\texttt{string}, \texttt{int})$, where τ is any variable-free monotype: we know that $\bullet_4$ should be replaced by a function that returns a string-integer pair, but as we do not know what type of argument it should take we need to consider all possibilities.

The algorithm detects that $\bullet_5$ is the same type as $\bullet_2$, and thus will create one entry in M for *both* wildcards (although we might have to rename variables in the HOSs to achieve the same typing context). This ensure we do not duplicate work for wildcards of the same type, even if they appear in different HOSs.

Figure 3.3 shows the map M, where each key corresponds to the inferred type of one or more of the wildcards in the HOSs. Each value in M is a set of programs of a given type. For instance, we see that for $\texttt{int} \to \texttt{int} \to \texttt{int}$, M contains two programs.

Producers populate each set $M(\tau^\Gamma)$ with programs in τ^Γ (Γ is the typing context, described later). Consumers query M with the goal of replacing the wildcards in H with complete programs. For instance, consumers might complete the HOS h_2 as follows, using programs from appropriate locations in M to fill the wildcards $\bullet_{\{3,4,5\}}$:

```
●3 ← fun x -> split x " "
●4 ← fun x -> (x, 1)
●5 ← fun x -> fun y -> x + y
```

This results in the same program we saw in Section 3.1, which is a solution to the wordcount task.

The Algorithm

The algorithm is presented in Figure 3.4 as a set of inference rules that update M and C. The algorithm uses the rules (Init) and (InitM) to initialize the map M as follows: for each wildcard $\bullet$ appearing in a HOS $h \in H$, the algorithm infers a type τ for $\bullet$, along with a typing context Γ. The typing context contains all $f \in C$, as well as all variables in the scope at $\bullet$. For example, consider the HOS $h : \lambda x. \, (\texttt{map} \bullet) \; i$, and suppose that our input-output examples are both of type $\texttt{mset}\,[\texttt{int}]$. Then, the function $\textsc{Infer}\,(\bullet, h)$ detects that $\bullet$ must have the type $\texttt{int} \to \texttt{int}$, and that the variable i of type $\texttt{mset}\,[\texttt{int}]$ is in scope. Note that $\textsc{Infer}\,(\cdot, \cdot)$ can be implemented using standard Hindley-Milner type inference.

Synthesis Phase

The synthesis rules — (PVar), (PApp), and (PAbs) — construct programs of a given type τ under context Γ. This is a *top-down* synthesis process: it starts with an incomplete program and gradually replaces its wildcards with complete terms. Being type-directed, synthesis rules maintain the invariant that, for all $p \in M(\tau^{\Gamma})$, $p \in \tau^{\Gamma}$. As we shall see, these rules can synthesize every possible complete program for a given type and context.

Rule (PVar) replaces a wildcard $\bullet$ in some program p with a variable that is in scope at the location of $\bullet$. For instance, suppose p is the program $\lambda x. \, f(\bullet)$, then (PVar) may replace $\bullet$ with x, or another variable that is in scope. We use the auxiliary function $\textsc{InScope}\,(\bullet, p)$ to denote the set of variables in scope at $\bullet$ in p (which include variables in context Γ). Rule (PApp) replaces a wildcard with a function application f from components Σ. The arguments of f are fresh wildcards. Finally, the rule (PAbs) introduces a λ-abstraction.

Example 3.3. *Suppose that we wanted to synthesize a program of type* $\texttt{int} \to \texttt{int}$ *and that the program* $p \coloneqq \lambda x. \bullet$ *is in* $M(\tau^{\Gamma})$*. Then, using* p*, (*PVar*) can*

construct $q \coloneqq \lambda x.\, x$*, which can be of the desired type* $\mathtt{int} \rightarrow \mathtt{int}$*.*

Example 3.4. *Suppose that we want a program of type* $\tau_1 \rightarrow \tau_2 \rightarrow \tau_3$*. Suppose also that* $p \coloneqq \lambda x.\, \bullet$ *is in* $M(\tau^{\Gamma})$*. Then, (*PAbs*) can construct a new program* $q \coloneqq \lambda x.\, \lambda y.\, \bullet$ *from* p *by adding an additional* λ*-abstraction. Now, to complete* q*, we need to replace* $\bullet$ *with a term of type* τ_3*.*

Composition Phase

This phase composes programs in M to synthesize a program p that is a solution to the synthesis task. We use two rules to define this phase. First, for a HOS $h \in H$, the rule (Cons) attempts to find a completion of h by finding a program $p \in M(\tau^{\Gamma})$ for each wildcard of type τ and context Γ in h. If this results in a program that is consistent with the type $\tau_I \rightarrow \tau_O$ (the type of input-output examples), then we consider it a candidate solution and add it to the set C.

The rule (Verify) picks a candidate program p from C and checks that (*i*) $p \models E$ and (*ii*) p is *deterministic*, using the function Determ $(\cdot)$. If the rule applies, then p is a solution to the synthesis task (Definition 3.2). For this section, we assume that Determ $(\cdot)$ is an *oracle* that determines whether, for every input, the program produces the same output for any order of application of the binary reduce functions in `reduce` and `reduceByKey`, if used in p. In Section 3.4, we present a sound implementation of Determ $(\cdot)$.

Soundness and Completeness

The following theorem states that the algorithm is sound.

Theorem 3.5 (Soundness)**.** *Given a synthesis task* $S = (E, \Sigma, H)$*, if the synthesis algorithm returns a program* p*, then* p *is a solution to* S *(as per Definition 3.2).*

Initialization rules

$$\frac{}{M \leftarrow \emptyset \qquad C \leftarrow \emptyset}\ (\textsc{Init})$$

$$\frac{h \in H \qquad \tau, \Gamma = \textsc{Infer}\,(\bullet, h) \qquad \tau^{\Gamma} \notin \mathrm{Dom}(M)}{M(\tau^{\Gamma}) \leftarrow \{\bullet\}}\ (\textsc{InitM})$$

Synthesis phase *(producers)*

$$\frac{p\,[\bullet] \in M(\tau^{\Gamma}) \qquad x \in \mathrm{Scope}(\bullet, p) \qquad p\,[x] \in \tau^{\Gamma}}{M(\tau^{\Gamma}) \leftarrow M(\tau^{\Gamma}) \cup p\,[x]}\ (\textsc{PVar})$$

$$\frac{\begin{array}{c} p\,[\bullet] \in M(\tau^{\Gamma}) \qquad c : \tau_1 \to \cdots \to \tau_n \to \tau \in C \\ p\,[c(\bullet_1, \ldots, \bullet_n)] \in \tau^{\Gamma} \qquad \{\bullet_i\}_i \text{ fresh} \end{array}}{M(\tau^{\Gamma}) \leftarrow M(\tau^{\Gamma}) \cup p\,[c(\bullet_1, \ldots, \bullet_n)]}\ (\textsc{PApp})$$

$$\frac{p\,[\bullet] \in M(\tau^{\Gamma}) \qquad p\,[\lambda v.\, \bullet'] \in \tau^{\Gamma} \qquad \bullet' \text{ and } v \text{ are fresh}}{M(\tau^{\Gamma}) \leftarrow M(\tau^{\Gamma}) \cup p\,[\lambda v.\, \bullet']}\ (\textsc{PAbs})$$

Composition phase *(consumers)*

$$\frac{h \in H \quad v : W \to T_{\Sigma} \quad \forall \bullet \in h.\, \tau, \Gamma = \textsc{Infer}\,(\bullet, h) \Rightarrow v(\bullet) \in M(\tau^{\Gamma})}{C \leftarrow C \cup v^{*}(h)}\ (\textsc{Cons})$$

$$\frac{p \in C \quad p \models E \quad \textsc{Determ}\,(p) \quad p : \tau_I \to \tau_O}{p \text{ is a solution to synthesis task}}\ (\textsc{Verify})$$

Figure 3.4: Inference rules defining the Bigλ synthesis algorithm

The algorithm, as presented, is non-deterministic. To ensure completeness, we need to impose a notion of fairness on rule application. A *fair schedule* is an infinite sequence of rules $c_1, c_2, \ldots$, where if at any point i in the sequence some rule c is applicable on some set of parameters, and c has not yet been applied to those parameters, then c applied on those parameters eventually appears in the sequence. A fair execution is an application of rules under a fair schedule. The following theorem states completeness of the algorithm, relative to existence of an oracle $\textsc{Determ}(\cdot)$ and existence of a solution.

Theorem 3.6 (Relative completeness)**.** *Given a task $S = (E, \Sigma, H)$ with a solution, a fair execution will find some solution p of S in finitely-many rule applications.*

Determinization and Optimality

Smaller programs are desirable in inductive synthesis, as they are considered more likely to generalize (Albarghouthi et al., 2013; Osera and Zdancewic, 2015; Feser et al., 2015). Given a per-component weight function $w : C \to \mathbb{N}$ and a constant $k > 0$, we can define a Σ^C-algebra where sort $p = \mathbb{N}$ and:

$$[\![\bullet]\!] \coloneqq 0 \qquad [\![x]\!] \coloneqq k \text{ (for } x \in X) \qquad [\![c]\!] \coloneqq w(c) \text{ (for } c \in C)$$
$$[\![\mathtt{abs}_x]\!]\,(w) \coloneqq k + w \qquad [\![\mathtt{app}]\!]\,(w_1, w_2) \coloneqq w_1 + w_2$$

An *optimal execution* is a fair execution where (*i*) synthesis rules produce programs in M in order of increasing weight, and (*ii*) composition rules compose candidate solutions in C and check them in order of increasing size. Finding an optimal execution is made feasible by the fact that the weight algebra operates additively. In Section 3.5, we describe how we practically implement an optimal schedule.

The proofs of Theorems 3.5 and 3.6 are given in Appendix A.1.

3.4 Commutative Semigroup Reducers

We will now provide a sound implementation of the oracle DETERM $(\cdot)$ used in Section 3.3. To ensure that synthesized programs are deterministic, a sufficient condition is that each binary function $r : \tau \to \tau \to \tau$, synthesized as an argument to `reduce` or `reduceByKey`, forms a *commutative semigroup* over τ.

Definition 3.7 (Commutative semigroup (CSG))**.** *A* semigroup *is a pair* $(S, \otimes)$*, where* S *is set of elements,* $\otimes : S \times S \to S$ *is an* associative *binary operator over elements of* S*, and* S *is closed with respect to* $\otimes$*. A* commutative semigroup *(*CSG*) is a semigroup* $(S, \otimes)$ *where* $\otimes$ *is also commutative.*

We say that $\otimes$ *forms a* CSG *over* S *if* $(S, \otimes)$ *is a* CSG*.*

Note that this condition is sufficient, but not necessary. To see why, consider the following function:

```
let r x y = max (abs x) y
```

This is not a commutative function, as `r -3 2 = 3` but `r 2 -3 = 2`. However, suppose we know that r will only ever operate on positive integers, perhaps as an artifact of the mapper, and always returns a positive integer. One can show that r forms a CSG over positive integers, and can thus operate deterministically in a distributed environment.

To check the necessary conditions, we would need a fine-grained type for the reducer—*refinement types* (Freeman and Pfenning, 1991), for instance—that specifies the range of values on which it is invoked. While this would enable us to encode that r only operates over positive integers, fine-grained types necessitate the use of a heavyweight type system and the ability to reason about *all* operations of the synthesized program, not just reducers. In our experience, this is unnecessary.

We employ a two-tiered strategy to prove that a reducer forms a CSG:

1. *Dynamic Analysis*: First, using the input-output examples, we run the synthesized program simulating every possible shuffle and order of applications of binary reduce functions. Any reduce functions that fail to form a CSG are *rejected*.

2. *Static Analysis*: If our dynamic analysis finds no evidence that a reduce function does not form a CSG, we apply a verification phase that checks whether a reduce function is a CSG by encoding it as a first-order SMT formula.

Even though our dynamic analysis might explore exponentially-many executions, in practice we are given a small enough set of examples that exploring all possible evaluation orderings is feasible. The aggressive filtering the dynamic analysis provides dramatically reduces the number of times we apply our static analysis, which requires expensive satisfiability checking in an external SMT solver.

Hyperproperty Verification Condition

Commutativity and associativity are considered *hyperproperties* (Clarkson and Schneider, 2010): they require reasoning about multiple executions of a function. Specifically, commutativity is a *2-safety property*, as it requires two executions, while associativity is a *4-safety property*. We will exploit this fact to encode the static analysis portion of our CSG check into a single verification problem using the *self-composition* technique (Zaks and Pnueli, 2008; Barthe et al., 2004).

We encode a binary reduce function r as a ternary relation $R(i_1, i_2, o)$, where i_1 and i_2 represent the parameters of r, and o represents its return value. Then we know that r forms a CSG over its input type if and only if the following formula VC_R is valid:

$$VC_R \coloneqq \forall V.\ \varphi_{com} \wedge \varphi_{assoc} \Rightarrow \psi_{CSG}$$

where

$$\begin{aligned}
\varphi_{com} &\coloneqq R(i_1, i_2, o_1) \wedge R(i_2, i_1, o_2) \\
\varphi_{assoc} &\coloneqq R(o_1, i_3, o_3) \wedge R(i_2, i_3, o_4) \wedge R(i_1, o_4, o_5) \\
\psi_{CSG} &\coloneqq o_1 = o_2, \wedge o_3 = o_5 \\
V &= \{i_1, i_2, i_3, o_1, \dots, o_5\}
\end{aligned}$$

The formula φ_{com} encodes two executions of r with flipped arguments, i_1 and i_2, for checking commutativity. Formula φ_{assoc} encodes three executions of r, despite associativity being a 4-safety property, by reusing one of the executions in φ_{com}. Finally, ψ_{CSG} encodes the correctness condition for r to form a CSG.

Theorem 3.8 (VC correctness)**.** *Given a binary function* $r : \tau \to \tau \to \tau$ *and its encoding* R *as a ternary relation, then* (τ, r) *is a* CSG *if and only if* VC_R *is valid.*

The proof of Theorem 3.8 is given in Appendix A.2.

To finish implementing our analysis, we need to convert a binary function r into the corresponding ternary relation R. Since r is binary, it is of the form $\lambda i_1. \lambda i_2. e$, where e is a synthesized term. We make the simplifying assumption that e uses no higher-order components. As is standard (Suter et al., 2011; Kneuss et al., 2013; Jha et al., 2010; Gulwani, 2011), we assume that each component $c \in C$ has a corresponding encoding $R_c(a_1, \dots, a_n, o)$ provided by an expert. We will encode r as $R(i_1, i_2, o)$ using the function $\textsc{Enc}_o(e)$, defined below, in a manner analogous to other encodings of

functional and imperative programs (Suter et al., 2011; Kneuss et al., 2013):

$$\begin{aligned}\textsc{Enc}_o\left(i_i\right) &\coloneqq \left(o = i_i\right)\\ \textsc{Enc}_o\left(f\right) &\coloneqq R_f(o)\\ \textsc{Enc}_o\left(f\left(p_1, \ldots, p_n\right)\right) &\coloneqq R_f(a_1, \ldots, a_n, o) \wedge \\ &\quad \bigwedge_{i=1}^{n} \textsc{Enc}_{o_i}\left(p_i\right) \wedge a_i = o_i\end{aligned}$$

where $\{a_1, \ldots, a_n, o\}$ are fresh variables, constructed uniquely in every recursive call to $\textsc{Enc}\left(\cdot\right)$. All variables other than i_1, i_2 and the top-most o are implicitly existentially quantified.

Example 3.9. *The algorithm* $\textsc{Enc}\left(\cdot\right)$ *traverses a program* p *recursively, constructing a logical representation* R_f *for each sub-term* f*. Consider, for example, the following binary reduce function:*

```
let r i_1 i_2 = max i_1 i_2
```

We use $\textsc{Enc}_o\left(\mathit{max}(i_1, i_2)\right)$ *to construct the logical representation of this function. Here, the third case of* $\textsc{Enc}\left(\cdot\right)$ *matches and we get the following relation over the variables* i_1, i_2*, and* o*:*

$$\exists a_1, a_2, o_1, o_2.\, R_{\mathit{max}}(a_1, a_2, o) \wedge \bigwedge_{i \in \{1,2\}} o_i = i_i \wedge a_i = o_i$$

where

$$R_{\mathit{max}}(a_1, a_2, o) \equiv \left(a_1 > a_2 \Rightarrow o = a_1\right) \wedge \left(a_1 \leqslant a_2 \Rightarrow o = a_2\right)$$

Observe that the above formula can only be satisfied if o *is set to the value of the larger of* i_1 *or* i_2*.*

3.5 Implementation and Evaluation

We implemented our algorithm in a modular tool we call BIGλ. Components in BIGλ are represented as *annotated* functions in a separate extensible library. These annotations provide typing information and a logical encoding of each component. Producers generate an infinite list of programs in increasing weight order, for each type in the map M, while consumers lazily combine these programs with the appropriate HOSs. Each producer and consumer runs in a separate process, with one producer process per key of M and one consumer process per HOS. Candidate solutions are checked for determinism by a separate CSG checker, which invokes the Z3 SMT solver (de Moura and Bjørner, 2008). Synthesized programs are converted into Apache Spark code and are ready to be executed on an appropriate platform.

Optimal execution We ensure that BIGλ always generates an optimal program with respect to the weight function ω. Producers generate infinitely many programs in increasing weight order; by exploiting additivity of our weight function, consumers can efficiently explore the Cartesian products of these infinite lists in increasing weight order. If a consumer produces a solution p, we are guaranteed that p is an optimal solution (with respect to that consumer). In practice, we have multiple consumers; when the first consumer reports a solution p of weight w, we continue executing all other consumers until they produce a solution p' of weight $w' < w$ or a candidate solution p' of weight $w' \geqslant w$.

To prevent producers from getting lost down expansions of irrelevant types, we start with uniform weights and automatically inject a bias towards components over types present in the given examples.

Type checking BIGλ employs incremental type inference, where sets of typing constraints are maintained with each program. Since producers do

Component name	Description
general	
`pair` : $\alpha \to \beta \to (\alpha, \beta)$	create pair
`cons` : $\alpha \to$ `mset`$[\alpha] \to$ `mset`$[\alpha]$	add element to a multiset
`emit` : $\alpha \to$ `mset`$[\alpha]$	create singleton multiset
arithmetic	
`one` : `int`	integer constant 1
`add` : `int` → `int` → `int`	integer addition
`eq?` : `int` → `int` → `Bool`	check two ints for equality
`mult` : `int` → `int` → `int`	integer multiplication
`max` : `int` → `int` → `int`	return maximal integer
`factors` : `int` → `mset[int]`	return list of factors of int
`div` : `int` → `int` → `float`	integer division to float
`round` : `float` → `int`	round float to int
string	
`pattern` : `string` → `Bool`	string selector (e.g. regex)
`chars` : `string` → `mset[string]`	convert to list of chars
`split` : `string` → `mset[string]`	split text by whitespace
`lower` : `string` → `string`	convert to lowercase
`len` : `string` → `int`	get length of string
`order` : `string` → `string`	orders the chars of a string
data-based	
`hashtag` : `string` → `Bool`	regex selecting hashtags
`canonical` : $(\alpha, \alpha) \to$ `Bool`	checks if `left` $\leqslant$ `right`
`get_tag` : `Json` → `string` → `Json`	get value of tag in JSON file
`find_tags` : `Json` → `mset[string]`	get top-level tags in JSON file
`gen_perms` : `mset`$[\alpha] \to$ `mset`$[(\alpha, \alpha)]$	convert multiset into all pairs

Table 3.2: A sample of components used by BIGλ

not communicate during synthesis, different wildcards with the same type variables might specialize to different variable-free monotypes. In order to resolve these inconsistencies, producers keep track of constraints over type variables as they generate programs. The consumers then ensure that the intersection of the constraints are satisfiable before producing a candidate solution.

	Set:task	Wall time	CPU time	AST size	|E|	# of cand.	WL time
General MapReduce tasks	*Strings*						
	anagram	2.1	10.7	17	1	911	✗
	dateextract	0.2	1.2	13	1	78	0.4
	grep	0.6	4.2	14	2	593	33.2
	histogram	0.1	0.5	12	2	34	0.8
	postagging	0.2	1.0	16	2	154	3.1
	letteranalysis	4.4	25.1	16	2	6978	✗
	wordcount	0.2	1.0	15	1	146	2.1
	Numerical						
	factors	0.4	2.8	15	2	623	35.0
	max	0.4	2.2	9	3	551	0.8
	min	0.4	2.1	10	3	392	0.8
	roundedsum	0.6	4.1	11	2	1047	2.2
	squaredsum	0.8	5.0	10	2	1728	2.9
	sum	0.1	0.6	9	2	72	0.3
	sumoffactors	0.1	0.6	10	2	64	0.3
	sumrounded	2.9	14.0	13	2	8734	36.8
	sumsquared	.3	15.3	13	3	10822	58.6
	Databases						
	union	0.1	0.5	10	1	30	1.3
	selection	0.3	2.2	17	1	476	✗
	join	1.3	7.6	18	1	380	✗
Data analysis tasks	*Cycling*						
	bpm	4.3	22.0	14	1	1287	✗
	watts	4.1	21.9	14	1	1327	✗
	speed	2.0	12.4	13	1	2323	37.1
	Twitter						
	hashtags	0.9	6.7	16	1	1361	50.1
	co-occurrence	0.4	2.7	17	1	419	✗
	Wikipedia						
	pageviews (log)	0.3	2.0	13	1	242	2.1
	bytes (log)	0.3	2.1	13	1	253	2.2
	filtered (dump)	12.8	53.6	20	1	22423	✗
	Shakespeare						
	characters	3.5	12.0	19	1	1319	✗
	sentiment	4.9	15.0	19	1	1450	✗
	Yelp						
	city	0.2	1.3	13	1	171	2.01
	state	0.2	1.2	13	1	157	1.99
	kids	0.1	0.5	13	1	34	0.24
	Enron						
	to	0.2	1.0	15	1	120	0.44
	from	0.6	4.6	15	1	995	4.02

Table 3.3: BIGλ synthesis task results

Synthesis Tasks

In order to test the effectiveness of BIGλ, we curated a set of synthesis tasks with data-analysis problems and general MapReduce programs (see

Table 3.3).

Data-analysis tasks We have collected a number of datasets, with unstructured and semi-structured data, on which we applied our approach to synthesize MapReduce programs that compute useful information.

Our datasets include a large set of tweets from Twitter that we collected via its streaming API (Twitter). We have synthesized programs that extract *hashtags* and compute their occurrence as well as their *co-occurrence* frequencies (which are often used in *topic modeling* (Blei et al., 2003)).

We also acquired a cycling dataset generated by a *bike computer*. The owner of this data (a cyclist and computer scientist) has used Apache Spark to perform a series of complex analyses (blo). We have used this dataset to synthesize programs that generate a number of histograms of interest to cyclists, e.g., amount of time spent in a speed range and maximum power output in ten-minute intervals.

Our datasets also include Shakespeare's full works, where, for example, we synthesized a program that detects and counts the number of lines said by each character in Shakespeare's plays. We also synthesized programs that analyzed Yelp reviews (Yelp), English Wikipedia dumps and log files (Wikipedia), and Enron emails (Cohen).

General MapReduce tasks These tasks represent the most common MapReduce tasks seen in tutorials and demonstrations, as well as tasks that can be parallelized in the MapReduce paradigm. In addition, we include (relational algebra) database operations—join, union, etc.—that are often compiled to MapReduce for application to large databases (Leskovec et al., 2014).

Components and sketches Each synthesis task uses a set of core components for common base types (such as integers, strings, pairs, lists) along with several higher-order components representing maps and filters. Each

task also has more domain-specific components for the input data. For example, when dealing with our Twitter dataset, we add components to handle the metadata and manipulate hashtags. Table 3.2 lists and describes a sample of the components appearing in our synthesis tasks.

For all tasks, we fix a set of eight HOSs with various compositions of the data-parallel operations in Table 3.1 and an average of 2-3 wildcards per sketch. These compositions are commonly used in Spark programs and represent most common MapReduce-like patterns (Miner and Shook, 2012).

Evaluation

We designed our experiments to investigate the following questions:

RQ1 How fast is the synthesis process?

RQ2 How many examples do we need for synthesis?

RQ3 Are the synthesized programs scalable?

To address these questions, we perform two sets of experiments. The first set involves synthesis of our collected tasks, which we conducted on a Linux machine with a 4-core Intel i7-4790k processor and 16GBs of memory. The second set of experiments takes the solution of synthesized tasks (in the form of executable Apache Spark code) and tests parallel scalability by applying them to gigabytes of data on Google Cloud clusters with `n1-standard-8` nodes (Cloud).

Table 3.3 describes the synthesis tasks we collected and results of applying BIGλ on these tasks. All tasks were successfully synthesized under a time limit of 90 seconds and a memory limit of 8GBs. For each task, the table shows (*i*) the amount of wall and CPU time (aggregate time over all cores) taken by BIGλ; (*ii*) the size of the synthesized programs (measured by AST nodes); (*iii*) the number of examples needed for generating a desired

solution; (*iv*) the number of candidate solutions examined for each task (applications of VERIFY $(\cdot)$); and (*v*) the amount of time taken by a worklist algorithm.

RQ1: Efficiency The results show that BIGλ can synthesize all tasks in a few seconds at most, with only a single benchmark exceeding 5 seconds. To demonstrate the difficulty of these benchmarks, we show runtime results for a (sequential) type-directed, top-down synthesis algorithm that maintains a *single worklist* that initially contains all HOSs. The algorithm, which we call WL, uses the worklist to explore all well-typed completions of the HOSs. This is analogous to a technique of Feser et al. (2015), but without the *deduce* step, which is inapplicable in our generic setting. The results show that BIGλ outperforms WL, which exceeds time limit in many instances. WL keeps a single worklist with a HOS h and partial completions for each $\bullet \in \mathit{wild}(h)$ as elements. Due to this, if h has two wildcards with n completions each, WL might require n^2 elements in the worklist. BIGλ breaks up h into two producers, one for each wildcard, both of which maintain a separate worklist of at most size n. By breaking h into subproblems, BIGλ turns a multiplicative cost into an additive one and saves on space and time.

RQ2: Usability Our results indicate that BIGλ can synthesize desired programs with a very small set of examples, despite the complex nature of the programs we synthesize (with solutions consisting of anywhere between 9 and 20 AST nodes). For example, BIGλ correctly synthesizes the following program, which computes hashtag co-occurrence patterns in tweets with only a single example multiset:

```
let hashtag_pairs = map m ∘ reduceByKey r
where
    m s = map
        (λp. (p, 1))
```

```
      (filter canonical (gen_perms (match "#[\w]" s)))
  r x y = x + y
```

Throughout our benchmarks, each example is relatively small, consisting of an input multiset of between 3 and 8 elements and an output value of approximately the same size. We checked correctness of synthesized programs manually against our own solutions (which we constructed in the process of collecting the tasks). We attribute the fact that a small number of examples is needed to (*i*) the restricted structure programs can take, as imposed by HOSs, and (*ii*) the optimality criterion that favors smaller programs.

RQ3: Scalability Our evaluation shows that restricting search to higher-order sketches resembling common data-parallel programming patterns indeed results in scalable implementations. For most tasks, synthesized programs closely resembled our own solutions. Figure 3.5 shows the time it took for three of our synthesized analyses to run on Twitter data, Wikipedia log files, and Wikipedia page dumps, respectively. The plots show the decreasing running time as we increase the number of available compute nodes, from 2 to 10, in our Google cloud cluster. All data sets are on the order of ~20GBs. We see an expected log-like increase in speedup as we increase the number of nodes (reducers need to apply a binary function $log\ n$ times on n items), indicating that our synthesized solutions are indeed data-parallel, and thus fit naturally on distributed architectures.

Summary In summary, our implementation and evaluation indicate our technique's ability to efficiently synthesize non-trivial data-parallel programs. Our evaluation also shows that, despite the rich language we have and the size of the data we wish to analyze, a small number of examples suffices for synthesizing powerful programs that can execute and scale on cloud infrastructure.

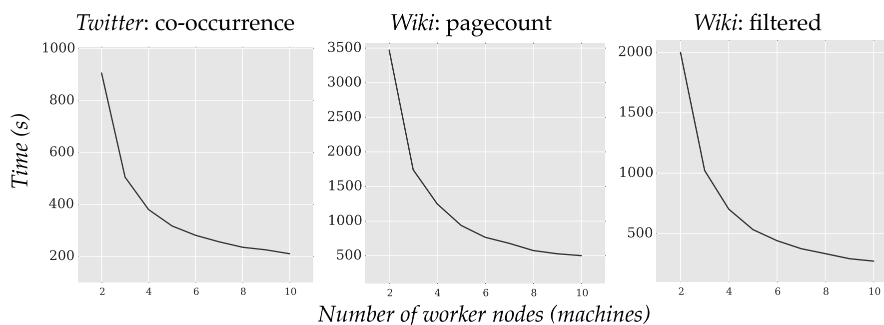


Figure 3.5: Scalability experimental results for Bigλ

3.6 Related Work

Functional program synthesis A number of works have addressed synthesis of functional programs (Summers, 1976; Albarghouthi et al., 2013; Feser et al., 2015; Osera and Zdancewic, 2015; Kneuss et al., 2013; Kuncak et al., 2010; Gvero et al., 2013; Kitzelmann and Schmid, 2006). The works of Feser et al. (2015), Osera and Zdancewic (2015), and Frankle et al. (2016), like our work, utilize both examples and types to search the space of programs. The works of Kneuss et al. (2013), Kuncak et al. (2010), and Polikarpova et al. (2016) synthesize functional programs from logical specifications or refinement types. Gvero et al. (2013) synthesize code snippets from types by enumerating all terms inhabiting a type (similar to what producers do in our algorithm). In comparison with these works, our work addresses the question of synthesizing functional programs that (*i*) utilize data-parallel operations and (*ii*) are robust to network non-determinism and reducer parallelization. Our work also introduces higher-order sketches to direct synthesis towards efficient, parallel implementations. Algorithmically, our work is inspired by the approaches of Escher (Albarghouthi et al., 2013), Myth (Osera and Zdancewic, 2015), and λ^2 (Feser et al., 2015).

Data transformation synthesis Gulwani's FlashFill (Gulwani, 2011) initiated a promising line of work on program synthesis for data manipulation by end users, particularly for spreadsheets. The work has been extended to string and number transformations (Singh and Gulwani, 2012b,a), table transformations (Harris and Gulwani, 2011), and data extraction from spreadsheets (Barowy et al., 2015; Le and Gulwani, 2014). The techniques have also been cast into a generic synthesis framework (Polozov and Gulwani, 2015).

The aforementioned works are primarily targeted at data extraction and transformation. Our work differs in two ways: (*i*) our primary goal is to synthesize programs that can run on large clusters; (*ii*) our work is also suited for data *aggregation* tasks—e.g., counting, compressing, building histograms—and not only data *transformation* tasks. We believe that combining our program synthesis technique with domain-specific data transformation synthesis, *data wrangling* (Kandel et al., 2011), and *query synthesis* (Tran et al., 2009; Zhang and Sun, 2013) is a promising direction towards enabling end-user data analysis.

Synthesis of parallel programs Numerous works have addressed the problem of synthesizing parallel programs—for high-performance applications (Xu et al., 2014), automatic vectorization (Barthe et al., 2013), and graph algorithms (Prountzos et al., 2012, 2015). Our work is fairly different both in application and technique: we synthesize data-parallel programs for MapReduce-like systems using input–output examples, as opposed to reference implementations or high-level specifications.

Data-parallel programming and compilation A range of communities have studied data-parallel programming. We address the most related works. Radoi et al. (2014) studied the problem of compiling sequential loops into MapReduce programs by translating Java loops into a λ-calculus with `fold` and using rewrite rules to create mappers. Our domain here

is different: synthesis from examples. However, our approach opens the door to blackbox parallelization, in which a sequential program is queried for input–output examples and a synthesis engine proposes candidate data-parallel programs.

Raychev et al. (2015) recently proposed parallelizing sequential user-defined aggregations (over lists) by *symbolically executing* aggregations on chunks of the input list in parallel. This development is interesting from our perspective as we might be able to (if needed) synthesize sequential reducers that can be run in parallel. Yu et al. also looked at the problem of parallelizing aggregations by detecting that an aggregation is *associatively decomposable* (Yu et al., 2009).

Hyperproperty verification Hyperproperty-verification techniques include self-composition (Barthe et al., 2004), product programs (Barthe et al., 2011, 2014; Zaks and Pnueli, 2008) and relational Hoare logic (Benton, 2004; Carbin et al., 2012). Our CSG verification can be seen as a self-composition encoding of programs into SMT formulas. Recently, Chen et al. (2015) studied decidability of the problem of verifying determinism of Hadoop-style reducers (over lists), and proposed a reduction to sequential assertion checking. Our problem is different in that our setting is functional, and we need to only consider binary reduce functions to prove determinism.

4 PRIVACY-AWARE SYNTHESIS

We have seen multiple techniques proposed to tackle a variety of program synthesis problems in the space of data analysis (Gulwani et al., 2012; Yaghmazadeh et al., 2017; Wang et al., 2017a; Zhang and Sun, 2013; Smith and Albarghouthi, 2016; Feng et al., 2017) An implicit assumption in these works is that the data is fully accessible to the user. However, today privacy is paramount, and not only have a number of systems have been proposed by the research community for enforcing differential privacy (McSherry, 2009; Roy et al., 2010; Johnson et al., 2018b; Proserpio et al., 2014), major corporations (Erlingsson et al., 2014; Apple, accessed 11-11-2017; Johnson et al., 2018b) and governments (Bureau, accessed 11-11-2017; Haney et al., 2017) have started incorporating differential privacy to protect sensitive personal information in their data analysis.

Our next goal is to aid users in constructing programs to execute on DP-enforcing systems. Given a user-provided specification of a program, we seek to synthesize an approximate differentially private program that while ensuring that it has a low privacy cost. Reasoning about a program's privacy cost is a complex process, akin to reasoning about a program's runtime complexity. The *linear dependent type system* DFuzz (Gaboardi et al., 2013) allows us to reason about the sensitivity of a randomized program—which is treated as a resource—and hence its privacy cost.

Unfortunately, type-directed synthesis procedures are inversions of top-down type-checking algorithms, and the only known type-checking algorithm for DFuzz de Amorim et al. (2014) works *bottom-up*. To enable synthesis in this setting, we will need to (*i*) introduce a language of *symbolic context constraints*, which succinctly characterize the infinitely-many possible typing contexts of an incomplete program, and (*ii*) develop a constraint abduction technique that answers when linear dependent subtyping holds. Our solutions to these challenges, amongst others, allow

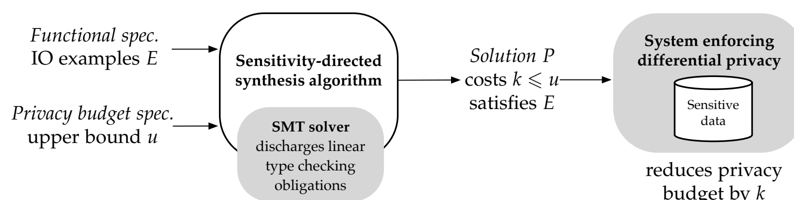


Figure 4.1: Overview of privacy-aware synthesis technique and setting

us to construct a powerful synthesis algorithm, which we call *sensitivity-directed synthesis*. The contents of this chapter are based on the work of Smith and Albarghouthi (2019b).

4.1 Illustrating Privacy-Aware Synthesis

In this section, we discuss two examples of programs that our approach can synthesize, and use them to illustrate the DFuzz type system as it relates to differential privacy.

Example 1: Aggregation Query

Suppose we would like to calculate the number of patients diagnosed with cancer in a hospital without sacrificing the privacy of individual patients. Differential privacy stipulates that a certain amount of random noise needs to be added to the computation. For example, the randomized function `f` computes the total number of patients diagnosed with cancer (stored in `c`), assuming the database `m` is a multiset of pairs of "Patient ID" and "Diagnosis". `f` returns a noisy version of the true count `c` by sampling from a Laplace distribution with mean `c` and scale $1/\epsilon$:

```
let f m = Laplace c
    where c = count p
    where p = filter (fun (k, v) -> v = "Cancer") m
```

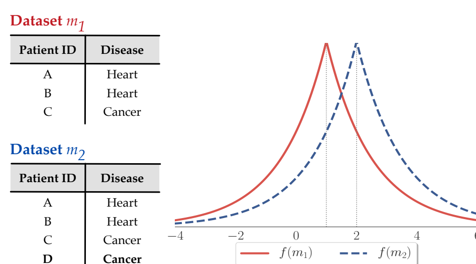


Figure 4.2: Applying the function f to adjacent databases m_1 and m_2

The type of f is $\texttt{mset}\,[\texttt{row}]\,[\infty] \multimap_{\epsilon} \bigcirc\mathbb{R}$: it takes a multiset of rows of unbounded size, denoted by the *indexed type* $\texttt{mset}\,[\texttt{row}]\,[\infty]$, and returns a *sample* from a distribution over real numbers, denoted by the probability monad $\bigcirc\mathbb{R}$. The ϵ denotes the *sensitivity* of the function (Definition 2.4). In our case, adding or deleting one row from the input multiset changes the output probability distribution by a multiplicative factor of ϵ. Formally, the *metric* used over probability distributions by the type system implies *ϵ-differential privacy*: for any two multisets, m_1 and m_2 differing by an element (so $d_{\mathcal{D}}(m_1, m_2) \leqslant 1$) and for any subset $S \subseteq \mathbb{R}$, we have

$$\Pr[f(m_1) \in S] \leqslant e^{\epsilon} \cdot \Pr[f(m_2) \in S]$$

As a visual illustration, consider m_1 and m_2 in Figure 4.2. The two multisets differ by a single element, patient D. The differential privacy guarantee ensures that a physician applying f before and after the new patient D is added should not be able to tell whether patient D has cancer, since the two distributions are close to each other, and the physician only observes a single sample. By applying repeated queries, the physician can infer the actual value of the query with high confidence, but the system executing queries on the dataset will enforce a fixed *privacy budget*, protecting from such statistical-inference attacks.

Synthesis Technique To synthesize the function `f` above, the user needs to supply (*i*) a set of input–output examples describing `f` and (*ii*) an upper bound on the sensitivity of the function, e.g., ϵ. Note that since `f` is randomized, the input–output examples will refer to the de-noised version of the function; i.e., when checking whether `f` satisfies the examples, our algorithm evaluates `f` on the examples and replaces privacy mechanisms like `Laplace c` with the function that deterministically returns `c` (the mean). For instance, the user may provide a toy dataset like m_2 in Figure 4.2 as an input example with the corresponding output 2.

Our synthesis algorithm operates in a top-down fashion, refining an incomplete program until it is complete, satisfies the input-output examples, and is within the given sensitivity upper bound. For an illustration, consider the following incomplete function:

```
let f m = Laplace (sum ●)
```

The function `sum` computes the sum of elements in a multiset of real numbers, so it has type `mset` $[\mathbb{R}]\ [\infty] \multimap_\infty \mathbb{R}$. Notice that `sum` is *infinitely* sensitive: adding a number to the input multiset may result in an arbitrary change to the total sum. Our algorithm realizes that it is impossible to complete `f` and still result in an ϵ-DP function, as adding any amount of noise to the result of an ∞-sensitive deterministic computation corresponds to an ∞-DP function—i.e., no privacy is guaranteed. Therefore, the whole search space rooted at the incomplete program f is pruned. The same would hold if we were to replace `sum` with a 2-sensitive function of type `mset` $[\mathbb{R}]\ [\infty] \multimap_2 \mathbb{R}$: our synthesis algorithm would determine that any completion would result in, at best, a 2ϵ-DP function.

This *sensitivity-directed pruning* of the search space is guided by a set of type constraints over symbolic sensitivity variables that are maintained along with incomplete programs. We show that these type constraints can be translated to the theory of real closed fields and checked for satisfiability using SMT solvers. When those constraints are unsatisfiable, we know

that no completion is possible for an incomplete program.

Types of Higher-Order Functions To give a better a taste of the type system, consider the type of `map` over multisets:

$$\texttt{map} : \forall \alpha, \beta. \forall s. (\alpha \to \beta) \multimap_{2 \cdot s} \texttt{mset}\,[\alpha]\,[s] \multimap_{1} \texttt{mset}\,[\beta]\,[s]$$

Semantically, `map` takes a function g and a multiset m (with up to s elements) and applies g to every element of m to get a new dataset with the same size upper bound. The type signature of `map` encodes its *privacy* semantics. If the function g is modified in any way, the resulting multiset is different by at most $2s$ elements, as denoted by the sensitivity of the first argument. Similarly, if the input multiset is modified by adding or deleting one element, this results in a multiset that is different by at most 1 element, as denoted by the sensitivity of the second argument.

Example 2: Iterative Privacy Mechanism

The differential privacy literature has a number of privacy-preserving variants of popular algorithms. We will see, for instance, how to synthesize a differentially private version of the `k-means` clustering algorithm, which has the following type:

$$\forall i, k. \underbrace{\mathbb{N}[i]}_{\text{\# iterations}} \multimap_{\infty} \underbrace{\texttt{L}(\langle \mathbb{R}, \mathbb{R} \rangle)[k]}_{\text{initial clusters}} \multimap_{\infty} \underbrace{\texttt{mset}\,[\langle \mathbb{R}, \mathbb{R} \rangle]\,[\infty]}_{\text{dataset}} \multimap_{3 \cdot i \cdot \epsilon} \underbrace{\bigcirc \texttt{L}(\langle \mathbb{R}, \mathbb{R} \rangle)[k]}_{\text{final clusters}}$$

The goal of `k-means` is to map a multiset of points to k clusters. It takes the number of iterations of the `k-means` update procedure (`k-step`) to execute, a list of initial clusters, and a multiset of points (in $\mathbb{R}^2$). Observe that the privacy (sensitivity) of `k-means` is a function of the number of iterations of a single step: $3 \cdot i \cdot \epsilon$. Given a small example input dataset and expected output, our algorithm is able to synthesize a recursive implementation

```
let k-means iter centers data = match iter with
    | 0 -> return centers
    | n + 1 ->
       let-draw centers = k-means n centers data in
       k-step centers data
```

Figure 4.3: Implementation sketch for `k-means`, adapted from Gaboardi et al. (2013)

```
let idc iter data queries = match iter with
    | 0 -> return init_approx
    | n + 1 ->
       let-draw approx = idc n data queries in
       let-draw query = q-select queries approx data in
       let-draw actual = Laplace (eval-q query data) in
       return (dua approx query actual)
```

Figure 4.4: Implementation sketch for `idc`, adapted from Gaboardi et al. (2013).

of differentially private `k-means`, similar to the one presented by Gaboardi et al. (2013).

An implementation of `k-means` is given in Figure 4.3, and an implementation of the *iterative database construction* algorithm by Gupta et al. (2012), which iteratively learns a synthetic database (using `dua`, a database update mechanism such as multiplicative weights (Hardt et al., 2012)) to accurately answer a set of queries, is given in Figure 4.4. Our algorithm is also able to synthesize `idc` when provided with a small set of examples.

4.2 The Synthesis Problem

Before defining the synthesis problem, we present a simplified view of the linear dependent type system DFuzz . Refer to Gaboardi et al. (2013) for a full description.

Differences from DFuzz The system described below is a *variant* of DFuzz containing minor changes. The changes are not substantial enough to qualify the variant as a distinct system, so we will continue to refer to our type system as DFuzz and simply highlight the changes here.

To support compositions of higher-order operators (such as `map` from Section 5.1), we enable type polymorphism by augmenting DFuzz with type quantifiers (similar to de Amorim et al. (2014)).

Databases in DFuzz are elements of a type with no size information. We treat databases as elements of the type $\texttt{mset}\,[\tau]\,[S]$, which are multisets containing *at most* S elements of type τ. Including multisets smaller than S in the type allows us to compute the distance between multisets whose dependent indices are not equal by using the resulting subtyping relation (discussed below) to coerce their types into agreeing. Refer to Appendix B.1 for details of this modification.

Overview DFuzz uses a *modal* operator $!_k$ from linear logic to keep track of the sensitivity of a value. A function of type $!_k\sigma \multimap \tau$ is k-sensitive in its first argument, and a *typing context* containing the assumption $x : !_k\tau$ indicates the typing context can type expressions that are at most k-sensitive in x. We will simplify the syntax of these statements, and instead write them as $\sigma \multimap_k \tau$ and $x :_k \tau$. Furthermore, we use the traditional arrow $\sigma \to \tau$ for ∞-sensitive functions (i.e., $\sigma \multimap_\infty \tau$).

Syntax DFuzz types, context, and expressions are constructed from the grammar in Figure 4.5, which we elaborate below:

1. **Sensitivity and size:** Sensitivity and size expressions (R and S) consist of variables (k and i), constants (0, c, and ∞), and simple arithmetic. These expressions are used in (*i*) modal types and (*ii*) as constraints on our *precise types*.

2. **Precise types:** Precise types are types *dependent* on sensitivity and size expressions. $\mathbb{R}[R]$ and $\mathbb{N}[S]$ are the reals and naturals whose value is precisely R or S; $\texttt{mset}\ [\tau]\ [S]$ are multisets with *at most* S elements of type τ, and $L(\tau)[S]$ are lists with precisely S elements of type τ.

3. **Probability monad:** Types of probabilistic values of type τ are written using a *monadic type* $\bigcirc\tau$. Values can be lifted to probabilistic values using $\texttt{return}\ e$, and sampled from probabilistic values using $\texttt{let-draw}\ x = e_1\ \texttt{in}\ e_2$ (where x is drawn from distribution e_1 and used in e_2).

4. **Dependent pattern-matching:** Precise types $\mathbb{N}[S]$ and $L(\tau)[S]$ are eliminated using dependent pattern-matching expressions, which use information about the constructors of the precise types to constrain the value of the size term. For example, if we match a precise natural with the constructor 0, we know $S = 0$.

5. **Quantifiers:** Quantifiers ($\forall$) bind size, sensitivity, and type variables. We restrict type quantifiers to be *predicative*, but let size and sensitivity quantifiers appear anywhere. We use *free*(τ) to denote free variables in τ.

6. **Contexts and constraints:** A typing context Γ maps variables to modal types. Our typing judgements will depend on constraints Φ on sensitivity variables. For instance, Φ may specify that $k \leqslant 5$, where k is a sensitivity variable. We use $\textsc{Sat}(\Phi)$ to denote that a constraint Φ is satisfiable. A constraint Φ is a conjunction of inequalities over integers and reals, with (*i*) non-linear arithmetic and (*ii*) ∞, following the semantics of de Amorim et al. (2017); multiplying by ∞ is non-commutative, so $r \cdot \infty = \infty$ but $\infty \cdot r = 0$ when $r = 0$ and $\infty \cdot r = \infty$ for $r > 0$.

7. **Expressions:** We have a simple expression language over variables x and components c from a provided set C—the CFG defining e induces a signature, which we'll call Σ^C (Definition 2.8). We also allow for term-, type-,

and sensitivity-abstractions and applications as in System F, and recursion using $\texttt{fix}\, x.\, e$.

Typing and Context Arithmetic DFuzz provides a typing judgement of the form $\Phi, \Gamma \vdash e : \tau$, specifying that e is of type τ under context Γ and assuming constraint Φ holds. We highlight the crucial typing rule—function application—and leave the rest to Gaboardi et al. (2013).

$$\frac{\Phi, \Gamma \vdash f : \sigma \multimap_R \tau \qquad \Phi, \Delta \vdash e : \sigma}{\Phi, \Gamma + R \cdot \Delta \vdash f\ e : \tau}\ (\multimap\text{-E}\textsc{lim}.)$$

Observe the typing context arithmetic in the consequent; DFuzz defines *context scaling* and *addition* as a generalization of context union. This arithmetic encodes the fact that sensitivities multiply through function composition in the type system. Formally:

Definition 4.1 (Context Arithmetic)**.** *Let Γ and Δ be typing contexts where, for all variables x, if $(x :_T \sigma) \in \Gamma$ and $(x :_{T'} \tau) \in \Delta$, then $\sigma = \tau$. For sensitivity expressions R, we define:*

$$\begin{aligned}\Gamma + R \cdot \Delta = & \{x :_{T+R\cdot T'} \sigma \mid (x :_T \sigma) \in \Gamma,\ (x :_{T'} \sigma) \in \Delta\} \\ & \cup \{x :_T \sigma \mid (x :_T \sigma) \in \Gamma,\ x \notin \mathit{dom}(\Delta)\} \\ & \cup \{x :_{R\cdot T'} \sigma \mid (x :_{T'} \sigma) \in \Delta,\ x \notin \mathit{dom}(\Gamma)\}\end{aligned}$$

Example 4.2. *Let f be the function $\lambda x : \mathbb{R}.\, 2 \cdot x$ and assume some context Γ gives us the judgement $\top, \Gamma \vdash f : \mathbb{R} \multimap_2 \mathbb{R}$. To apply f to a variable $y : \mathbb{R}$, we can apply ($\multimap$-E*LIM*.) to the natural judgement*

$$\top, \{y :_1 \mathbb{R}\} \vdash y : \mathbb{R}$$

producing the consequent

$$\top, \Gamma + 2 \cdot \{y :_1 \mathbb{R}\} \vdash f\ y : \mathbb{R}$$

Sensitivity and size expressions
k is a *sensitivity variable*, i is a *size variable*, and t is a *type variable*

$R \coloneqq k \mid c \in \mathbb{R}^{>0} \mid S \mid R + R \mid R \cdot R \mid \infty$ (sensitivity expression)
$S \coloneqq i \mid 0 \mid S + 1 \mid \infty$ (size expression)

Linear dependent types

$\tau \coloneqq a \mid Z \mid \alpha \mid A \multimap \tau \mid \langle \tau, \tau \rangle \mid \bigcirc \tau$ (types)
$A \coloneqq\ !_R \tau$ (modal types)
$\alpha \coloneqq \mathbb{R} \mid \texttt{bool} \mid \ldots$ (base types)
$Z \coloneqq \mathbb{R}[R] \mid \mathbb{N}[S] \mid L(\tau)[S] \mid \texttt{mset}\ [\tau]\ [S]$ (precise types)
$a \coloneqq \forall n.\, \tau$ (quantifiers)
$n \coloneqq k \mid i \mid t$ (kinds of variables)

Constraints and typing context

$\Phi \coloneqq \top \mid \Phi \wedge \Phi \mid S = S \mid R \leqslant R$ (constraints)
$\Gamma \coloneqq \emptyset \mid \Gamma, x : A$ (typing context)

Programs

$e \coloneqq x \mid c \in C$ (expressions)
$\mid e\ e$ (application)
$\mid \lambda x.\, e \mid \texttt{fix}\ x.\, e$ (abstraction and fixpoints)
$\mid \texttt{return}\ e \mid \texttt{let-draw}\ x = e\ \texttt{in}\ e$ (return and bind for $\bigcirc$)
$\mid \texttt{match}_{\mathbb{N}}\ e\ \texttt{with}\ 0 \to e \mid x \to e$ (pattern-matching)
$\mid \texttt{match}_{L}\ e\ \texttt{with}\ \texttt{nil} \to e \mid \texttt{cons}\ (x, y) \to e$ (pattern-matching)
$\mid e[R] \mid e[\tau]$ (sens./size and type app.)
$\mid \Lambda k.\, e \mid \Lambda i.\, e \mid \Lambda t.\, e$ (sens., size, and type abs.)

Figure 4.5: Grammars of DFuzz types and programs

By Definition 4.1, $\Gamma + 2 \cdot \{y :_1 \mathbb{R}\} = \Gamma \cup \{y :_2 \mathbb{R}\}$, assuming y is not bound in Γ.

Subtyping DFuzz defines a notion of subtyping using the judgement $\Phi; \Gamma \models \sigma \sqsubseteq \tau$, where the constraint Φ determines whether the relations between sensitivity variables are appropriate to subtype in context Γ. This judgement is defined by a set of inference rules, two of which we present here. The rule

$$\frac{\Phi \models r' \leqslant r \quad \Phi; \Gamma \models \sigma \sqsubseteq \sigma'}{\Phi; \Gamma \models\ !_r \sigma \sqsubseteq\ !_{r'} \sigma'} (\sqsubseteq .!_r)$$

states that we can replace usage of an r'-sensitive σ' with an r-sensitive σ, as long as we do not decrease the sensitivity (or $r' \leqslant r$). As subtyping functions is standard (contravariant in the domain and covariant in the codomain), with $\sqsubseteq$-reflexivity we can encode the fact that a c-sensitive function is also c'-sensitive when $c \leqslant c'$ in the subtyping system with the valid judgement:

$$R \leqslant T \ \models \ \sigma \multimap_R \tau \sqsubseteq \sigma \multimap_T \tau$$

To subtype databases with size terms, we use the rule

$$\frac{\Phi \models S \leqslant T \quad \Psi; \Gamma \models \sigma \sqsubseteq \tau}{\Phi \wedge \Psi; \Gamma \models \texttt{mset}\ [\sigma]\ [S] \sqsubseteq \texttt{mset}\ [\tau]\ [T]} (\sqsubseteq .\texttt{MSET})$$

which allows us to weaken the type of a database by *increasing* the size bound. In Appendix B.1, we show how this rule integrates with DFuzz's metric preservation claims.

Formalizing the Synthesis Task

A privacy-aware synthesis task S is given as a tuple $S = (\sigma, C, \phi_k, \phi_s)$, where

1. σ is a *goal type* with *free*$(\sigma) = \{k\}$.

2. C is a set of components. We assume all functions $f \in C$ are provided with an interpretation in the canonical algebra and a *deterministic alternative interpretation* (Definition 4.3).

3. ϕ_k is a first-order formula encoding a *budget specification* constraining a sensitivity variable k.

4. ϕ_s is a *functional specification* encoding the desired semantics of the solution.

A program p satisfies S if the sensitivity of p satisfies ϕ_k—i.e., $\phi_k \coloneqq k \leqslant 1$ imposes an upper bound on sensitivity. Additionally, p should be well-formed, well-typed, and should satisfy the functional specification ϕ_s. If p uses components that compute probabilistic values, we cannot check satisfaction of the functional specification ϕ_s. Instead, we require that the canonical deterministic alternative interpretation of p—denoted $[\![p]\!]_d$—satisfies ϕ_s.

We assume that every probabilistic component $c \in C$—in addition to being equipped with an interpretation $[\![c]\!]$ in the canonical algebra—is equipped with a deterministic alternative interpretation $[\![c]\!]_d$ with the following properties: (*i*) $[\![c]\!]_d$ returns a probability distribution with support on a single value, and (*ii*) $[\![c]\!]$ and $[\![c]\!]_d$ have the same domain and codomain in the canonical carrier set. Semantically, deterministic alternative interpretations should return the *most likely output* of their probabilistic counterparts. For example, the function `Laplace`, which samples from the Laplace distribution, has the alternative interpretation $[\![\texttt{Laplace}]\!]_d(c)$ that simply returns the mean c. Further examples are given in Table 4.1.

Using deterministic alternative interpretations, we can define a notion of *deterministic satisfaction*:

Definition 4.3. *Let* p *be a probabilistic program with components from* C*, and let* p_d *be the map defined as follows: for all* i, o *in the canonical carrier set,* $p(i) =$

Deterministic Alternative (w/description)
$[\![\texttt{Bern}]\!]_d = \lambda p.\,\texttt{return}\ (p > 1/2)$ *returns true if* $p > 1/2$, *false otherwise*
$[\![\texttt{Laplace}]\!]_d = \lambda x.\,\texttt{return}\ x$ *returns the mean* x
$[\![\texttt{ExpMech}^S_d]\!] = \lambda u.\,\lambda d.\,\text{argmax}_{s \in S} u(d, s)$ *returns the* $s \in S$ *maximizing* u *on database* d

Table 4.1: Deterministic alternatives to probabilistic functions

o *if and only if* o *has non-zero support in the distribution* $[\![p]\!]_d\,(i)$. *As* $[\![p]\!]_d$ *has support on a single value,* p_d *is a* function.

We say p deterministically satisfies *the functional specification* ϕ_s *(written* $p \models_d \phi_s$*) if* $p_d \models \phi_s$.

Deterministic satisfaction does not ensure *all* executions of a probabilistic program p satisfy ϕ_s. Rather, as DFuzz models probabilistic states by distributions with finite support, we are simply ensuring that the executions where $p \models \phi_s$ classically have non-zero support. If we further assume that all probabilistic components operate independently (which may not always be the case, due to control-flow effects), we get a slightly stronger property: *the most likely execution* of p satisfies ϕ_s.

We will construct a *correctness constraint* (Definition 2.17) from S:

Definition 4.4. *Let* $S = (\sigma, C, \phi_k, \phi_s)$ *be a privacy-aware synthesis task, and let* ϕ_S *be the induced correctness constraint. A program* $p \in T_{\Sigma^C}$ *is a solution to* ϕ_S*, denoted* $p \models \phi_S$*, if and only if:*

1. $p \models_d \phi_s$*, and*

2. *there is a model* M *inducing the interpretation* $[\![\cdot]\!]_M$ *over contexts, expressions, and types that replaces free sensitivity variables with constants such that*

$$M \models \phi_k \quad \textit{and} \quad [\![\Gamma_C]\!]_M \vdash [\![p]\!]_M : [\![\sigma]\!]$$

where Γ_C *is the typing context* $\{c :_\infty \tau \mid c : \tau \in C\}$.

The following example gives a concrete synthesis task and a solution of the induced correctness constraint:

Example 4.5. *Consider the synthesis problem* $\langle \mathbb{R} \multimap_k \mathbb{R}, C, \phi_k, \phi_s \rangle$*, where*

$$\phi_k \coloneqq 0 < k \leqslant 2 \quad \textit{and} \quad \phi_s \coloneqq p(0) = 1 \wedge p(2) = 5$$

and the components are

$$C = \{\mathit{square} : \mathbb{R} \multimap_\infty \mathbb{R},\ \mathit{double} : \mathbb{R} \multimap_2 \mathbb{R},\ \mathit{succ} : \mathbb{R} \multimap_1 \mathbb{R}\}$$

A solution is $p = \lambda x.\, \mathit{succ}(\mathit{double}(x)) : \mathbb{R} \multimap_k \mathbb{R}$*, with witnessing model* M *setting* k *to 2.*

4.3 Synthesis with Linear Dependent Types

Our sensitivity-directed synthesis algorithm operates in a top-down fashion, iteratively refining an incomplete program into a complete one that satisfies the specification and budget constraints. The process is enabled and guided by two key ideas:

1. **Symbolic context constraints** (SCC): A SCC succinctly captures all possible typing contexts that can type a program. This allows us to symbolically type-check incomplete programs and prune the search space when the sensitivity budget is insufficient.

2. **Constraint abduction**: As discussed in Section 5.1, DFuzz's subtyping judgement relies on constraints over sensitivity and size variables. During synthesis, to replace a wildcard of type τ with an expression of type $\sigma \sqsubseteq \tau$, we must ask "*what constraints do we need to ensure subtyping works?*" This

is a *logical abduction* question. We present a technique to abduce the *most general subtyping constraints*.

Our algorithm is defined by a set of inference rules that let us derive *synthesis states* from a *synthesis state*. A synthesis state $\langle \phi,\ p \rangle$ is a pair consisting of: (*i*) a proof obligation ϕ determining when p is well-typed and satisfies the privacy budget, and (*ii*) an incomplete program p containing annotated wildcards $\bullet_{\tau}^{\Omega}$, where τ is a goal type and Ω is a *symbolic typing context* (defined shortly).

Invariants Our algorithm maintains the invariant that if, for some synthesis state $\langle \phi,\ p \rangle$, the proof obligation ϕ is unsatisfiable, then for every synthesis state $\langle \phi',\ p' \rangle$ derived from $\langle \phi,\ p \rangle$, formula ϕ' is also unsatisfiable. This allows us to discontinue inference from unsatisfiable subproblems, as it means they cannot satisfy sensitivity budget constraints.

Inference Rules and Symbolic Constraints

We now detail our algorithm's inference rules , given in Figures 4.6 and 4.7. Implementation details are left to Section 4.5.

Initialization and Termination

The rule (Init) initiates the synthesis process: starting with the synthesis problem $\langle \sigma, C, \phi_k, \phi_s \rangle$, we build the synthesis state $\left\langle \phi_k \wedge \Omega = \Gamma_C,\ \bullet_{\sigma}^{\Omega} \right\rangle$ indicating that we want an expression that satisfies the sensitivity requirements of ϕ_k and is of type σ. The functional specification ϕ_s is uniform across all synthesis states derivable from the same synthesis problem, and so we do not explicitly propagate it. Γ_C is the typing context $\{c :_{\infty} \tau \mid c : \tau \in C\}$ of all components in the synthesis domain C.

Rule (Finish) encodes the definition of a solution to the synthesis problem (Definition 4.4): a solution p must be complete and must determin-

Initialization and termination

$$\frac{\langle \sigma, C, \phi_k, \phi_s \rangle \quad \Omega \text{ fresh}}{\langle \phi_k \wedge \Omega = \Sigma_C,\ \bullet^{\Omega}_{\sigma} \rangle}\ (\textsc{Init}) \qquad \frac{\langle \phi,\ p \rangle \quad \textsc{Sat}(\phi) \quad p \models_d \phi_s \quad p \text{ complete} \quad \textsc{Term}(p)}{p \text{ is a solution}}\ (\textsc{Finish})$$

Application, abstraction, and variable/function introduction

$$\frac{\langle \phi,\ t\left[\bullet^{\Omega}_{\sigma}\right] \rangle \quad \Omega_1, \Omega_2, r \text{ fresh}}{\left\langle \phi \wedge \Omega = \Omega_1 + r \cdot \Omega_2,\ t\left[\bullet^{\Omega_1}_{\tau \multimap_r \sigma}\ \bullet^{\Omega_2}_{\tau}\right] \right\rangle}\ (\textsc{App}) \qquad \frac{\langle \phi,\ t\left[\bullet^{\Omega}_{\tau \multimap_r \sigma}\right] \rangle \quad \Omega_1 \text{ is fresh}}{\langle \phi \wedge \Omega_1 = \Omega \oplus \{x :_r \tau\},\ t\left[\lambda x : \tau.\ \bullet^{\Omega_1}_{\sigma}\right] \rangle}\ (\textsc{Abs})$$

$$\frac{\langle \phi,\ t\left[\bullet^{\Omega}_{\sigma}\right] \rangle \quad \sigma \prec_{[t/t']} \tau \quad t \text{ is fresh}}{\langle \phi,\ t\left[\bullet^{\Omega}_{\forall t.\tau}[t']\right] \rangle}\ (\text{TA}\textsc{pp})$$

$$\frac{\langle \phi,\ t\left[\bullet^{\Omega}_{\sigma}\right] \rangle \quad \gamma; \psi \vdash \tau \leftarrow \sigma \quad \nu : \tau \in \textsc{Scope}(\phi, \Omega)}{\langle \phi \wedge \gamma \wedge \psi \wedge \Omega = \{\nu :_1 \tau\},\ t\left[\nu\right] \rangle}\ (\textsc{Id})$$

Figure 4.6: Basic synthesis rules; $\prec_{\gamma}$ is *generalization* (Definition 4.9), $\textsc{Term}(\cdot)$ is a termination oracle, and $\textsc{Scope}(\phi, \Omega)$ returns the set $\{x_i : \tau_i\}$ such that, for all models $M \models \phi$, for all i we have $M \models x_i : \tau_i \in \Omega$

istically satisfy the functional specification ϕ_s. Furthermore, the proof obligation ϕ must be satisfiable—our inference rules maintain the invariant that when ϕ is satisfiable, the expression p is well-typed and obeys the sensitivity constraint ϕ_k. In proving Theorem 4.18 we will formalize this notion.

Symbolic Context Constraints

The inference rules can be viewed as inversions of the rules defining the DFuzz typing judgement. Consider the rule $(\textsc{App})$ applied to $\langle \phi,\ t\left[\bullet^{\Omega}_{\sigma}\right] \rangle$. $(\textsc{App})$ is an inversion of the DFuzz rule for $\multimap$-elimination, from Section 5.1, which specifies that an expression of type σ can be generated if we have (for some choice of τ)

Recursion and pattern-matching

$$\frac{\langle \phi,\ t\,[\bullet^{\Omega}_{\sigma}]\rangle \quad \Omega_1, \Omega_2 \text{ fresh}}{\langle \phi \wedge \Omega = \infty \cdot \Omega_2 \wedge \Omega_1 = \Omega_2 \oplus \{x :_\infty \sigma\},\ t\,[\texttt{fix}\ x.\ \bullet^{\Omega_1}_{\sigma}]\rangle}\ (\textsc{Fix})$$

$$\frac{\langle \phi,\ t\,[\bullet^{\Omega}_{\sigma}]\rangle \quad \Omega_1, \Omega_2, \Omega_3, \Omega_4, r \text{ fresh}}{\left\langle \phi',\ t\left[\texttt{match}_{\mathbb{N}}\ \bullet^{\Omega_1}_{\mathbb{N}[s]}\ \texttt{with}\ 0 \to \bullet^{\Omega_2}_{\sigma[s/0]} \mid x[i]+1 \to \bullet^{\Omega_3}_{\sigma[s/i+1]}\right]\right\rangle}\ (\textsc{Match})_{\mathbb{N}}$$

where $\phi' = \phi \wedge\ \Omega = \Omega_4 + r \cdot \Omega_1 \wedge \Omega_2 = \Omega_4\,[s/0] \wedge\ \Omega_3 = \Omega_4\,[s/i+1] \oplus \{x :_r \mathbb{N}[i]\}$

$$\frac{\langle \phi,\ t\,[\bullet^{\Omega}_{\sigma}]\rangle \quad \Omega_1, \Omega_2, \Omega_3, r \text{ fresh}}{\left\langle \phi',\ t\left[\texttt{match}_{L}\ \bullet^{\Omega_1}_{L(\tau)[s]}\ \texttt{with nil} \to \bullet^{\Omega_2}_{\sigma[s/0]} \mid \texttt{cons}\,(y, x[i]) \to \bullet^{\Omega_3}_{\sigma[s/i+1]}\right]\right\rangle}\ (\textsc{Match})_{L}$$

where $\phi' = \phi \wedge\ \Omega = \Omega_1 + r \cdot \Delta \wedge \Omega_2 = \Omega_1\,[s/0] \wedge\ \Omega_3 = \Omega_1\,[s/i+1] \oplus \{y :_r \tau, x :_r L(\tau)[i]\}$

Monadic operations

$$\frac{\langle \phi,\ t\,[\bullet^{\Omega}_{\bigcirc\sigma}]\rangle \quad \Omega_1 \text{ fresh}}{\langle \phi \wedge \Omega = \infty \cdot \Omega_1,\ t\,[\texttt{return}\ \bullet^{\Omega_1}_{\sigma}]\rangle}\ (\textsc{Return})$$

$$\frac{\langle \phi,\ t\,[\bullet^{\Omega}_{\bigcirc\sigma}]\rangle \quad \Omega_1, \Omega_2, \Omega_3 \text{ fresh}}{\left\langle \phi',\ t\left[\texttt{let-draw}\ x = \bullet^{\Omega_1}_{\bigcirc\tau}\ \texttt{in}\ \bullet^{\Omega_2}_{\bigcirc\sigma}\right]\right\rangle}\ (\textsc{LetDraw})$$

where $\phi' = \phi \wedge\ \Omega = \Omega_1 + \Omega_3 \wedge\ \Omega_2 = \Omega_3 \oplus \{x :_\infty \tau\}$

Figure 4.7: Advanced synthesis rules extending the procedure in Figure 4.6

1. an expression of type $\tau \multimap_R \sigma$ in some context Ω_1, and
2. an expression of type τ in some context Ω_2,

where $\Omega = \Omega_1 + R \cdot \Omega_2$. In order to invert this rule to construct (App), we *must know how to split* Ω; otherwise, we cannot construct the appropriate wildcards, which track contexts.

Unfortunately, there are infinitely-many choices for contexts Ω_1, Ω_2, and sensitivity expression R. To allow us to express such arbitrary context

splits in a finite manner, we introduce *symbolic context constraints* (SCC), which succinctly encode infinitely many contexts symbolically.

Definition 4.6 (Symbolic Context Constraints)**.** *A* symbolic context *is a term* C *in the grammar*

$$C \coloneqq \emptyset \mid \{x :_R \tau\} \mid \Omega \mid C + R \cdot C \mid C \oplus C \mid C[s/R],$$

where Ω *is a context variable,* R *is a sensitivity term,* τ *is a type, and* s *is a sensitivity variable.*

A symbolic context constraint E *is a conjunction of equalities of symbolic contexts, or equivalently, a term in the grammar* $E \coloneqq E \wedge E \mid C = C$.

The grammar in Definition 4.6 defines symbolic contexts C, which are expressions containing *symbolic context variables* Ω and a limited set of concrete contexts, namely the empty context $\emptyset$ and the singleton context $\{x :_R \tau\}$. Symbolic contexts can be constructed through linear combinations $C + R \cdot C$, disjoint unions $C \oplus C$ that require the symbolic contexts have no variables in common, and sensitivity substitutions $C\,[s/R]$ which replace sensitivity variables (s) with arbitrary sensitivity expressions (R). Symbolic context constraints E are conjunctions of equalities over symbolic contexts. An interpretation of a SCC E maps symbolic context variables to concrete contexts. We formalize the interpretation of these constraints in Definition 4.15.

SCCs also appear in the inference rule (Abs), which introduces a λ-abstraction. The rule specifies that we can construct a function of type $\sigma \multimap_r \tau$ in context Ω, but only if the context in the wildcard $\bullet_\tau^{\Omega_1}$ contains r uses of x. This is expressed in the constraint $\Omega_1 = \Omega \oplus \{x :_r \sigma\}$, which not only ensures that Ω_1 has r uses of x, but that Ω has *no* uses of x (as x is out of scope).

Example 4.7 (SCC)**.** *Consider the initial synthesis state*

$$\left\langle 0 < k \leqslant 2,\ \ \bullet^{\emptyset}_{\mathbb{R} \multimap_k \mathbb{R}} \right\rangle.$$

After applying (Abs)*, we derive the synthesis state*

$$\left\langle 0 < k \leqslant 2 \land \Omega = \emptyset \oplus \{x :_k \mathbb{R}\},\ \ \lambda x : \mathbb{R}.\, \bullet^{\Omega}_{\mathbb{R}} \right\rangle$$

indicating that the context Ω must contain k copies of x. Then, after applying the rule (App)*, we derive the following synthesis state*

$$\left\langle \phi,\ \ \lambda x : \mathbb{R}.\, \bullet^{\Omega_1}_{\mathbb{R} \multimap_l \mathbb{R}}\ \bullet^{\Omega_2}_{\mathbb{R}} \right\rangle$$

where $\phi = 0 < k \leqslant 2 \land \Omega = \emptyset \oplus \{x :_k \mathbb{R}\} \land \Omega = \Omega_1 + l \cdot \Omega_2$, indicating that if we wish to apply an l-sensitive function (where l is fresh), Ω must contain l uses of variables in context Ω_2.

Due to our maintained invariant, an unsatisfiable subproblem can be pruned from the search, as it yields no solutions. The following example illustrates pruning:

Example 4.8 (Pruning)**.** *Let us start from the last synthesis state in Example 4.7. Suppose we replace the $\bullet^{\Omega_1}_{\mathbb{R} \multimap_l \mathbb{R}}$ with the ∞-sensitive function `square` (which squares a real number). Using the rule* (Id)*, this results in the subproblem with the following constraints (where we have replaced Ω_1 with $\emptyset$):*

$$(0 < k \leqslant 2) \land (\Omega = \{x :_k \mathbb{R}\}) \land (\Omega = \emptyset + l \cdot \Omega_1) \land l \geqslant \infty$$

This simplifies to $0 < k \leqslant 2 \land \{x :_k \mathbb{R}\} = \infty \cdot \Omega_2$, which is unsatisfiable: the context on the right has ∞ copies of x, while the left as at most 2 copies.

Pattern Matching

Pattern-matching over the precise types $\mathbb{N}[S]$ and $L(\tau)[S]$, introduced using rules $(\text{Match})_{\mathbb{N}}$ and $(\text{Match})_{L}$, gives a mechanism for concretizing our understanding of the sensitivity of an expression by *refining* the value of S. For instance, if we match expression e of type $\mathbb{N}[s]$ with the precise natural constructor 0, in the 0-branch wildcard we may freely assume that $s = 0$. This assumptions is propagated in two ways: (*i*) by substituting all instances of s with 0 in the type annotation of the wildcard, and (*ii*) by restricting the wildcard context via the constraint $\Gamma_1 = \Gamma[s/0]$, which requires that Γ_1 is simply Γ with all instances of s replaced by 0.

Recursion

The (Fix) rule introduces fixpoint expressions $\texttt{fix}\, x.\, e$, allowing for the construction of recursive programs. We must allow for unlimited uses of the recursion variable in e, as constrained by the conjunct $\Omega_1 = \Omega_2 \oplus \{x :_\infty \sigma\}$ in the consequent, and for potentially infinitely-many recursive expansions of e, restricting our context to be infinitely-sensitive with respect to all variables used in e (as expressed by $\Omega = \infty \cdot \Omega_2$).

Recursion can, of course, produce non-terminating programs. We assume the existence of a termination oracle, $\text{Term}(p)$, which returns *true* if p terminates for all inputs. In practice, our oracle is a procedure that soundly identifies terminating recursive expansions using the natural well-founded order over the types $\mathbb{N}[S]$ and $L(\tau)[S]$.

Probability Monad

The rules (Return) and (LetDraw) are used to enable synthesis of randomized programs. Both rules have monadic types in their antecedent synthesis states, and so the type-directed nature of synthesis ensures

that rules constructing probability distributions are only used when a probability distribution is required.

The expression `return` e lifts into the probability monad. As an inversion, rule (Return) provides a mechanism to construct distributions with support for a single value. The expression generated is ∞-sensitive with respect to all variables in Ω_1 (captured by the constraint $\Omega = \infty \cdot \Omega_1$) because *close values do not result in close distributions*. That is, `return` 1 and `return` 2 are infinitely far apart in the distribution metric, despite 1 and 2 being only distance 1 away. [1]

The rule (LetDraw) introduces the DFuzz mechanism for *sampling* from distributions, the `let-draw` $x = e_1$ `in` e_2 expression. (LetDraw) enables synthesis to perform additional computations on sampled values by using x in e_2. Note that e_2 has unrestricted use of x, encoded in the constraints by the conjunct $\Gamma_2 = \Gamma_3 \oplus \{x :_\infty \tau\}$, as once a random value is sampled no post-computation can give any more information about the distribution than that one point.

Polymorphism

The rules (TApp), (SenApp), and (SizeApp) are used to enable applying polymorphic functions in our signature, e.g., `map`, by abstracting types, sensitivities, and sizes. We only show (TApp): the others are structurally identical.

Each of the three rules operates similarly: given a synthesis state with wildcard $\bullet^{\Omega}_{\sigma}$, they attempt to replace the goal type σ with a *polymorphic* type $\forall n.\, \tau$. To do so, it is necessary to *generalize* the goal type σ to introduce a free variable.

[1] For a fixed ϵ, define $d_{\bigcirc\tau}(\delta_1, \delta_2)$ to be $1/\epsilon \cdot \max_{x \in \tau} |\ln(\delta_1(x)/\delta_2(x))|$. The metric is defined precisely to ensure DFuzz 's *metric preservation theorem* guarantees privacy.

Definition 4.9 (Generalization)**.** *Let γ be a type-variable substitution. We say a type τ* type-generalizes *a type σ using γ if $\tau = \gamma(\sigma)$ and dom$(\gamma) \subseteq$ free(τ). We represent this as $\sigma \prec_\gamma \tau$.*

Sensitivity- and size-generalization are defined symmetrically.

In practice, we explore all possible generalizations, of which there are linearly-many in the size of σ.

Consider the rule (TApp): it refines a wildcard $\bullet^{\Omega}_{\sigma}$ by finding a type τ with a free *type* variable t such that $\sigma \prec_{[t/t']} \tau$. This reduces the search to finding a refinement for the wildcard $\bullet^{\Omega}_{\forall t.\tau}$, possibly enabling the introduction of a polymorphic function (such as `map`) using rule (Id).

Example 4.10. *Suppose we have a synthesis state $\left\langle \phi,\ \bullet^{\Omega}_{\mathtt{mset}[\mathit{row}][\infty] \multimap_2 \mathbb{R}} \right\rangle$, and we wish to apply the 1-sensitive function $\mathtt{count} : \forall \beta. \forall k. \mathtt{mset}\,[\beta]\,[k] \multimap_1 \mathbb{R}$. Since*

$$\mathtt{mset}\,[\mathit{row}]\,[\infty] \multimap_2 \mathbb{R} \prec_{[\alpha/\mathit{row}]} \mathtt{mset}\,[\alpha]\,[\infty] \multimap_2 \mathbb{R} \prec_{[s/\infty]} \mathtt{mset}\,[\alpha]\,[s] \multimap_2 \mathbb{R}$$

using rules (TApp) *and* (SizeApp) *we can* type-generalize *and* size-generalize $\mathtt{mset}\,[\mathit{row}]\,[\infty] \multimap_2 \mathbb{R}$ *and generate the synthesis state*

$$\left\langle \phi,\ \bullet^{\Omega}_{\forall \alpha. \forall s. \mathtt{mset}[\alpha][s] \multimap_2 \mathbb{R}} [\mathit{row}][\infty], \right\rangle$$

where the inner application is type-application *and the outer application is* size-application*. An application of the rule* (Id) *results in the subtyping abduction rules (Section 4.3) generating the judgement*

$$\top; 1 \leqslant 2 \vdash \forall \beta. \forall k. \mathtt{mset}\,[\beta]\,[k] \multimap_1 \mathbb{R} \leftarrow \forall \alpha. \forall s. \mathtt{mset}\,[\alpha]\,[s] \multimap_2 \mathbb{R},$$

allowing us to replace the wildcard with `count`*.*

$$\frac{v \notin t \quad \textsc{SensFree}(t) = \emptyset}{v = t; \top \vdash_{\{v\}} v \leftarrow t}\ (\textsc{LVar}) \qquad \frac{v \notin t \quad \textsc{SensFree}(t) = \emptyset}{v = t; \top \vdash_{\{v\}} t \leftarrow v}\ (\textsc{RVar})$$

$$\frac{}{\top; \top \vdash_{\emptyset} t \leftarrow t}\ (\textsc{Refl}) \qquad \frac{}{\top; S = S' \vdash_{\emptyset} \mathbb{N}[S] \leftarrow \mathbb{N}[S']}\ (\textsc{Nat})$$

$$\frac{\gamma; \psi \vdash_A \sigma \leftarrow \tau}{\gamma; \psi \wedge S = S' \vdash_A L(\sigma)[S] \leftarrow L(\tau)[S']}\ (\textsc{List})$$

$$\frac{\gamma; \psi \vdash_A \sigma \leftarrow \tau}{\gamma; \psi \wedge S' \leqslant S \vdash_A \texttt{mset}\,[\sigma]\,[S] \leftarrow \texttt{mset}\,[\tau]\,[S']}\ (\textsc{MSet})$$

$$\frac{\gamma; \psi \vdash_A \sigma \leftarrow \tau}{\gamma; \psi \wedge S' \leqslant S \vdash_A !_S \sigma \leftarrow !_{S'} \tau}\ (\textsc{Modal}) \qquad \frac{\gamma; \psi \vdash_A \sigma \leftarrow \tau}{\gamma; \psi \vdash_A \bigcirc \sigma \leftarrow \bigcirc \tau}\ (\textsc{Monad})$$

$$\frac{\gamma; \psi \vdash_A \sigma_2 \leftarrow \sigma_1 \quad \delta; \phi \vdash_B \tau_1 \leftarrow \tau_2}{\gamma \wedge \delta; \psi \wedge \phi \vdash_{A \cup B} \sigma_1 \multimap \tau_1 \leftarrow \sigma_2 \multimap \tau_2}\ (\textsc{Arrow})$$

$$\frac{\gamma; \psi \vdash_A \sigma\,[\alpha/\rho] \leftarrow \tau\,[\beta/\rho] \quad \rho \notin A}{\gamma; \psi \vdash_A \forall \alpha.\, \sigma \leftarrow \forall \beta.\, \tau}\ (\textsc{ForAll}) \qquad \frac{\gamma; \psi \vdash_A \sigma \leftarrow \tau}{\gamma; \psi \vdash \sigma \leftarrow \tau}\ (\textsc{Exit})$$

Figure 4.8: Inference rules defining abduction (the remaining rules are in Appendix B.5); $\textsc{SensFree}(t)$ is the set of free sensitivity variables in t, and the relation $\gamma; \psi \vdash_A \sigma \leftarrow \tau$ is *avoiding abduction*, where the subscript A is a set of type variables to be avoided by quantifiers

Constraint Abduction

The subtyping judgement for DFuzz (of the form $\Phi, \Gamma \models \sigma \sqsubseteq \tau$) depends on a set of constraints Φ over sensitivity and size variables. During synthesis, if we require an expression of type τ, we will always be satisfied to find an otherwise acceptable expression of type $\sigma \sqsubseteq \tau$.

To appropriately apply rule (Id), we must be able to *abduce* the constraint Φ and the type constraints γ under which σ is a subtype of τ. Informally, we need to answer the question: *"what constraints have to be true so that we can use* σ *in place of* τ*?"*

We present a set of inference rules in Figure 4.8—derived by combining an inversion of DFuzz's subtyping rules with a unification procedure—that define an abduction judgement $\gamma; \psi \vdash \sigma \leftarrow \tau$ stating that, if the type constraint γ (which is a conjunction of equalities over type variables and constructors) and the sensitivity constraint ψ hold, then we can use σ in place of τ during synthesis. More formally, we use the following interpretation: if M is a model over type and sensitivity variables such that $M \models \gamma \wedge \psi$, then $\top \models [\![\sigma]\!]_M \sqsubseteq [\![\tau]\!]_M$.

The definition of our abduction judgement depends in part on an auxiliary *avoiding abduction* judgement, written $\gamma; \psi \vdash_A \sigma \leftarrow \tau$. A is a set of type variables that have been constrained by abduction, and appears in the antecedent in rule (ForAll): to ensure the constraints abduced are the most general, we cannot universally quantify over type variables that have already been constrained.

The following theorem guarantees our abduction procedure abduces the most general constraint, which is necessary to ensure we do not miss solutions by over-constraining the proof obligation:

Theorem 4.11 (Abduction most-generality)**.** *If* $\gamma; \psi \vdash \sigma \leftarrow \tau$*, then for all constraints* δ *and* ϕ *such that* $\delta; \phi \vdash \sigma \leftarrow \tau$*, the formulas* $\delta \wedge \phi \Rightarrow \gamma \wedge \psi$ *is valid.*

The proof of Theorem 4.11 is given in Appendix B.6.

This abduction judgement appears in only one inference rule. Given a synthesis state $\langle \phi,\ t\left[\bullet_{\sigma}^{\Omega}\right]\rangle$, rule (Id) lets us replace the wildcard $\bullet_{\sigma}^{\Omega}$ with an identifier (either a function or variable in scope) of type τ under the proof obligation $\gamma \wedge \psi$ if $\gamma; \psi \vdash \tau \leftarrow \sigma$. We also accumulate the obligation that context $\Omega = \{\nu :_1 \tau\}$, as the expression we replace the wildcard with—the identifier ν—is surely 1-sensitive with respect to ν.

Example 4.12 (Abduction)**.** *Consider the last synthesis state in Example 4.7. Replacing* $\bullet_{\mathbb{R} \multimap_{\iota} \mathbb{R}}^{\Omega_1}$ *with the 1-sensitive function* `succ` *using* (Id) *invokes abduction. Rules* (Arrow), (Modal), *and* (Refl) *abduce the constraint* $\psi = 1 \leqslant \iota$ *and the empty unification constraint* $\top$, *indicating that* ι *should be at least the sensitivity of* `succ`.

Satisfiability of Proof Obligations

We have discussed the construction of symbolic context constraints, but not their interpretation. In this section, we present a technique for checking satisfiability of SCCs.

Applying rule (Finish) to a subproblem $\langle \phi,\ p\rangle$ relies on checking satisfiability of ϕ, denoted $\text{Sat}(\phi)$. By construction, the proof obligation ϕ is of the form $\phi_d \wedge \phi_c$, where ϕ_d contains no symbolic context constraints, and ϕ_c is only over symbolic context constraints (following Definition 4.6).

Formula ϕ_d lies in the combined theory of (*i*) non-linear arithmetic over integers and reals extended with ∞ and (*ii*) equality of uninterpreted functions. However, ϕ_c formulas lie in our context expression language, for which there is no default first-order theory. We now demonstrate how to translate a formula ϕ_c into an *equisatisfiable* formula over arithmetic constraints.

Context Interpretations A *model* M is a map from symbolic context variables to *concrete* contexts (containing no sensitivity variables) in the DFuzz

system; consequently, M also maps sensitivity (resp., size) variables to the reals (resp., naturals). A model M *satisfies* a context constraint ϕ_c (denoted $M \models \phi_c$) if ϕ_c is true when evaluated using variable assignments from M.

Example 4.13 (Models)**.** *Consider* $\phi_c \coloneqq (\Omega = \emptyset + 2 \cdot \{x :_k \mathbb{R}\})$*. Let* M *be a model mapping* Ω *to the context* $\{x :_2 \mathbb{R}\}$ *and* k *to 1. Clearly,* $M \models \phi$*, since the right-hand side of the equality reduces to* $\{x :_2 \mathbb{R}\}$ *through context arithmetic.*

Two contexts are equal in our interpretation if (*i*) they contain the same variables and (*ii*) every variable has the same sensitivity and type in both contexts.

Translation To check satisfiability of ϕ_c, we will transform it into a formula ψ_c over sensitivity and size variables. Let $\textsc{Support}(\phi_c)$ denote the *support* of ϕ_c: the set of expression variables that appear *explicitly* in ϕ_c. For example, if $\phi \coloneqq \Omega = \emptyset + 2 \cdot \{x :_k \tau\}$, then $\textsc{Support}(\phi) = \{x\}$. By definition, ϕ_c can only restrict the sensitivities of variables that appear in $\textsc{Support}(\phi_c)$, and so a translation of ϕ_c need only constrain sensitivities of variables in $\textsc{Support}(\phi_c)$.

We will explicitly state the constraint ϕ_c puts on all variables in the support. For each variable in the support, we compute the *symbolic sensitivity* as follows:

Definition 4.14 (Symbolic Sensitivity)**.** *Let* $x \in \textsc{Support}(\phi_c)$*, and let* C *be a symbolic context term. We define the* symbolic sensitivity of x in C under sensitivity substitution δ*—written* $\Delta_x^\delta(C)$*—recursively by:*

1. $\Delta_x^\delta(\emptyset) \coloneqq \langle 0, \top \rangle$
2. $\Delta_x^\delta(\Omega) \coloneqq \left\langle r_x^\Omega, \top \right\rangle$*, where* r_x^Ω *is a fresh sensitivity variable*
3. $\Delta_x^\delta(\{x :_R \tau\}) \coloneqq \langle \delta(R), \top \rangle$

4. $\Delta_x^\delta(C\,[\nu/s]) \coloneqq \Delta_x^{\delta'}(C)$, *where* $\delta' = \delta \circ [\nu/s]$ *and composition is right-associative*

5. $\Delta_x^\delta(C_1 + R \cdot C_2) \coloneqq \langle S_1 + \delta(R) \cdot S_2, \phi_1 \wedge \phi_2 \rangle$, *where* $\Delta_x^\delta(C_1) \coloneqq \langle S_1, \phi_1 \rangle$ *and* $\Delta_x^\delta(C_2) \coloneqq \langle S_2, \phi_2 \rangle$

6. $\Delta_x^\delta(C_1 \oplus C_2) \coloneqq \langle S_1 + S_2, \phi_1 \wedge \phi_2 \wedge (S_1 = 0 \vee S_2 = 0) \rangle$, *where* $\Delta_x^\delta(C_1) \coloneqq \langle S_1, \phi_1 \rangle$ *and* $\Delta_x^\delta(C_2) \coloneqq \langle S_2, \phi_2 \rangle$

   $\Delta_x^\delta(C)$ *is a pair* $\langle S, \phi \rangle$, *where* S *is a sensitivity expression (i.e., an arithmetic combination of sensitivity variables and constants) and* ϕ *is a constraint on sensitivity expressions sufficient to ensure equisatisfiability (see Theorem 4.17).*

We can now lift a constraint over symbolic contexts to a constraint over symbolic sensitivities:

Definition 4.15 (Symbolic Sensitivity Constraint)**.** *Let* $\phi_c = \bigwedge_{i=1}^n C^{l,i} = C^{r,i}$ *be a constraint over symbolic contexts. The associated* symbolic sensitivity constraint—*written* $\Delta(\phi_c)$—*is computed as follows:*

$$\Delta(\phi_c) \coloneqq \bigwedge_{x \in \textsc{Support}(\phi_c)} \bigwedge_{i=1}^{n} S_x^{l,i} = S_x^{r,i} \wedge \phi_x^{l,i} \wedge \phi_x^{r,i},$$

where $\Delta_x^\iota(C^{l,i}) = \left\langle S_x^{l,i}, \phi_x^{l,i} \right\rangle$, $\Delta_x^\iota(C^{r,i}) = \left\langle S_x^{r,i}, \phi_x^{r,i} \right\rangle$, *and* ι *is the empty substitution.*

Example 4.16. *Consider the symbolic context constraint*

$$\phi_c \coloneqq \Omega \oplus 7 \cdot \{x :_k \tau\} = \{x :_{10} \tau\}$$

Then we have

$$\Delta(\phi_c) \coloneqq r_x^\Omega + 7 \cdot k = 10 \wedge (r_x^\Omega = 0 \vee 7 \cdot k = 0)$$

This formula is satisfiable, as witnessed by the model assigning $r_x^\Omega = 10$ *and* $k = 0$*.*

Importantly, this transformation from symbolic context constraints to symbolic sensitivity constraints *preserves satisfiability*. This gives a mechanism for checking $\textsc{Sat}(\phi_c)$ using modern SMT solvers without extensive modifications. This notion is formalized as follows:

Theorem 4.17. *Consider the constraints $\phi_d \wedge \phi_c$. We have $\textsc{Sat}(\phi_d \wedge \phi_c)$ iff $\textsc{Sat}(\phi_d \wedge \Delta(\phi_c))$.*

Theorem 4.17 is proven in Appendix B.7.

Soundness and Relative Completeness

Our inference rules result in a *sound* synthesis algorithm, meaning that if a program p is returned, it is guaranteed to be a solution to the synthesis problem. Formally:

Theorem 4.18 (Soundness). *Let $S = \langle \sigma, C, \phi_k, \phi_s \rangle$ be a synthesis problem, and let p be an expression returned from a sequence of inference rule applications beginning with* (Init) *applied to S and ending with an application of* (Finish)*. Then p satisfies the correctness constraint ϕ_S.*

Given a signature Σ, our search's inference rules generate every production in our expression grammar *except* type-, size-, and sensitivity-abstractions. Thus, our search is *relatively complete*: assuming we have an oracle for satisfiability and termination checking, our search is able to derive any solution to a synthesis problem that lies in the space of expressions with only term-abstractions and primitives from the provided set of components C. We have the following formalism:

Theorem 4.19 (Relative completeness). *Let $S = \langle \sigma, C, \phi_k, \phi_s \rangle$ be a synthesis problem, and let p be a solution of S with no sensitivity-, size-, or type-abstractions. Then there is a sequence of inference rule applications beginning with* (Init) *applied to S and ending with $\langle \phi,\ p \rangle$, with $\textsc{Sat}(\phi)$.*

Functions
`map`
$\forall \alpha, \beta. \forall s. (\alpha \to \beta) \multimap_{2 \cdot s} \texttt{mset}\,[\alpha]\,[s] \multimap_1 \texttt{mset}\,[\beta]\,[s]$
`filter`
$\forall \alpha. \forall s. (\alpha \to \texttt{bool}) \multimap_s \texttt{mset}\,[\alpha]\,[s] \multimap_1 \texttt{mset}\,[\alpha]\,[s]$
`part`
$\forall \alpha, \beta. \forall s, n. \texttt{set}(\alpha)[n] \to (\beta \to \alpha) \multimap_s \texttt{mset}\,[\beta]\,[s] \multimap_2 L(\langle \alpha, \texttt{mset}\,[\beta]\,[s] \rangle)[n]$
`count`
$\forall \alpha. \forall k. \texttt{mset}\,[\alpha]\,[k] \multimap_1 \mathbb{R}$
Privacy Mechanisms (type `DB` is an alias for `mset` [`row`] [∞])
`LapMech`
$\forall k. (\texttt{DB} \multimap_k \mathbb{R}) \to \texttt{DB} \multimap_{k \cdot \epsilon} \bigcirc \mathbb{R}$
`ParaMech`
$\forall \alpha. \forall n, k. L(\alpha)[n] \to (\texttt{row} \to \alpha) \to (\texttt{DB} \multimap_k \mathbb{R}) \to \texttt{DB} \multimap_{k \cdot \epsilon} \bigcirc L(\langle \alpha, \mathbb{R} \rangle)[n]$
`ExpMech`
$\forall k. D \to (\texttt{DB} \multimap_k D \to \mathbb{R}) \to \texttt{DB} \multimap_{k \cdot \epsilon} \bigcirc D$

Table 4.2: Examples of functions in our synthesis domain and privacy mechanisms; `part` returns a *list* of key-value pairs, whose distance metric is the *sum* of element-wise distances for lists of the same length, and ∞ otherwise

The proof of Theorems 4.18 and 4.19 are provided in Appendices B.2 and B.3.

4.4 Applications of Privacy-Aware Synthesis

In Section 4.3, we described how to solve a synthesis problem $\langle \sigma, C, \phi_k, \phi_s \rangle$. In this section, we will present two applications of our synthesis algorithm. First, we focus on synthesizing queries over sensitive databases, before turning synthesis towards the problem of combining private operators to construct a differential privacy mechanism from scratch. Finally, we present an analysis of the security model of using our synthesis algorithm for sensitive applications.

Differentially Private Data Analysis

Data analysis tasks are the main application of differential privacy, and we are interested in using synthesis to aid end-users in interfacing with sensitive data. Rather than reasoning about the complex interactions of sensitivities and information leakage, data analysts should be able to specify a DP query by providing only a *semantic specification* of the query and their *privacy budget*.

We model datasets as multisets of elements of type `row`, a tuple indexed by *keys*. To facilitate the construction of efficient and expressive queries, we instantiate our set of components C with the following sets:

1. A set of higher-order combinators—`map`, `filter`, etc.—similar to the components used by BIGλ (Chapter 3).
2. Aggregation operators—`sum`, `max`, `count`, and `average`—over multisets.
3. Standard arithmetic operations and Boolean predicates.
4. Dataset-dependent constants and projections to extract fields from rows.

Table 4.2 shows four of the combinators in our data-analysis domain. All are standard, but their type annotations provide a detailed view of their underlying privacy semantics. A strength of synthesis is that it can shield users from having to reason about such complex types.

Privacy Mechanisms

To answer queries in a differentially private manner, dataset maintainers use *privacy mechanisms* that return sensitive information in provably safe ways. Privacy mechanisms take in some specification of a query (and relevant supporting information) and construct a differentially private function from the dataset to the desired output domain. Data analysts can expect to be restricted to interacting with sensitive data via a small set of

mechanisms supported by the dataset maintainer. We therefore focus on synthesizing *inputs* to privacy mechanisms.

We will briefly discuss the usage of the privacy mechanisms in Table 4.2. To simplify the presentation, we assume the components C and the privacy parameter ϵ are determined *a priori* by the target dataset. As a further simplification, we use *input-output examples* to specify the desired query semantics: if a user presents a set E of pairs $\langle i, o\rangle$, we assume they desire a program p that agrees on all $\langle i, o\rangle$ pairs, i.e. $p_d(i) = o$. Note that, while input-output examples are a straightforward way to encode desired semantics, what follows can be adapted for other forms of specifications.

Laplace Mechanism Recall that the Laplace mechanism is applied to programs with real-valued outputs. When given a set of examples from databases to *real numbers*, we can use the Laplace mechanism to reduce our synthesis problem to a function of type $\texttt{mset}\,[\texttt{row}]\,[\infty] \multimap_k \mathbb{R}$.

More precisely, given a set of input-output examples E with inputs of type $\texttt{mset}\,[\texttt{row}]\,[\infty]$ and outputs of type $\mathbb{R}$,and a sensitivity budget b, we can construct the privacy-aware synthesis task:

$$\mathcal{S} = \langle \texttt{mset}\,[\texttt{row}]\,[\infty] \multimap_k \mathbb{R},\ C,\ k \leqslant b,\ \forall \langle i, o\rangle \in E.\, p(i) = o\rangle$$

Given a program p that is a solution to $\mathcal{S}$, a dataset maintainer can apply the Laplace mechanism (via application of `LapMech`) to answer the user's query in a privacy-preserving manner.

Parallel Composition Sequential composition of privacy mechanisms incurs a privacy cost of the *sum* of the costs of the two mechanisms. This is often not the best way to guarantee DP. *Parallel composition* (McSherry, 2009) is a property of DP that allows us to evaluate an ϵ-DP function on arbitrarily-many disjoint datasets with a total privacy cost of ϵ. Following PINQ (McSherry, 2009), we implement parallel composition by allowing

users to *partition* the multiset into disjoint subsets before analyzing each partition independently.

When given a set of examples E of type $\texttt{mset}\,[\texttt{row}]\,[\infty] \times \langle \texttt{key}, \mathbb{R} \rangle$—converting databases to real values indexed by *keys*—a sensitivity budget b, *a set of keys* K, and a projection $\pi : \texttt{row} \multimap K$ selecting the key for every row, we build the synthesis problem

$$S = \langle \texttt{mset}\,[\texttt{row}]\,[\infty] \multimap_k \mathbb{R},\ C,\ k \leqslant b,\ \forall \langle i, o \rangle \in E.\, \texttt{part}(\pi, K, p, i) = o \rangle\,,$$

where the constraint ensures per-partition correctness. A dataset maintainer given K and a function p that is a solution to S can construct the query $\texttt{ParaMech}\ \pi\ K\ p$ to answer the user's request while using only ϵ of the privacy budget.

Parallel composition is more involved than Laplace mechanism, as now a user must also provide a set of keys along with their input-output examples E. This additional information is required to preserve privacy: ParaMech only evaluates p on partitions constructed by projecting onto a particular $k \in K$. Restricting evaluation to provided keys ensures the presence or absence of a key in the dataset is revealed in a privacy-preserving manner.

Exponential Mechanism We are often interested in performing computations whose output is some *categorical type*, e.g., computing the most common medical condition, yet the mechanisms introduced so far are restricted to numerical outputs. To handle categorical outputs, we use the *exponential mechanism* (McSherry and Talwar, 2007). Instead of adding noise to the output, the exponential mechanism expects the user to provide a *utility function*. This function assigns a real-valued *utility* (or quality) to each output $d \in D$ given an input. The type of such a utility function is $\texttt{mset}\,[\texttt{row}]\,[\infty] \multimap_k D \multimap_\infty \mathbb{R}$. We leave the details of converting such a function to an ϵ-DP query to McSherry and Talwar (2007).

If a user provides a set of examples E of type $\texttt{mset}\,[\texttt{row}]\,[\infty] \times D$ and a sensitivity budget b, we can construct the synthesis problem

$$S = \left\langle \texttt{mset}\,[\texttt{row}]\,[\infty] \multimap_k D \multimap_\infty \mathbb{R},\ C,\ k \leqslant b,\ \phi^{e}_{E,D} \right\rangle,$$

where the functional specification is

$$\phi^{e}_{E,D} \coloneqq \forall \langle i, o \rangle \in E.\, \forall d \in D.\, d \neq o \Rightarrow p(i, o) > p(i, d)$$

encoding the semantics that o, the *desired output*, has the highest utility of all elements in D.

Unlike the previous mechanisms, the synthesis problem encodes a function whose semantics are *not the same* as the desired query. The requirement that a user builds a utility function that captures their desired semantics while still being appropriately sensitive has prevented full adoption of the exponential mechanism in data analysis. Application of our technique can remove this burden.

Mechanism Design

While data analysts are the primary *users* of differential privacy, much research is focused on *privacy mechanism design* and implementation. Combining privacy primitives to produce new mechanisms is non-trivial: proving even simple mechanisms such as `report-noisy-max` require complicated *coupling arguments* (Albarghouthi and Hsu, 2018b). Fortunately, the particular combination of features provided by DFuzz —dependent pattern-matching, precise types, the probability monad, and recursion—allow for the typing of full mechanisms such as `k-means` and `idc` (Figures 4.3 and 4.4).

The original presentation of DFuzz (Gaboardi et al., 2013) presents three implementations of iterative privacy mechanisms, each of which

use an input argument of known size (such as the precise natural `iter` with type $\mathbb{N}[i]$ in `k-means`) to bound the number of iterations. By bounding the number of iterations, the applications of privacy primitives also becomes bounded, which is reflected in the sensitivity of the mechanism, e.g. `k-means` is $3i\epsilon$ sensitive in the argument `data`.

Unlike the applications of synthesis for data analysis, constructing a synthesis problem whose solution is a privacy mechanism is straightforward. The following example demonstrates this using `k-means`.

Example 4.20. *Suppose a mechanism designer wants to synthesize a version of* `k-means` *by carefully composing applications of the cluster-updating function*

$$\textit{k-step} : \forall k.\, L(\langle \mathbb{R}, \mathbb{R} \rangle)[k] \to \textit{mset}\,[\langle \mathbb{R}, \mathbb{R} \rangle]\,[\infty] \multimap_{3\epsilon} \bigcirc L(\langle \mathbb{R}, \mathbb{R} \rangle)[k]$$

that, when given a list of cluster centers and a database, updates the list of cluster centers, accumulating a privacy cost of 3ϵ*. Instead of manually reasoning about the recursion required, the mechanism designer can instead construct the synthesis problem*

$$S = \langle \sigma, \{\textit{k-step}\}, \top, \forall \langle i, o \rangle \in E.\, p(i) = o \rangle\,,$$

where σ *is the type of* `k-means` *given in Section 5.1 and* E *is a set of input–output examples. The full version of the implementation sketch given for* `k-means` *in Figure 4.3 is a solution to* S*.*

Utility of Synthesis Solutions

In this section, we frame instantiations of our technique as a mechanism for finding privacy-aware solutions to a functional specification ϕ_s, usually provided as a set of input-output examples. These instantiations provide a mechanism for easily maintaining privacy while enabling access to sensitive data. However, in most real-world applications, users also desire

accuracy. For randomized programs, like those we synthesize, accuracy amounts to the intuition that we arrive at close to the right answer most of the time (Dwork and Roth, 2014). As users define "the right answer" by providing ϕ_s, we expect users desire a program p that *maximizes* the probability that $p \models \phi_s$.

Unfortunately, proving accuracy of randomized programs is a challenging task. Most proofs are constructed by hand, and as such are tailored to the algorithm of interest. While there exist program logics for reasoning about accuracy (Barthe et al., 2016b), automations of said logics (*i*) are so slow as to dwarf the cost of synthesis (see Section 4.6), and (*ii*) themselves reduce to synthesis (Chapter 5, also (Smith et al., 2019)). Considering accuracy in addition to privacy therefore poses a significant challenge to program synthesis, which necessitates our use of the weak notion of *deterministic satisfaction* (Definition 4.3) to ensure programs have some utility to an end user.

Usage Scenario and Privacy Guarantee

Here we propose a usage scenario based on the data analysis mechanisms in this section and consider the resulting privacy guarantee. The following discussion can be adapted to the mechanism design, although we expect mechanism designers will not be interacting with real-world sensitive information. We will speak of two distinct entities: (*i*) the dataset maintainer (**DM**), and (*ii*) the user (**U**).

Dataset Maintainer The dataset maintainer **DM** controls access to a dataset D containing sensitive information. For all authorized users (including **U**), **DM** maintains a privacy budget, the sum of which is the total privacy budget for D. **DM** makes public as much of the *semantic structure* of D as possible without leaking information. This includes the database schema, descriptions of fields, and *numerical data ranges* when they hold

for every possible entry in that field. For instance, a field containing a *percentage* will always have values in the range $[0, 100]$. However, a field recording *age* is (theoretically) unbounded, and so any range published by **DM** will leak information about the contents of D. Lastly, **DM** makes public Σ, a set of functions whose implementation is trusted and projections with sensitivities derived from the provided data ranges (e.g., a function `grade` of type $\texttt{row} \multimap_{100} \mathbb{R}$ that selects a student's final grade from the relevant row).

As our technique also produces sensitivity proofs, the dataset maintainer can easily verify that executing q on D will be $\mathrm{c}\epsilon$-DP. When **DM** receives a c-sensitive query q with a proof of sensitivity from a user, they (*i*) verify the proof of sensitivity, (*ii*) evaluate q on D, and (*iii*) decrement the privacy budget of **U** appropriately.

User To query D, the user **U** can synthesize a query and resulting proof of sensitivity. At a minimum, this requires **U** to construct (*i*) input-output examples and (*ii*) a maximal acceptable privacy cost. Input-output examples can be hand-crafted to appropriately disambiguate the desired query, or be randomly-generated inputs and the corresponding outputs. The maximal privacy cost is up to **U**, and might depend on total privacy budget left and the possible value of the query result. Using these inputs, **U** can use a privacy mechanism to instantiate a synthesis problem and send the resulting query and associated proof of privacy to **DM** for consideration.

For numerical fields where **DM** has not provided a data range, **U** can fine-tune Σ by providing best-guess data ranges. These ranges can be converted to projections whose sensitivity is the size of the range using the clipping technique in Airavat (Roy et al., 2010), lowering the privacy cost.

More realistically, we might expect the user **U** to be multiple individuals fulfilling different roles. For example, one could have a *domain expert* that

constructs the relevant best-guess data ranges and random examples, and an *analyst* that computes the appropriate outputs to specify some query.

Effective Privacy Guarantee In the scenario above, queries are synthesized *independently* of a sensitive dataset, and are built exclusively from trusted code and clipped projections. Numerical data ranges are only provided if they hold for *every possible entry*, and so the only means of egress for sensitive information is through noisy query answers released by the dataset manager. A dataset manager is therefore able to maintain the privacy guarantee on D, as each release of information is $c\epsilon$-DP with a known c.

4.5 Implementation Details

We implemented our technique in a tool called Zinc, comprising ~4500 lines of OCaml that interface with the Z3 SMT solver (de Moura and Bjørner, 2008) for satisfiability checking.

Zinc implements a fair scheduling of the inference rules in Section 4.3, so that any rule that is applicable will eventually be applied. The application order of the rules is determined by a heuristic function that assigns costs to subproblems. Zinc takes as input a set of input–output examples and an upper bound on the budget. In the data analysis case, the choice of privacy mechanism to use is determined by the types of the provided examples.

Determinization Zinc maintains a work-list of candidate partial solutions, and explores candidates in increasing heuristic order. When considering a candidate, all relevant inference rules are applied to generate new candidates. To prevent the search space from exploding, Zinc utilizes two common techniques from the type-directed synthesis literature to

minimize the number of term-, type-, and sensitivity-applications. The rules for introducing such applications either introduce free variables or increase the number of wildcards, both of which expand the search space unnecessarily.

To limit term applications, ZINC only enumerates terms in β-normal form, rather than the equivalent β-expanded terms. That is, ZINC will consider $f\ x$, but not $(\lambda y.\, f\ y)\ x$, despite the two terms being equivalent. Empirically, such terms are smaller and more interpretable. ZINC achieves this by only applying the rule (APP) to a wildcard whose goal type is a *function type*, e.g. $\sigma \multimap_k \tau$.

ZINC limits type and sensitivity applications by only applying rules (TAPP), (SIZEAPP), and (SENSAPP) when necessary to enable an application of rule (ID). For example, in Example 4.10, to replace $\bullet^{\Omega}_{\mathtt{mset}[\mathtt{row}][\infty] \multimap_2 \mathbb{R}}$ with the function `count` of type $\forall \beta.\, \forall k.\, \mathtt{mset}\ [\beta]\ [k] \multimap_1 \mathbb{R}$, we must first apply (TAPP) and (SIZEAPP) to introduce quantifiers in the goal type. Rather than expanding the goal type unnecessarily, ZINC greedily attempts to apply (ID), and defaults to applying (TAPP) and the like when abduction fails due to a universal quantifier in the type of the term being substituted.

Pruning Strategy In Section 4.3, we give the invariant that any synthesis state $\langle \phi, p \rangle$ that has an unsatisfiable proof obligation ϕ can be pruned without losing relative completeness. In our original strategy we eagerly called the SMT solver to check satisfiability for every state. While every call typically took less than a second, the overhead of SAT checking far outweighed the benefits of pruning, resulting in an impractical algorithm.

Instead of calling the SMT solver on every subproblem, we implemented a simple constraint solver that only performs *unit propagation*. The solver looks for conjuncts of the form $x = S$, where x is a variable and S is a sensitivity expression, and replaces all occurrences of x by S in the conjunction. If unit propagation fails to prove satisfiability, we use the number of

remaining sensitivity variables as a heuristic quantitative measure for *how likely it is for the formula ϕ to be unsatisfiable*. We use this heuristic value to order the states for exploration: intuitively, the more variables we cannot easily eliminate with constant propagation, the harder it will be to satisfy the constraints.

Constraint Solving Once Zinc has found a *closed* candidate program that satisfies ϕ_s, it attempts to prove that the program satisfies the budget constraints ϕ_k. Theorem 4.17 provides a mechanism for transforming the proof obligation to an equisatisfiable formula in the undecidable theory of mixed real and integer non-linear arithmetic. Fortunately, in our setting, such constraints are reliably checkable (de Amorim et al., 2014). If necessary, Zinc can *relax* the obligation by treating integers as reals, placing the constraints in the decidable theory of *real closed fields*.

4.6 Evaluation

Our evaluation seeks to answer the following research questions:

RQ1 Is Zinc able to synthesize interesting differentially private queries in a reasonable time?

RQ2 Is sensitivity-directedness useful in guiding the synthesis process?

RQ3 Can Zinc synthesize full privacy mechanisms?

Benchmarks Our first goal with benchmarking was to mimic a setting where a data analyst wants to query a dataset with sensitive information through a DP-enforcing system. We collected 4 real-world datasets (Table 4.3) containing personal information that would warrant DP. For example, the Adult Income dataset (Lichman, 2013) is census data and contains data on gender, ethnicity, and income.

Data Analysis Benchmark
Adult Income (1.5, 5, 8)
from 1994 census data, contains info on gender, ethnicity, profession, etc.
1 number of women who work > 40 hrs a week
2 cumulative years of education in military
3 number of people who make > 50k in trade
4 most common gender working in local government
5 population per ethnicity
6 profession with highest total work hours
7 number of people making > 50k per branch of government
Student Alcohol Consumption (1, 4, 8)
Portuguese schools student info on grades, family life, and weekend/weekday alcohol consumption
8 # of students who drink on the weekend and pay for extra classes
9 average final grade of students attending for rep. reasons
10 average weekend alcohol consumption per address type
11 family relationship status with highest grade
12 average final grade of students not drinking on the weekend
13 total absences per attendance reason
14 most common address type for those with poor family relations
Student Performance (2, 4, 6)
same as above, with performance, resource usage, participation, and personal info
15 number of students with satisfied parents and many absences
16 performance level with the highest average participation
17 average resource usage per parent satisfaction
18 hands raised by students with low discussion activity
19 average grade in the low performance bracket
20 are parents satisfied when their child is absent a lot?
21 total resource usage per performance bracket
COMPAS (3, 6, 8)
ProPublica data with info on criminal history and COMPAS *scores, including risk of recidivism*
22 number of elderly with high risk of violence
23 average failure to appear for youths with no priors
24 ethnicity with highest average recidivism
25 total priors per gender
26 # of people with many juvenile felonies and high recidivism risk
27 average recidivism per ethnicity
28 age category with the most priors

Iterative Privacy Mechanism Benchmark	$\mathtt{i}$ Represents...	Sens. Budget
`k-means` - compute $\mathtt{k}$ cluster centers from data	# of updates	$3 \cdot \mathtt{i} \cdot \epsilon$
`idc` - answers query set by iteratively constructing dataset	# of updates	$2 \cdot \mathtt{i} \cdot \epsilon$
`cdf` - given list of buckets, count elements per bucket	# of buckets	$\mathtt{i} \cdot \epsilon$

Table 4.3: Datasets Zinc was tested over, including (*i*) the average number of examples per benchmark, (*ii*) the maximum number of rows in an example, and (*iii*) the number of projections added to the signature Σ; for iterative mechanisms, ϵ is a fixed privacy cost

Benchmark	Sensitivity-Directed Time	Speedup	Count	Baseline Time	Count	$\lvert p \rvert$	$\lvert \phi \rvert$
Adult Income							
1 *Lap.*/1ϵ	2.12	1.1x	16,825	2.36	16,721	31	11
2 *Lap.*/20ϵ	3.43	1.9x	26,506	6.65	39,365	33	12
3 *Lap.*/168ϵ	0.04	4.3x	601	0.17	1,722	26	7
4 *Exp.*/1ϵ	1.99	4.2x	11,610	8.44	32,299	26	8
5 *PC*/1ϵ	0.004	2.3x	36	0.009	75	24	9
6 *Exp.*/168ϵ	3.77	8.7x	20,033	32.97	111,466	31	8
7 *PC*/1ϵ	3.53	1.4x	11,856	4.96	13,072	48	21
Student Alcohol Consumption							
8 *Lap.*/1ϵ	0.42	2.3x	4,495	0.95	7,561	25	8
9 *Lap.*/20ϵ	5.03	1.5x	37,539	7.67	45,438	33	12
10 *PC*/5ϵ	3.28	1.4x	12,300	4.72	12,558	44	20
11 *Exp.*/20ϵ	5.27	4.1x	28,503	21.69	81,563	34	12
12 *Lap.*/100ϵ	6.26	3.0x	45,648	18.53	90,664	35	12
13 *PC*/100ϵ	3.35	1.4x	12,549	4.70	12,379	44	20
14 *Exp.*/5ϵ	0.125	9.9x	1,030	1.24	6,161	24	6
School Performance							
15 *Lap.*/1ϵ	0.087	1.7x	1,210	0.15	1,529	26	7
16 *Exp.*/100ϵ	9.67	15.3x	46,069	147.8	432,903	38	12
17 *PC*/100ϵ	5.53	1.1x	19,504	6.19	16,525	44	20
18 *Lap.*/100ϵ	8.11	6.3x	56,131	51.03	213,231	37	12
19 *Lap.*/100ϵ	4.23	1.3x	30,563	5.42	32,167	37	12
20 *Exp.*/1ϵ	1.91	4.1x	11,499	7.77	29,824	26	8
21 *PC*/100ϵ	38.95	2.2x	98,935	85.37	143,448	51	25
COMPAS							
22 *Lap.*/1ϵ	0.015	1.6x	227	0.024	365	17	6
23 *Lap.*/10ϵ	155.21	1.4x	813,878	219.78	858,686	36	15
24 *Exp.*/10ϵ	0.98	2.3x	7,254	2.27	12,603	26	9
25 *PC*/15ϵ	0.042	13.1x	329	0.55	2,110	39	15
26 *Lap.*/1ϵ	7.37	1.3x	55,240	9.38	55,157	33	11
27 *PC*/10ϵ	0.036	9.7x	284	0.35	1339	39	15
28 *Exp.*/15ϵ	0.745	1.9x	6,184	1.41	6897	23	6
Iterative Privacy Mechanisms							
`k-means`	0.278	3.5x	822	0.956	2,100	21	26
`idc`	3.491	2.82x	4440	9.845	10654	40	40
`cdf`	-	N/A	-	-	-	32	-

Table 4.4: Results of evaluating ZINC; benchmark descriptions specify the privacy mechanism and budget, and for all experiments, we report (*i*) the CPU time needed by ZINC, (*ii*) the speedup in time compared to the baseline, (*iii*) the number of programs explored, (*iv*) the size of the solution in AST nodes, and (*v*) the size of the proof obligation in conjuncts

There is no existing set of queries over the chosen datasets (or any other dataset) that are (*i*) differentially private and (*ii*) designed to stress synthesis tasks. Therefore, for every dataset, we created a number of benchmark queries (data analysis benchmarks in Table 4.3) that are designed to extract interesting information, be of varying complexity, and exercise the privacy mechanisms from Section 4.4. We discuss the selection of benchmark datasets and queries further in Section 4.6.

Our second goal was to explore the synthesis of full privacy mechanisms. We chose three mechanisms whose privacy guarantees can be verified by DFuzz (iterative privacy mechanism benchmarks in Table 4.3). These mechanisms represent a large portion of the case studies presented in DFuzz (Gaboardi et al., 2013), and are based on real-world privacy mechanisms found throughout the literature. These mechanisms take advantage of every feature of DFuzz our synthesis algorithm supports, including recursion, the probability monad, and dependent pattern-matching.

Experimental Setup We have two instantiations of Zinc:

1. **Sensitivity-directed Zinc:** Our primary interest is the *sensitivity-directed* strategy, where Zinc utilizes symbolic context constraints to direct the search, as described in Section 4.5.

2. **Baseline (type-directed):** To contrast with the sensitivity-directed strategy, we built a *baseline* version of Zinc that is type-directed in the style of existing type-directed synthesis tools (Osera and Zdancewic, 2015; Feser et al., 2015; Smith and Albarghouthi, 2016). The baseline searches the space of programs in ascending size order, and does not exploit the generated constraints to guide the search. The choice to construct a baseline, rather than use an existing tool, is discussed in Section 4.6.

For each strategy, we ran Zinc on each benchmark with a timeout limit of 5 minutes. Table 4.4 shows the results. All times reported are in seconds.

RQ1: Synthesis Time Consider the results in Table 4.4. Overall, the results demonstrate that our technique can synthesize non-trivial differentially private computations over real datasets in a small amount of time. In all benchmarks, our sensitivity-directed technique was able to discover a solution, and in most benchmarks synthesis terminates in under 10 seconds.

The programs synthesized by Zinc are non-trivial, comprising functions with complex types and involving advanced privacy mechanisms. For example, let us consider the solution to benchmark 23, which takes as input three example datasets of three rows each:

```
let bm23 x = avg (map f m) + Laplace(1/ϵ)
    where f = failure-to-appear
    where m = filter (fun y -> age-category(y) == "young") p
    where p = filter (fun z -> priors(z) == none) x
```

The constants `"young"` and `none` are instantiated from the schema of the COMPAS dataset. The solution involves 3 composed higher-order functions, several projections and comparisons, and an aggregation, and is a 10ϵ-DP function (as the projection `failure-to-appear` maps on a scale from 0 to 10). Note program size $|p|$ reported in Table 4.4 contains sensitivity annotations, which we elide here for clarity. The proof obligation (also elided) is satisfiable, and consists of 15 conjuncts.

RQ2: Sensitivity-Directed Synthesis To answer **RQ2**, we compare the synthesis time of the sensitivity-directed configuration and the baseline configuration. Ignoring benchmarks with approximately comparable performance (difference in synthesis time $\leqslant 2$ seconds), the *average speedup* afforded by incorporating sensitivity into the search is ~3.9x, and the maximum speedup is 15.3x. This significant improvement is a clear indication of the importance of the sensitivity-directed strategy to our algorithm's efficiency.

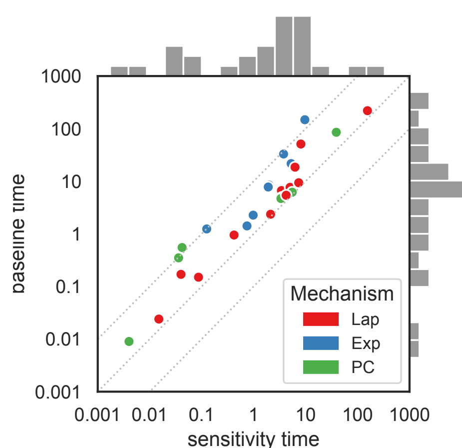


Figure 4.9: Performance comparison between Zinc and baseline, where each benchmark is a pair in log-space comparing sensitivity-directed and size-directed performance; top, middle, and bottom dotted lines are the 10x, 1x, and 0.1x levels of the efficiency gradient, and marginals projected on the border

Consider benchmark 18: here, sensitivity-directed synthesis considers 56,131 programs before discovering the solution, while the baseline technique requires 213,231 programs. Similar patterns are seen across our benchmarks. See Figure 4.9 for a clearer picture of the distribution of times across benchmarks.

The sensitivity-directed implementation consistently outperforms the baseline. Most benchmarks lie between the 10x and 1x efficiency lines in Figure 4.9, although there are several *above* the 10x line. Note the log-scale on the graph: the benchmark furthest above this line boasts a 30x improvement. This distribution of benchmarks indicates that sensitivity-directed synthesis is an improvement over the baseline.

These improvements in the sensitivity-directed synthesis over the baseline come despite the fact that Zinc does not explicitly prune the search space. As the primary difference in implementation is the inclusion of

```
let cdf buckets data = match buckets with
    | [] -> return []
    | x :: xs[i] ->
        let-draw count = ●◯ℝ in
        let-draw rest = ●◯L(ℝ)[i] in
            ●◯L(ℝ)[i+1]
```

Figure 4.10: Partial synthesis problem for `cdf` (Gaboardi et al., 2013); sensitivity-directed finishes in 190s, while size-directed times out.

the constant propagation heuristic (Section 4.5) in the sensitivity-directed approach, we can infer that the reordering of candidate solutions given by the heuristic effectively directs the search towards programs that are *likely* to have satisfiable constraints, and thus be solutions to the synthesis problem. The effectiveness of the heuristic is aided by the fact that, even with relatively simple higher-order combinators, such as `map` and `filter`, the structure of the queries places considerable restrictions on the sensitivity of the input functions.

RQ3: Privacy Mechanism Synthesis Consider privacy mechanism results in Table 4.4. ZINC was able to synthesize `k-means` and `idc` completely in under 4 seconds, but timed out while synthesizing `cdf`. In the two benchmarks that terminated, sensitivity-directedness contributed an average ~3.1x improvement in synthesis speed. The benchmarks that terminate are quite intricate (recall the implementation sketches of the terminating benchmarks in Figures 4.3 and 4.4), as they contain recursion, the manipulation of probability distributions, and dependent pattern-matching.

Note that ZINC timing out on a benchmark does not mean ZINC is unhelpful to a mechanism designer. Rarely is an expert in differential privacy synthesizing a mechanism from scratch. To explore the utility of ZINC for *partial program synthesis*, we manually constructed a solution to the `cdf` benchmark (also taken from Gaboardi et al. (2013)) and introduced

four *holes* by replacing whole expressions with wildcards. By varying the amount (from 1 to 4) and location of holes, we construct a total of 16 partial program synthesis benchmarks. The sensitivity-directed approach finds completions for the holes in all but one benchmark (in Figure 4.10), while the size-directed approach times out on two (Figure 4.10 plus one more). When both terminate, however, the sensitivity-directed strategy has a reduction in efficiency of ~0.94x. As 10 benchmarks terminate in under 1 second, and 2 more terminate in under 10 seconds, this tradeoff is likely worthwhile to be able to fill in one more partial program.

Discussion of Design Choices

Choice of Baseline Our use of a baseline version of ZINC, as opposed to an existing synthesis tool, is necessitated by the limitations in the implementations of existing type-directed synthesis algorithms (e.g. Myth (Osera and Zdancewic, 2015), Myth2 (Frankle et al., 2016), Bigλ (Smith and Albarghouthi, 2016), and λ^2 (Feser et al., 2015)) and the complexity of the DFuzz type system. We found no tool that was amenable to modification to handle the combination of recursion, dependent pattern-matching, the probability monad, and sensitivity annotations.

Selection of Benchmarks Differential privacy is usually benchmarked for (*i*) accuracy of a particular mechanism (Proserpio et al., 2014; Erlingsson et al., 2014; Johnson et al., 2018b; Roy et al., 2010), (*ii*) scalability (Proserpio et al., 2014; Narayan and Haeberlen, 2012; Roy et al., 2010), or (*iii*) case studies highlighting features of a particular system (McSherry, 2009). None of these cases produce large numbers of queries for testing synthesis. Program synthesis benchmarks on tables and databases (Feng et al., 2017) focus on tasks that are not differentially private.

In light of these limitations, we constructed a set of benchmarks to evaluate ZINC. The benchmark datasets were selected because they have

appeared in works on privacy and bias (Feldman et al., 2015; Larson et al., 2016; Datta et al., 2017). The queries cover a range of query complexities and instantiations of our technique, but are motivated by existing analysis when possible (as in the COMPAS recidivism dataset (Larson et al., 2016)).

Sensitivity For each of our benchmark queries, we provide a sensitivity budget of the *minimal sensitivity* necessary to compute the desired query, which is derivable from the database schema and the desired semantics of the benchmark query. We do not expect the user to always be able to explicitly compute the appropriate sensitivity upper-bound, but in some cases real-world considerations—e.g., the remaining privacy budget—can inform the choice of sensitivity bound.

Accuracy/Privacy Tradeoff This work presents mechanisms that add randomness to the output scaled *only* by the privacy parameter ϵ. Often, randomness is dependent on the sensitivity of the query in addition to ϵ (Dwork and Roth, 2014). This allows the mechanism to enforce a stronger privacy guarantee: ϵ-DP instead of $(c \cdot \epsilon)$-DP. Our technique still applies: low sensitivity now represents higher accuracy, instead of a cheaper privacy cost.

4.7 Related Work

Type-directed Synthesis Our contributions are inspired by works on type-directed synthesis. Compared to Myth (Osera and Zdancewic, 2015; Frankle et al., 2016), λ^2 (Feser et al., 2015), and BIGλ (Smith and Albarghouthi, 2016), which use a Hindley-Milner type system, we use a richer type system that adds linear and dependent types to aid in synthesizing programs to meet privacy constraints. Polikarpova et al. (2016) perform synthesis over the powerful refinement type system of liquid types (Ron-

don et al., 2008). However, our techniques are incomparable, since liquid types (as defined and used) cannot reason about probabilistic hyperproperties like differential privacy.

Synthesis for Data Manipulation The main target of DP in general, and this work in particular, is data analysis. Our work follows the tradition of program synthesis for various data-manipulation tasks (Gulwani et al., 2012; Le and Gulwani, 2014; Polozov and Gulwani, 2015; Le and Gulwani, 2014; Yaghmazadeh et al., 2017; Zhang and Sun, 2013; Wang et al., 2017b,a; Smith and Albarghouthi, 2016; Miltner et al., 2018). Perhaps the most closely related work to ours is Bigλ (Smith and Albarghouthi, 2016), also presented in Chapter 3, which targets data-analysis programs composed of higher-order combinators like *map* and *reduce*. Our instantiation of our technique extends these ideas to a DP setting. Additionally, our work can be applied to SQL synthesis (Zhang and Sun, 2013; Wang et al., 2017a; Yaghmazadeh et al., 2017) under DP constraints, as we discuss below.

Differential Privacy Systems Systems enforcing differential privacy on user queries are defined over languages of higher-order combinators (McSherry, 2009; Roy et al., 2010; Proserpio et al., 2014) or forms of SQL queries (Johnson et al., 2018b; Narayan and Haeberlen, 2012). Our presented approach, based on the DFuzz type system, is general to apply to a wide range of settings. We instantiated our algorithm with a language of higher-order combinators. SQL queries can also be captured with *join* operators, like the one used in PINQ (McSherry, 2009).

Work on DFuzz includes the construction of metric-preserving semantics (de Amorim et al., 2017) and a type-checking algorithm (de Amorim et al., 2014). Type-checking DFuzz programs is non-trivial: the natural top-down approach requires context splitting, which necessitates a search over sensitivity terms. Our approach avoids this search by using SCCs, while the work of de Amorim et al. (2014) extends DFuzz sensitivity expressions.

Recently, constraint-based program synthesis techniques have been applied to the problem of proving differential privacy of advanced differentially private algorithms (Albarghouthi and Hsu, 2018b). There, the authors use a heavy-weight logical encoding of the space of proofs of ϵ-DP to discover complex proofs using *coupling arguments*. We utilize DFuzz to compute program sensitivities in a more lightweight fashion to ease synthesis.

5 AUTOMATING PROOFS OF HIGH-PROBABILITY GUARANTEES

The *utility* of a differential privacy mechanism is measured in terms of *accuracy*, which, due to the probabilistic nature of DP, is phrased as a high-probability guarantee: for all program inputs, an output sampled from the final distribution satisfies some accuracy condition φ except with some probability β. While simple to state, accuracy guarantees pose challenges for automated verification: current techniques (surveyed by Baier et al. (2018) and Katoen (2016)) focus on more tractable models of probabilistic programs, and are mostly restricted to closed programs with fixed inputs and finite state spaces.

In this chapter we present an automated technique for proving high-probability guarantees. Our approach is based on *trace abstraction* (Heizmann et al., 2009, 2010, 2013; Farzan et al., 2013), a proof technique that (*i*) represents a program P by a language $\mathcal{L}(P)$ of execution traces through the CFG, and (*ii*) proves every trace is correct by covering $\mathcal{L}(P)$ with proof automata computed via computed via *predicate abstraction* (Graf and Saïdi, 1997) or *Craig interpolation* (McMillan, 2006). To extend this to the probabilistic setting, we prove that individual traces τ satisfy φ except with some failure probability β_τ, and then generalize the proof into an automata. P then satisfies φ if every trace is correct, and the accumulated failure probability $\sum_{\tau \in \mathcal{L}(P)} \beta_\tau$ is less than β.

The most technically intricate step in this process is proving a trace τ correct. We follow the design principle of *reducing probabilistic reasoning to non-deterministic reasoning* (Barthe et al., 2016c, 2012, 2014; Hsu, 2017) to enable the application of classical verification techniques to probabilistic traces. Specifically, we encode the verification condition as a *constraint-based synthesis problem* of the form $\exists f. \forall X. \varphi$, where f chooses between different *non-deterministic axiomatizations* of probability distributions. This

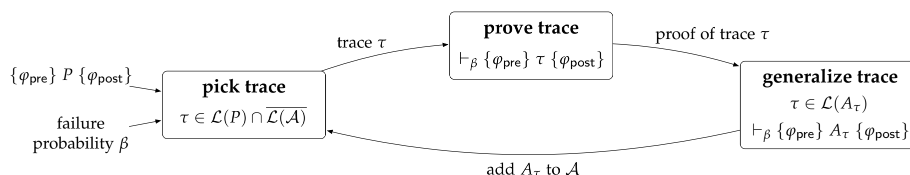


Figure 5.1: Main loop of verification algorithm

limits the precision of the analysis, but the resulting algorithm compares favorably to standard trace abstraction (Heizmann et al., 2009, 2013) and aHL (Barthe et al., 2016c), and boasts a high degree of automatability.

The contents of this chapter are based on work by Smith et al. (2019).

5.1 Overview and Illustration

Suppose we are given a probabilistic program P, pre- and post-conditions φ_{pre} and φ_{post}, and a numeric expression β representing the maximum allowed failure probability. Our goal is to prove that if we start executing P from any state satisfying φ_{pre}, the probability that the output state does not satisfy φ_{post} upon termination is at most β. This property is denoted by the following formula, reminiscent of a Hoare triple:

$$\vdash_\beta \{\varphi_{pre}\}\ P\ \{\varphi_{post}\}$$

We view P as a *control-flow automaton* whose language $\mathcal{L}(P)$ is the set of all *traces* from the program's entry location to its exit location. Our proof rule *overapproximates* $\mathcal{L}(P)$ by a larger set of traces, represented by a set of finite automata $\mathcal{A}$, while ensuring that the total failure probability across all traces in $\mathcal{L}(\mathcal{A})$ is at most β.

To apply our proof rule automatically, we apply an algorithmic technique summarized in Figure 5.1. The technique repeatedly tries to (*i*) pick a program trace $\tau \in \mathcal{L}(P)$ outside the approximation $\mathcal{L}(\mathcal{A})$, (*ii*) prove that

$\vdash_\beta \{\varphi_{pre}\}\ \tau\ \{\varphi_{post}\}$, i.e., the probability that the trace falsifies the Hoare triple is at most β, and then (*iii*) generalize the trace τ into an automaton A_τ encoding a set of traces with total failure probability at most β. Our approach succeeds if it constructs a set of automata $\mathcal{A}$ modeling all program traces $\mathcal{L}(P)$, with total failure probability at most β.

Illustrative Example: Loop-free Program

To warm up, we consider the loop-free program in Figure 5.2a. The function `ex1` takes a single [0,1]-valued input p and returns a Boolean value y. Our goal is to prove the following accuracy property:

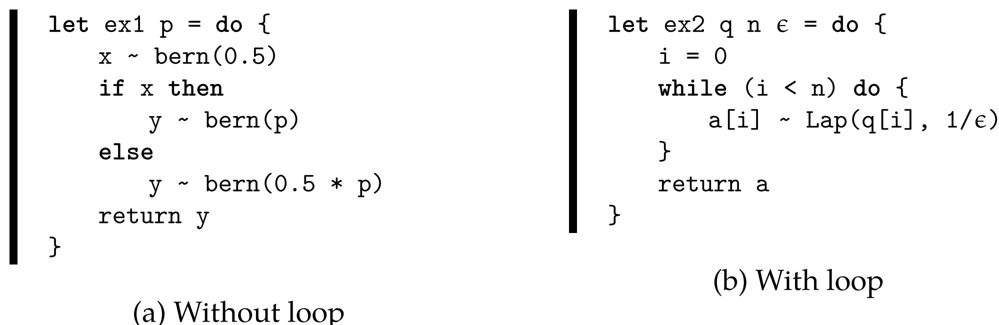
$$\vdash_p \{\mathit{true}\}\ \texttt{ex1}(p)\ \{\neg y\}$$

In words, the program fails to return $y = \mathit{false}$ with probability at most p. This property can be established informally: (*i*) the probability that the program takes the *then* branch and returns $y = \mathit{true}$ is $0.5p$; (*ii*) the probability that it takes the *else* branch and returns $y = \mathit{true}$ is $0.25p$. Therefore, the failure probability is $0.5p + 0.25p \leqslant p$.

Illustrating Proof Artifacts

We begin by describing the proof artifacts constructed by our approach. The program `ex1` is presented as a *control-flow automaton* over the alphabet of program statements, as shown in the left side of Figure 5.3. Edge labels of the form $[c]$ are guards (also known as *assume* statements) encoding possible branches of the conditional statement. Accepted traces start from the initial node `in` and end in the final, accepting node `ac`.

Our verification approach focuses on one trace at a time. There are two possible traces in our example program: one through the then branch and one through the else branch of the conditional. We refer to these traces as τ_1 and τ_2, respectively.



```
let ex1 p = do {
    x ~ bern(0.5)
    if x then
        y ~ bern(p)
    else
        y ~ bern(0.5 * p)
    return y
}
```

(a) Without loop

```
let ex2 q n ϵ = do {
    i = 0
    while (i < n) do {
        a[i] ~ Lap(q[i], 1/ϵ)
    }
    return a
}
```

(b) With loop

Figure 5.2: Examples of probabilistic programs

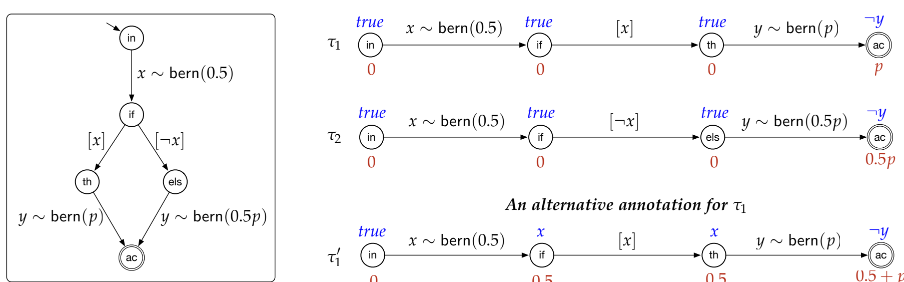


Figure 5.3: A simple probabilistic program and possible trace annotations

To prove accuracy properties about each trace, our technique annotates traces with auxiliary information. Let us consider the annotated trace τ_1 in Figure 5.2a. Each node along the trace is annotated with two labels: (*i*) the top/blue label is a logical formula representing a set of reachable program states at that point (these can be viewed as Hoare-style annotations); (*ii*) the bottom/red label is an expression representing the probability that the program does not end up in the blue states. Consider node `in` from τ_1: it is labeled by *true* and 0, indicating that the probability of failing to arrive in a program state satisfying *true* is 0 (as expected). However, consider node `ac`: it is labeled with $\neg y$ and p, indicating that the probability of failing to arrive in a state where $y = \mathit{false}$ is at most p. The other program trace τ_2, which traverses the other branch, is similar; the annotation of τ_2 demonstrates that its failure probability is at most $0.5p$.

At this point we have considered all of the `ex1`'s traces. If we naively sum up their probabilities of failure to bound the total failure probability, we get a failure probability of at most $p + 0.5p = 1.5p$, which is too weak—we wanted to prove an upper bound of p. However, we can give a more precise analysis since the two traces consider two mutually disjoint events: one path assumes x is true while the other assumes x is false. In this case, we can soundly take the *maximum* of the two failure probabilities, $0.5p$ and p, arriving at a total failure probability of p and concluding the proof.

Given a labeled trace, it is relatively straightforward to check if the annotations are valid. However, constructing the annotations may not be so easy. The main challenge is selecting labels for the results of sampling instructions—the invariants are not fully determined by the program, and in general the proper choice depends on the target property we are trying to establish. For instance, it is also possible to give an alternative annotation of τ_1, denoted τ_1' in Figure 5.2a. Node `if` is labeled with x and 0.5, indicating that the probability of *not* arriving in a state where $x = \mathit{true}$ is at most 0.5.

This annotated trace illustrates another general feature of our analysis: failure probabilities sum up along traces. Intuitively, this principle corresponds to a basic property of probabilities called the *union bound*: $\Pr[A \cup B] \leqslant \Pr[A] + \Pr[B]$ for any two events A and B. In particular, if A and B are interpreted as *bad* events—events violating labels at different nodes—the probability of any failure occurring along a trace is at most the sum of the failure probabilities of individual steps. In τ_1', the probability of $y = \mathit{true}$ at node `ac` of τ_1' is p, so the final failure probability computed for this trace is $0.5 + p$. While this annotation in τ_1' is sound, it is too weak to prove our desired property.

Encoding Trace Semantics

Our technique cleanly separates probabilistic assertions into two pieces: a non-probabilistic component describing the state of program variables (the blue annotations in Figure 5.3), and a single number summarizing the probabilistic part of the assertion (the red annotations in Figure 5.3). As a result, we can reduce probabilistic reasoning to logical reasoning, allowing us to harness the power of SMT solvers and *synthesis* techniques.

To illustrate, we show how to construct trace labels for τ_1. Our method proceeds in two steps. First, like in traditional *verification-condition generation*, we encode the semantics of trace τ_1 and the specification as a logical formula, which, if valid, implies that $\vdash_p \{true\}\ \tau_1\ \{\neg y\}$. Specifically, we construct the following (simplified, see Section 5.5) verification condition:

$$\exists f_x, f_y. \forall x, y, \omega_i. (\omega_0 = 0 \wedge \varphi) \Rightarrow (\neg y \wedge \omega_3 \leqslant p) \tag{5.1}$$

Above, φ is a set of conjuncts, each encoding the semantics of one statement in τ_1:

$$\varphi \coloneqq \underbrace{\begin{pmatrix} f_x = 1 \Rightarrow & x \wedge \omega_1 = \omega_0 + 0.5 \\ f_x = 2 \Rightarrow & \neg x \wedge \omega_1 = \omega_0 + 0.5 \\ f_x = 3 \Rightarrow & \omega_1 = \omega_0 \end{pmatrix}}_{x \sim \textsc{Bern}(0.5)}$$
$$\wedge \underbrace{(x \wedge \omega_2 = \omega_1)}_{[x]}$$
$$\wedge \underbrace{\begin{pmatrix} f_y = 1 \Rightarrow & y \wedge \omega_3 = \omega_2 + 1 - p \\ f_y = 2 \Rightarrow & \neg y \wedge \omega_3 = \omega_2 + p \\ f_y = 3 \Rightarrow & \omega_3 = \omega_2 \end{pmatrix}}_{y \sim \textsc{Bern}(p)}$$

Let us explain how the encoding models the program P. The variables

ω_i are fresh real-valued variables that represent the probability of failure along the path—ω_0, the initial probability at node `in`, is constrained to 0. The right-hand side of the implication in Equation (5.1) encodes the postcondition $\neg y$ and the upper bound on the failure probability $\omega_3 \leqslant p$.

The more interesting parts of the encoding are the existentially quantified variables f_x, f_y, which appear in φ; we assume that $f_x, f_y \in \{1, 2, 3\}$. These are used to *select* an *axiomatization* for each sampling statement. *Synthesizing* the right values for f_x and f_y allows us to show that Equation (5.1) is valid, and therefore prove correctness of τ_1. For instance, if f_x is set to 1, then $x \sim \text{Bern}(0.5)$ is encoded as an assignment statement $x \leftarrow$ *true* with an accumulated failure probability of 0.5, since x is not *true* with a probability of 0.5; if f_x is set to 3, then x is treated as a non-deterministic Boolean, incurring no probability of failure.

It is not hard to check that any proof of validity of Equation (5.1) must set $f_x = 3$ and $f_y = 2$, as otherwise we cannot establish the postcondition, $\neg y$, or the upper bound on failure, p. In general, we treat f_x and f_y as uninterpreted functions whose arguments are program inputs, so that the choice of axiomatization may depend on the program state (Section 5.5 presents the general form).

Labels via Craig Interpolation

Suppose that we have proved validity of Equation (5.1) and discovered that setting $f_x = 3$ and $f_y = 2$ yields a satisfiable formula. Plugging these values into Equation (5.1) and negating the postcondition, we arrive at the following unsatisfiable formula:

$$\omega_0 = 0 \wedge \underbrace{\omega_1 = \omega_0}_{x \sim \text{Bern}(0.5)} \wedge \underbrace{x \wedge \omega_2 = \omega_1}_{[x]} \wedge \underbrace{\neg y \wedge \omega_3 = \omega_2 + p}_{y \sim \text{Bern}(p)} \wedge (y \vee \omega_3 > p)$$

In first-order logic, it is known that if $A \wedge B$ is unsatisfiable, then there is a formula I over the shared vocabulary of A and B such that $A \Rightarrow$

I and $I \Rightarrow \neg B$ are valid. I is called a *Craig interpolant*. Intuitively, an interpolant overapproximates A while maintaining unsatisfiability with B; this overapproximation can be seen as trying to generalize the assertions as much as possible. In our unsatisfiable formula above, we can compute a *sequence* of interpolants by splitting the formula into A and B segments after every statement's encoding. The resulting interpolants compactly encode the two labels on traces, the sets of states and probabilities of failure. E.g., consider the split:

$$A \coloneqq \omega_0 = 0 \wedge \underbrace{\omega_1 = \omega_0}_{x \sim \textsc{Bern}(0.5)} \wedge \underbrace{x \wedge \omega_2 = \omega_1}_{[x]}$$

$$B \coloneqq \underbrace{\neg y \wedge \omega_3 = \omega_2 + p}_{y \sim \textsc{Bern}(p)} \wedge (y \vee \omega_3 > p)$$

A possible interpolant for $A \wedge B$ is $I \coloneqq \omega_2 = 0$. This indicates that any program state is reachable at node `th` (since program variables are unconstrained in I) with a probability of failure 0. The interpolant condition ensures that I can only mention ω_2, the only variable shared by A and B.

Illustrative Example: Handling Loops

We now consider a more complex example with loops, `ex2` in Figure 5.2b. `ex2` is a simplified sketch of a differential privacy mechanism. The program `ex2` takes an array of integers q of length n, and constructs an array a whose values are noisy versions of those in q. Specifically, for each element $q[i]$, $a[i]$ is noise drawn from the Laplace distribution with *mean* $q[i]$ and *scale* $1/\epsilon$, where $\epsilon > 0$ is a real-valued input to the program. All primitive distributions are defined in Section 5.2.

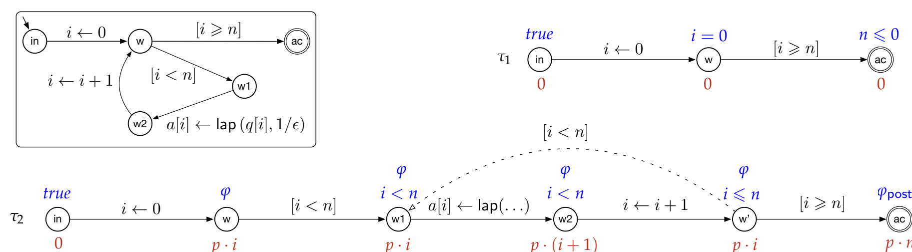


Figure 5.4: A looping illustrative example

Our goal is to prove the accuracy property

$$\vdash_{p \cdot n} \{\mathit{true}\}\ \mathtt{ex2}(q, n, \epsilon)\ \{\varphi_{\mathsf{post}}\}$$

where the post-condition is defined to be

$$\varphi_{\mathsf{post}} \coloneqq \forall j \in [0, n).\, |a[j] - q[j]| \leqslant \frac{1}{\epsilon} \log(1/p)$$

In other words, for any $p \in (0, 1]$, we want to verify that the difference between $a[j]$ and $q[j]$ is bounded by a function of ϵ and p. Observe that φ_{post} involves input parameters q, n, ϵ, and p, but p does not appear in the program—the accuracy property is a parameterized family of properties. From our postcondition, we see that we can guarantee tighter bounds on the error—the difference between the exact answer $q[j]$ and the noisy answer $a[j]$—if we are willing to allow this property to be violated with larger probability $p \cdot n$. This style of postcondition is common for many randomized algorithms, capturing the relationship between *accuracy*—how far the results are from the exact values—and probability of failure, or how often the target property will not hold.

Trace Generalization The control-flow automaton representation of `ex2` is shown in the box in Figure 5.4. While the total number of loop iterations is at most the input parameter n in the original program, the automaton

abstraction overapproximates these program behaviors with an infinite number of traces due to the loop. Therefore, unlike our first example, we cannot construct a proof for every trace individually. Our technique proceeds by *picking* traces, *proving* them correct, and *generalizing* them into automata representing infinite sets.

Let us first consider trace τ_1 in Figure 5.4; this trace does not enter the loop. The trace is easily shown to be correct since not entering the loop implies that $n \leqslant 0$, vacuously implying φ_{post} with failure probability 0. More interesting is trace τ_2 in Figure 5.4, which executes the loop body once and exits. The formula φ in the annotation is defined as follows:

$$\varphi \coloneqq \forall j \in [0, i).\, |a[j] - q[j]| \leqslant \frac{1}{\epsilon} \log(1/p)$$

Notice the probability of failure is $p \cdot i$ on nodes `w`, `w1`, `w2`, and `w'`. After loop exit, using the exit condition, we conclude that the probability of failure is $p \cdot n$. Informally, these labels capture the fact that the failure probability depends on how many times we have executed the loop, which is tracked in the counter i.

Our algorithm discovers that the labels are *inductive*: no matter how many times we execute the loop, the probability of failing to satisfy $\varphi \wedge i < n$ at loop entry is $p \cdot i$. Therefore, the algorithm generalizes this trace into an infinite set of traces by adding an edge from node `w'` to `w1` with the statement $[i < n]$.

With this additional edge in place, we now have an automaton representing all traces that go through the loop at least once. The total failure probability of those traces is the label under node `ac`: $p \cdot n$. Combined with trace τ_1, we have covered all the traces of `ex2`, proving that the total probability of failure is $p \cdot n + 0 = p \cdot n$ as desired.

Selecting Axioms for the Laplace Distribution The sampling statement $a[i] \sim \text{Lap}(\dots)$ in τ_2 is encoded by the following logical formula:

$$|a[i] - q[i]| \leqslant \frac{1}{\epsilon} \log(1/f_a(i,p,n)) \land \omega_3 = \omega_2 + f_a(i,p,n)$$

The left conjunct specifies that we can assume that the difference between $a[i]$ and $q[i]$ is at most $\frac{1}{\epsilon} \log(1/f_a(i,p,n))$; the right conjunct specifies that this assumption fails with a probability of $f_a(i,p,n)$. We treat f_a as an uninterpreted function with range $(0,1]$, so that there are infinitely many possible interpretations of f_a corresponding to different failure probability/accuracy tradeoffs for the Laplace distribution. To get the annotation proving correctness of τ_2 in Figure 5.4, our technique synthesizes the interpretation $f_a(i,p,n) = p$. With this choice, our analysis accumulates a probability of failure of p for every loop iteration, ending up with a total probability of failure of $p \cdot n$.

5.2 Programs, Automata, and Properties

In this section, we formalize our program model and accuracy specifications. Unlike in Chapters 3 and 4, the semantics of programs are important for reasoning about the validity of our proof technique.

Program Model and Semantics

To model probabilistic computation mathematically, we use probability sub-distributions. A function $\mu : C \to [0,1]$ defines a *discrete sub-distribution* over a set C if it is non-zero for at most countably many elements in C, and $\sum_{c \in C} \mu(c) \leqslant 1$; we will abbreviate discrete sub-distribution as distribution throughout this paper. We will often write $\mu(C')$ for a subset $C' \subseteq C$ to mean $\sum_{c \in C'} \mu(c)$. We write *dist*(C) for the set of

Name	Parameters	Semantics $s(d)$
Bernoulli ($\textsc{Bern}(e)$)	$e \in [0,1]$	$\mu(\mathit{true}) = s(e)$ and $\mu(\mathit{false}) = 1 - s(e)$
Uniform ($\textsc{Unif}(e)$)	e is a finite set	$\mu(c) = 1/\lvert s(e) \rvert$, for $c \in s(e)$
Laplace ($\text{Lap}\,(e_1, e_2)$)	$e_1 \in \mathbb{Z}$; $e_2 \in \mathbb{R}^{>0}$	$\mu(c) \propto \exp\left(-\frac{\lvert c - s(e_1) \rvert}{s(e_2)}\right)$, for $c \in \mathbb{Z}$
Exponential ($\textsc{Exp}(e_1, e_2)$)	$e_1 \in \mathbb{Z}$; $e_2 \in \mathbb{R}^{>0}$	$\mu(c) \propto \exp\left(-\frac{c - s(e_1)}{s(e_2)}\right)$, for $c \geqslant s(e_1)$

Table 5.1: Distribution expressions and their semantics

all distributions over C. The *support* of a distribution μ is defined as $\textsc{Support}(\mu) \coloneqq \{c \in C \mid \mu(c) > 0\}$.

We focus on discrete sub-distributions to keep the measure theory to a minimum. As a consequence, we only allow programs to sample from primitive discrete distributions. Supporting continuous primitive distributions—e.g., the Gaussian distribution—would not introduce any difficulties beyond requiring a more technically involved definition of the program semantics.

Program Expressions

We fix a set of variables V that appear in the program. A program *state* s is a map assigning every variable $v \in V$ to a value. We will use S to denote the set of all possible states. Given variable v, we use $s(v)$ to denote the value of v in state s. Given constant c, we use $s[v \mapsto c]$ to denote the state s with variable v mapped to c. The semantics of an expression e is a function $[\![e]\!] : S \to D$ from a state to an element of some type D. For instance, the expression $x + y$ in state s is interpreted as $[\![x + y]\!]\,(s) = s(x) + s(y)$. We will often abbreviate $[\![e]\!]\,(s)$ by $s(e)$.

Distribution Expressions A *distribution expression* d is interpreted as a distribution family $[\![d]\!] : S \to \mathit{dist}(D)$, mapping a state in S to a distribution over D with countable support. Our framework can naturally handle any distribution expression that can be interpreted as a discrete distribu-

tion. For concreteness, we will consider the four primitive distributions in Table 5.1.

Consider the Bernoulli distribution expression, $\textsc{Bern}(e)$. Given a state s, semantically $\textsc{Bern}(e)$ is the distribution $\mu \in \mathit{dist}(\mathbb{B})$ where $\mu(\mathit{true}) = s(e)$ and $\mu(\mathit{false}) = 1 - s(e)$. Similarly, the uniform distribution expression $\textsc{Unif}(e)$, where e encodes to a finite set, is interpreted as the distribution assigning equal probability to every element in $s(e)$.

We also use the (discrete) *Laplace distribution*. For a state s, the distribution expression $\mathrm{Lap}\,(e_1, e_2)$ is semantically the discrete Laplace distribution with *mean* $s(e_1)$ and *scale* $s(e_2)$: for every integer $c \in \mathbb{Z}$, it assigns a probability proportional to $\exp\left(-\frac{|c - s(e_1)|}{s(e_2)}\right)$. The (discrete) *exponential distribution* expression $\textsc{Exp}(e_1, e_2)$ is similar, but only assigns positive probability to integers above the *shift* $s(e_1)$.

We implicitly assume that arguments of distribution expressions are well-typed and valid.

Programs, Statements, and Traces

Our verification technique will target programs written in a probabilistic, imperative language. The *basic statements* are drawn from a set Σ:

1. **Assignment** statements $v \leftarrow e$, where e is an expression over V, e.g., $v_1 + v_2$.
2. **Sampling** statements $v \sim d$, where d is a distribution expression.
3. **Assume** statements $\mathsf{assume}(b)$, where b is a Boolean expression over V.

A *trace* τ is a finite sequence of statements $\mathit{st}_1; \cdots; \mathit{st}_n$, and a program P is interpreted as a (possibly infinite) set of traces $\mathcal{L}(P)$. We include full details of the programming language in Appendix C.5; the interpretation is standard, using assume statements to model typical control-flow constructs. For instance, a conditional statement **if** b **then** τ_1 **else** τ_2 can be modeled as the pair of traces $\mathsf{assume}(b); \tau_1$ and $\mathsf{assume}(\neg b); \tau_2$. By construction, traces

$$\begin{aligned}
[\![v \leftarrow e]\!]\,(s) &\coloneqq \mathit{unit}(s[v \mapsto [\![e]\!]\,(s)]) \\
[\![v \sim d]\!]\,(s) &\coloneqq \mathit{bind}([\![d]\!]\,(s), \lambda x.\, \mathit{unit}(s[v \mapsto x])) \\
[\![\mathsf{assume}(b)]\!]\,(s) &\coloneqq \text{if } [\![b]\!]\,(s) \text{ then } \mathit{unit}(s) \text{ else } 0 \\
[\![\mathit{st}\,;\tau]\!]\,(s) &\coloneqq \mathit{bind}([\![\mathit{st}]\!]\,(s), [\![\tau]\!])
\end{aligned}$$

Figure 5.5: Statement and trace semantics

in $\mathcal{L}(P)$ are semantically disjoint—no trace in $\mathcal{L}(P)$ is a prefix of (or equal to) any other trace in $\mathcal{L}(P)$, and the first differing statements between any two traces are of the form $\mathsf{assume}(b)$ and $\mathsf{assume}(\neg b)$.

Trace Semantics We interpret a trace τ as a function $[\![\tau]\!] : S \to \mathit{dist}(S)$ from input states to distributions over output states. To define this semantics formally, we need two standard constructions on distributions. The map $\mathit{unit} : D \to \mathit{dist}(D)$ maps $a \in D$ to the Dirac distribution δ_a at a, i.e., the distribution that returns 1 at a and 0 otherwise. The map $\mathit{bind} : \mathit{dist}(D_1) \to (D_1 \to \mathit{dist}(D_2)) \to \mathit{dist}(D_2)$ combines probabilistic computations in sequence: $\mathit{bind}(\mu, f)(a_2) = \sum_{a_1 \in D_1} \mu(a_1) \cdot f(a_1)(a_2)$. These maps are the usual unit and bind for the (sub-)distribution monad. Then, we can give semantics to basic statements and traces as shown in Figure 5.5.

Finally, the semantics of a program P is defined as the aggregate of its traces. Formally, $[\![P]\!] : S \to \mathit{dist}(S)$ is defined as

$$[\![P]\!]\,(s) \coloneqq \sum_{\tau \in \mathcal{L}(P)} [\![\tau]\!]\,(s)$$

where each term $[\![\tau]\!]\,(s)$ is the output distribution from running τ starting from input s, and the sum of distributions is defined pointwise. For any disjoint set of traces corresponding to a program P, the sum on the right-

hand side is indeed a distribution.

Programs as Automata

We can encode the set of possible traces of a program P as a regular language $\mathcal{L}(P)$ represented by all paths through its control-flow graph. We begin with a general definition of automata over program statements, and then show how we represent programs as automata.

Automata over Statements

A *finite-state automaton over statements* A is a graph $\langle Q, \delta \rangle$, where

1. Q is a finite set of *nodes*.
2. $\delta \subseteq Q \times \Sigma \times Q$ is the *transition relation*, where Σ are basic statements.
3. $q^{\mathsf{in}}, q^{\mathsf{ac}} \in Q$ are special nodes called the *initial* and *accepting* nodes, respectively.

We will use $q_i \xrightarrow{\mathit{st}} q_j$ to denote that $\langle q_1, \mathit{st}, q_j \rangle \in \delta$. We write $\mathcal{L}(A)$ for the language of traces *accepted* by A, where a trace $\mathit{st}_1, \ldots, \mathit{st}_n$ is accepted iff $\left\{ q^{\mathsf{in}} \xrightarrow{\mathit{st}_1} q_1, q_1 \xrightarrow{\mathit{st}_2} q_2, \ldots, q_{n-1} \xrightarrow{\mathit{st}_n} q^{\mathsf{ac}} \right\} \subseteq \delta$. It will sometimes be useful to use multiple automata to model the traces in a single program. We will use $\mathcal{L}(\mathcal{A})$ to denote the union of all languages accepted by a set of automata $\mathcal{A}$, i.e., $\bigcup_{A \in \mathcal{A}} \mathcal{L}(A)$.

We assume that all nodes $q \in Q$ can reach the accepting node q^{ac} via the transition relation δ, and that there are no transitions starting from q^{ac}. We also assume that automata model well-formed control flow, i.e., (*i*) all nodes $q_i \in Q$ have at most two outgoing transitions and (*ii*) if $q_i \xrightarrow{\mathit{st}_1} q_j$ and $q_i \xrightarrow{\mathit{st}_2} q_k$ for $j \neq k$, then $\mathit{st}_1, \mathit{st}_2$ are of the form $\mathsf{assume}(b_1)$ and $\mathsf{assume}(b_2)$, such that $b_1 \equiv \neg b_2$.

From Program Traces to Automata

We will identify a *program* with an automaton representing its its *control-flow graph* (CFG). A program P is of the form $\langle L, \delta \rangle$, where the nodes L of the automaton denote the set of *program locations* (e.g., line numbers). The special nodes $\ell^{in}, \ell^{ac} \in L$ model the first and last lines of the program. To ensure there is no control-flow non-determinism, we assume that for any $\ell_i \xrightarrow{\mathsf{assume}(b)} \ell_j$, there is a transition $\ell_i \xrightarrow{\mathsf{assume}(\neg b)} \ell_k$.

We use $V^{in} \subseteq V$ to denote the set of input variables, which are not modified by the program. We will also use $V^{det} \subseteq V$ to denote the set of program variables whose values are assigned deterministically, i.e., not affected by probabilistic choice—by definition, $V^{in} \subseteq V^{det}$. (We may not be able to determine V^{det} exactly in practice, but we can under-approximate it via a simple static analysis.)

Probabilistic Accuracy Properties

We will define specifications using the Hoare-style statement

$$\vdash_\beta \{\varphi_{pre}\}\, \tau \,\{\varphi_{post}\}$$

where the *precondition* $\varphi_{pre} \subseteq S$ and *postcondition* $\varphi_{post} \subseteq S$ are sets of program states, and the *failure probability* β is a $[0,1]$-valued function over input variables V^{in}. For simplicity, we will treat β as an expression over V^{in}—e.g., 0 or $p \cdot n$ in Section 5.1—and use $s(\beta)$ to denote the value of β in state s.

We say that $\vdash_\beta \{\varphi_{pre}\}\, \tau \,\{\varphi_{post}\}$ is *valid* iff for any state $s \in \varphi_{pre}$, we have $\mu(\overline{\varphi_{post}}) \leqslant s(\beta)$, where $\mu = [\![\tau]\!](s)$ and $\overline{\varphi_{post}} = S \setminus \varphi_{post}$. In other words, the probability that the trace starts in φ_{pre} and *does not* end up in φ_{post} is *upper bounded* by β. We extend this notation to programs P in the natural way, writing $\vdash_\beta \{\varphi_{pre}\}\, P \,\{\varphi_{post}\}$ iff for any input state $s \in \varphi_{pre}$, the output distribution $\mu = [\![P]\!](s)$ satisfies the bound $\mu(\overline{\varphi_{post}}) \leqslant s(\beta)$.

5.3 Trace Abstraction Modulo Probability

With the preliminaries out of the way, we begin to introduce a version of trace abstraction for probabilistic programs and show how to use it to prove accuracy specifications. Given a program P, suppose we want to establish the following accuracy specification: $\vdash_\beta \{\varphi_{\mathsf{pre}}\}\ P\ \{\varphi_{\mathsf{post}}\}$. We will overapproximate the traces of P with a set of automata $\mathcal{A}$ and analyze each automaton separately, so we can focus on smaller groups of possible traces. If we can show that the probability φ_{post} does not hold across all automata is at most β, this implies the accuracy specification. We formalize this argument in the following proof rule.

Theorem 5.1 (General proof rule)**.** *The specification* $\vdash_\beta\ \{\varphi_{\mathsf{pre}}\}\ P\ \{\varphi_{\mathsf{post}}\}$ *is valid if there exists a set of automata* $\mathcal{A}$ *such that*

$$\mathcal{L}(P) \subseteq \mathcal{L}(\mathcal{A}) \qquad \text{(Trace inclusion)}$$

$$\forall s \in \varphi_{\mathsf{pre}}.\ \sum_{\tau \in \mathcal{L}(\mathcal{A})} [\![\tau]\!]\,(s)(\overline{\varphi_{\mathsf{post}}}) \leqslant s(\beta) \qquad \text{(Failure upper bound)}$$

Theorem 5.1 is proven in Appendix C.1. This proof rule is concise but difficult to apply in practice, even given the set of automata $\mathcal{A}$—while the trace inclusion property can be checked via regular language inclusion, the failure probability upper bound is more complicated. To make this second condition easier to check, we enrich the automata with additional information on each state; local properties of these labeled automata will then imply the failure probability upper bound.

Enriching Automata with Labels

We work with automata where every node is labeled with a predicate on states (equivalently, a set of states), and a function representing the failure probability—we call such automata *failure automata*. The rough intuition

is that at each node q, the predicate label represents a program invariant that holds on all traces reaching q from the beginning of the program, except with probability given by the failure probability label.

Definition 5.2 (Failure Automata)**.** *A* failure automaton $A = \langle Q, \delta, \lambda, \kappa \rangle$ *is an automaton* $\langle Q, \delta \rangle$ *with two labeling functions,* λ *and* κ*, where*

1. λ *maps every node* $q \in Q$ *to a set of states, and*
2. κ *maps every node* $q \in Q$ *to a [0,1]-valued function over* V^{det}*.*

We say that A *is* well-labeled *iff the following conditions hold:*

1. $\kappa(q^{in}) = 0$ *and* $\kappa(q^{ac})$ *is a* $[0,1]$*-valued function over the input variables* $V^{in} \subseteq V^{det}$*, and*
2. *for every transition* $q_i \xrightarrow{st} q_j$*, the statement*

$$\vdash_{wp^f(\kappa(q_j), st) - \kappa(q_i)} \{\lambda(q_i)\}\, st\, \{\lambda(q_j)\}$$

is valid where wp^f *is a weakest-precondition operation over failure-probabilities:* $wp^f(e, st) := e$ *for assume and sampling statements, and* $wp^f(e_1, v \leftarrow e_2) := e_1[v \mapsto e_2]$*.*

The two conditions ensure that if we take any trace $\tau \in \mathcal{L}(()A)$, then $\vdash_{\kappa(q^{ac})} \{\lambda(q^{in})\}\, \tau\, \{\lambda(q^{ac})\}$ is valid. Point (2) ensures that failure probability accumulates additively as we move along the trace, starting from being 0 at q^{in}, as stipulated by point (1). Crucially, both points are *local* conditions: they can be easily checked given a failure automaton. However, coming up with well-labeled automata for a given program is not at all trivial—we return to this question in the next two sections.

Example 5.3. *Recall our example from Section 5.1, illustrated in Figure 5.3. The lower part of Figure 5.3 shows a failure automaton named* τ_1' *with* λ *and* κ *shown above and below the nodes, respectively. Notice that the initial node*

in is labeled with $\lambda(\mathit{in}) \coloneqq true$ and $\kappa(\mathit{in}) \coloneqq 0$. Focusing on the edge from node th, the labeling at ac satisfies condition (2) for well-labeledness in Definition 5.2. The condition says that the following statement must be valid: $\vdash_p \{x\}\ y \sim \textsc{Bern}(p)\ \{\neg y\}$. The failure probability p is the simplification of the expression $wp^f(0.5 + p, y \sim \textsc{Bern}(p)) - 0.5$. The statement is valid since y is true with probability p after executing $y \sim \textsc{Bern}(p)$.

The following theorem (whose proof is provided in Appendix C.1) establishes soundness of annotations on well-labeled automata. Specifically, the failure probability label on q^{ac}—namely, $\kappa(q^{ac})$—is an upper bound on the probability that executions through A do not end up in a state in $\lambda(q^{ac})$.

Theorem 5.4 (Well-Labeled Automata Soundness)**.** *Let A be a well-labeled failure automaton. Then, for every $s \in \lambda(q^{in})$ and $\mu = [\![\tau]\!](s)$, we have*

$$\sum_{\tau \in \mathcal{L}(A)} \mu(\overline{\lambda(q^{ac})}) \leqslant s(\kappa(q^{ac}))$$

Proofs from Well-labeled Automata

Now that we have established soundness of well-labeled automata, we refine our original proof rule (Theorem 5.1) using failure automata. The following theorem (proven in Appendix C.1) demonstrates how to establish correctness using a set of failure automata.

Theorem 5.5 (Proof Rule with Failure Automata)**.** *The statement*

$$\vdash_\beta \{\varphi_{pre}\}\ P\ \{\varphi_{post}\}$$

is valid if there exist well-labeled automata $\mathcal{A} = \{A_1, \ldots, A_n\}$ such that the fol-

lowing conditions hold:

$$\mathcal{L}(P) \subseteq \mathcal{L}(\mathcal{A}) \qquad \text{(Trace inclusion)}$$

$$\forall i \in [1, n].\, \varphi_{\mathsf{pre}} \subseteq \lambda_i(q_i^{\mathsf{in}}) \qquad \text{(Precondition inclusion)}$$

$$\forall i \in [1, n].\, \lambda_i(q_i^{\mathsf{ac}}) \subseteq \varphi_{\mathsf{post}} \qquad \text{(Postcondition inclusion)}$$

$$\forall s \in \varphi_{\mathsf{pre}}.\, \sum_{i=1}^{n} s(\kappa_i(q_i^{\mathsf{ac}})) \leqslant s(\beta) \qquad \text{(Failure upper bound)}$$

The trace, precondition, and postcondition inclusion conditions are the same as in trace abstraction for non-probabilistic programs. The failure probability upper bound condition ensures that the overapproximation of failure probability resulting from abstraction does not exceed β. Notice that precondition and postcondition inclusion checks can be performed using an SMT solver, assuming labels are encoded in a supported first-order theory. Similarly, the failure probability upper bound condition involves summing up the labels on the accepting nodes of all failure automata, allowing us to perform the check with an SMT solver.

Example 5.6. *Recall the example program* `ex2` *from Section 5.1, illustrated in Figure 5.4. The two automata, denoted τ_1 and τ_2 in Figure 5.4, are well-labeled. The automata cover all program traces. The initial nodes, denoted* `in`*, have the labels λ as true, therefore satisfying the precondition inclusion condition. The accepting nodes, denoted* `ac`*, both imply the postcondition, φ_{post}. Finally, the sum of the failure probabilities on accepting nodes is $0 + p \cdot n \leqslant p \cdot n$, satisfying the failure probability condition.*

5.4 Constructing Trace Abstractions

Theorem 5.5 reduces checking accuracy properties to finding a set of well-labeled automata. Our algorithm for automating this proof rule is technically complex, and spans the following two sections. Here, we will

$$\frac{}{\mathcal{A} \longrightarrow \emptyset}\ \textsc{Init}$$

$$\frac{\tau \in \mathcal{L}(P) \cap \overline{\mathcal{L}(\mathcal{A})} \qquad A_\tau = \mathsf{label}(\tau, \varphi_{pre}, \varphi_{post}, \beta)}{\mathcal{A} \longrightarrow \mathcal{A} \cup \{A_\tau\}}\ \textsc{Trace}$$

$$\frac{\begin{array}{c} A = \langle Q, \delta, \lambda, \kappa \rangle \in \mathcal{A} \qquad q_i, q_j \in Q \qquad s\!\!\!/ \in \Sigma \\ A' = \left\langle Q, \delta \cup \left\{ q_i \xrightarrow{s\!\!\!/} q_j \right\}, \lambda, \kappa \right\rangle \qquad \vdash_{wp^f(\kappa(q_j), s\!\!\!/) - \kappa(q_i)} \{\lambda(q_i)\}\, s\!\!\!/\, \{\lambda(q_j)\} \end{array}}{\mathcal{A} \longrightarrow (\mathcal{A} \setminus \{A\}) \cup \{A'\}}\ \textsc{Generalize}$$

$$\frac{A_1, A_2 \in \mathcal{A} \qquad A = A_1 \mathbin{⩕} A_2}{\mathcal{A} \longrightarrow (\mathcal{A} \setminus \{A_1, A_2\}) \cup \{A\}}\ \textsc{Merge}$$

$$\frac{\mathcal{L}(P) \subseteq \mathcal{L}(\mathcal{A}) \qquad \forall s \in \varphi_{pre}.\ \sum_{i=1}^{|\mathcal{A}|} s(\kappa_i(q_i^{ac})) \leqslant s(\beta)}{\vdash_\beta \{\varphi_{pre}\}\ P\ \{\varphi_{post}\}}\ \textsc{Correct}$$

Figure 5.6: Overall abstract algorithm for implementing Theorem 5.5

present the algorithm and prove soundness, assuming a procedure for well-labeling single traces; we will detail this last piece in Section 5.5. Then, we compare our algorithm with two existing techniques: the union bound logic aHL, and standard trace abstraction.

Algorithm Overview

Our algorithm maintains a set $\{A_i\}_i$ of well-labeled failure automata modeling some of the program traces, and repeatedly finds traces $\tau \in \mathcal{L}(P)$ that are not in $\{A_i\}_i$. If a trace can be well-labeled, it is converted into a well-labeled automaton A_i proving that $\vdash_\beta \{\varphi_{pre}\}\ \tau\ \{\varphi_{post}\}$ and added to the current automaton set. Throughout, the algorithm may simplify or

transform the automaton set by merging automata together and generalizing automata by adding new edges. The process terminates successfully if the set of failure automata $\{A_i\}_i$ satisfies the conditions in Theorem 5.5.

The input to the algorithm is a program P, a pre- and post-condition φ_{pre} and φ_{post}, and a target failure probability β, a function over the input variables of the program. The entire algorithm is presented in Figure 5.6 as a set of non-deterministic guarded rules. The algorithm preserves the invariant that the set of automata $\mathcal{A}$ are well-labeled. We briefly consider each rule in turn.

Initialization The rule INIT is the only rule with no premises and serves as the initialization rule. Not surprisingly, the set of failure automata $\mathcal{A}$ is initially empty.

Trace Sampling The rule TRACE picks a trace τ that is in the program P but not covered by the set of automata $\mathcal{A}$. It then uses the function label to construct a well-labeled automaton A_τ implying that $\vdash_\beta \{\varphi_{\mathsf{pre}}\}\, \tau \,\{\varphi_{\mathsf{post}}\}$. We will detail the label operation in Section 5.5; for now, we just note that label may fail, in which case the rule TRACE does not fire and the algorithm tries a different trace.

Generalizing Automata The rule GENERALIZE expands the language $\mathcal{L}(\mathcal{A})$ by adding new edges to an automaton $A \in \mathcal{A}$. When the new edges form loops, this rule can be seen as generalizing from automata modeling finite unrollings of looping statements to automata overapproximating loops. The side-conditions ensure that this transformation preserves well-labeledness.

Merging Automata The rule MERGE combines automata whose traces are *mutually exclusive*, allowing us to take the *maximum* failure probability instead of the *sum*. Intuitively, automata that begin with the same prefix of

statements before making mutually exclusive assumptions—say, $\mathsf{assume}(b)$ and $\mathsf{assume}(\neg b)$—can have their prefixes merged together if they have equivalent labels. This operation can be seen as constructing an automaton combining two branches of a conditional.

Concretely, the operator $\mathbin{\wedge\!\!\!\wedge}$ takes two automata, A_1 and A_2, and returns a new automaton that accepts the union of the traces. We formalize $\mathbin{\wedge\!\!\!\wedge}$ and its preconditions below:

Definition 5.7. *We assume the two automata* A_1, A_2 *are of the form* $A_i = \langle Q_i, \delta_i, \lambda_i, \kappa_i \rangle$ *with the initial and final nodes* $q_i^{\mathsf{in}}, q_i^{\mathsf{ac}}$. *Suppose there is a prefix of statements* $\mathit{st}_1, \ldots, \mathit{st}_n$ *such that*

1. *every path from* q_1^{in} *to* q_1^{ac} *is of the form:*

$$q_1^{\mathsf{in}} \xrightarrow{\mathit{st}_1} q_{1,1} \xrightarrow{\mathit{st}_2} q_{1,2} \ldots q_{1,n} \xrightarrow{\mathsf{assume}(b)} q_{1,n+1} \cdots q_1^{\mathsf{ac}}$$

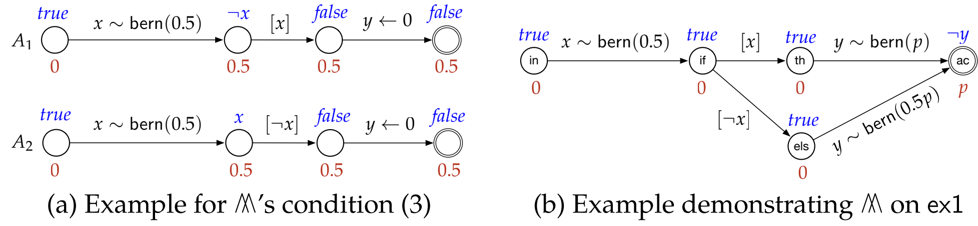
2. *every path from* q_2^{in} *to* q_2^{ac} *is of the form:*

$$q_2^{\mathsf{in}} \xrightarrow{\mathit{st}_1} q_{2,1} \xrightarrow{\mathit{st}_2} q_{2,2} \ldots q_{2,n} \xrightarrow{\mathsf{assume}(\neg b)} q_{2,n+1} \cdots q_2^{\mathsf{ac}}$$

3. *each prefix node* $q \in \{q_1^{\mathsf{in}}, q_{1,1}, \ldots, q_{1,n}\}$ *has equivalent labels (*λ *and* κ*) to its corresponding node in* $\{q_2^{\mathsf{in}}, q_{2,1}, \ldots, q_{2,n}\}$.

Then, $A_1 \mathbin{\wedge\!\!\!\wedge} A_2$ *yields a failure automaton* $A = \langle Q, \delta, \lambda, \kappa \rangle$ *with*

1. $Q = Q_1 \cup (Q_2 \setminus \{q_2^{\mathsf{in}}, q_{2,1}, \ldots, q_{2,n}, q_2^{\mathsf{ac}}\})$;
2. $\delta = \delta_1 \cup \delta_2 \cup \left\{ q_i \xrightarrow{\mathit{st}} q^{\mathsf{ac}} \mid q_i \xrightarrow{\mathit{st}} q_2^{\mathsf{ac}} \in \delta_2 \right\}$, *with all edges to/from undefined nodes removed;*
3. $q^{\mathsf{in}} = q_1^{\mathsf{in}}$ *and* $q^{\mathsf{ac}} = q_1^{\mathsf{ac}}$;
4. λ *agrees with* λ_1 *and* λ_2, *except that* $\lambda(q^{\mathsf{ac}}) = \lambda(q_1^{\mathsf{ac}}) \cup \lambda(q_2^{\mathsf{ac}})$; *and*
5. κ *agrees with* κ_1 *and* κ_2, *except that* $\kappa(q^{\mathsf{ac}}) = \max(\kappa(q_1^{\mathsf{ac}}), \kappa(q_2^{\mathsf{ac}}))$.



(a) Example for $\wedge\!\!\!\wedge$'s condition (3) (b) Example demonstrating $\wedge\!\!\!\wedge$ on ex1

Figure 5.7: Examples of merging automata

More advanced extensions of this operation are also possible—e.g., also merging common post-fixes along with common prefixes—but we stick with this version for concreteness.

If two automata are well-labeled and the MERGE rule applies, then the resulting merged automaton is also well-labeled. It is, however, important to note condition (3) in Definition 5.7, which states that the shared prefix between the two automata must have the same labels on both automata. If that condition is violated, the result may not be well-labeled, as illustrated in the following example.

Example 5.8. *Consider the two well-labeled single-trace automata A_1 and A_2 in Figure 5.7(a), which model a conditional statement and share the prefix $x \sim$ $\text{BERN}(0.5)$. The annotations prove that both traces satisfy $\vdash_{0.5} \{true\}\ A_i\ \{y > 0\}$. The operation $\wedge\!\!\!\wedge$ does not apply here, since the automata disagree on the label of the second node. However, suppose that we apply $\wedge\!\!\!\wedge$ nonetheless. This results in a final node with failure probability $\max(0.5, 0.5) = 0.5$. But this is not sound, since the probability of failing to achieve $y > 0$ is 1 when both traces are considered together, since both traces set y to 0.*

We also give an example of a sound application of merge.

Example 5.9. *Consider the two well-labeled automata τ_1 and τ_2 from Figure 5.3 in Section 5.1. They satisfy the conditions for $\wedge\!\!\!\wedge$. Figure 5.7(b) shows the result of applying $\wedge\!\!\!\wedge$ to these two automata. Notice that the accepting node, denoted $\mathtt{ac}$, has a label $\kappa(\mathtt{ac}) = \max(p, 0.5p)$, which is equal to p.*

Lemma 5.10. *If* A_1, A_2 *are well-labeled and satisfy the* $\mathbin{/\!\!\!\backslash\!\!\backslash}$ *conditions, then* $A = A_1 \mathbin{/\!\!\!\backslash\!\!\backslash} A_2$ *is well-labeled.*

Termination Finally, the rule CORRECT gives the termination condition for the algorithm, corresponding to the conditions from Theorem 5.5. Notice that precondition and postcondition inclusion hold by construction, since they were ensured by the labeling function label when the first trace in each automaton was added to the automaton set by rule TRACE.

Theoretical Properties

Soundness As expected, the algorithm is sound. The proof is given in Appendix C.2.

Theorem 5.11 (Soundness)**.** *If* CORRECT *applies, then* $\vdash_\beta \{\varphi_{\mathsf{pre}}\}\ P\ \{\varphi_{\mathsf{post}}\}$ *is valid.*

(In)completeness Our approach is incomplete, primarily from the application of the *union bound*, which, in some programs, does not allow us to prove the tightest possible failure probabilities. As an example, consider

$$\vdash_{0.75} \{\mathit{true}\}\ x \sim \textsc{Bern}(0.5); y \sim \textsc{Bern}(0.5)\ \{x \wedge y\}$$

Any well-labeled automaton will upper bound the failure probability by 1, since we have no means of assuming independent sampling in both statements. This example can be handled by coalescing the two sampling statements into a single statement; however, the general issue arises in loops, too.

Nevertheless, we can compare the expressivity of our approach with two existing techniques: the union bound logic (Barthe et al., 2016c) and trace abstraction (Heizmann et al., 2009, 2013).

Union Bound Logic The *union bound logic* (Barthe et al., 2016c) is an extension of Hoare logic with failure probabilities, where Hoare triples are analogous to our annotations $\vdash_\beta \{\varphi_{\mathsf{pre}}\}\, P\, \{\varphi_{\mathsf{post}}\}$. Our notion of well-labeled automata can capture proofs in the union bound logic with the exception of a few points, and our algorithm can recover a precise class of well-labeled automata. We formalize this correspondence and prove a completeness result in Appendix C.5.

Trace Abstraction Our technique generalizes trace abstraction for non-probabilistic, single-procedure programs (Heizmann et al., 2009, 2013). When given a non-probabilistic program P and Hoare triple $\{\varphi_{\mathsf{pre}}\}\, P\, \{\varphi_{\mathsf{post}}\}$, we can construct trace-abstraction proofs by simply setting the failure probability upper bound to 0 in the specification. Consequently, the failure probability labels of nodes of all automata in $\mathcal{A}$ must be 0 for the proof to hold. In this setting, the state labels (λ) are overapproximations of reachable states at a specific node, corresponding to the annotations of *Floyd–Hoare automata* defined by Heizmann et al. (2013).

5.5 Labeling Individual Traces

In the algorithm presented in Figure 5.6, the key subroutine is the label operation for rule TRACE. Recall that given a single trace τ, pre- and post-conditions φ_{pre} and φ_{post}, and failure probability β, label attempts to construct a well-labeled automaton A_τ for τ proving $\vdash_\beta \{\varphi_{\mathsf{pre}}\}\, \tau\, \{\varphi_{\mathsf{post}}\}$. We now show how to reduce this task to a constraint-solving problem. Our approach is inspired by *interpolation-based verification* McMillan (2006), where the semantics of τ are encoded as a formula in first-order logic to check if it can falsify a Hoare triple. If the trace does not falsify the triple, Craig interpolants are computed along the trace forming a Hoare-style annotation. However, our setting is richer: we need to (*i*) handle traces with

probabilistic semantics and (*ii*) construct two kinds of annotations—sets of states and failure probability expressions. We demonstrate how to reduce this problem to Craig interpolation over a first-order theory, thus eliminating probabilistic reasoning. We summarize our approach below:

1. **Axiomatizing Distributions**: We show how to encode $\vdash_\beta \{\varphi_{\mathsf{pre}}\}\, \tau \,\{\varphi_{\mathsf{post}}\}$ as a logical formula. The key challenge is in encoding semantics of sampling statements. We address this challenge by observing that we can encode sampling statements by introducing appropriate *logical axioms* about the distributions. This results in a *constraint-based synthesis* problem of the form $\exists f.\, \forall X.\, \varphi$, where discovering a function f amounts to finding an appropriate axiom for each sampling statement in order to establish correctness of the trace.

2. **Craig Interpolation**: Once we have solved the synthesis problem by finding a solution for f, we are left with a valid logical formula of the form $\forall X.\, \varphi$, which we can use to compute interpolants using standard techniques. We demonstrate that these interpolants can be converted to a well-labeling of A_τ.

Proofs via Distribution Axiomatization

We now describe how we can check validity of the specification

$$\vdash_\beta \{\varphi_{\mathsf{pre}}\}\, st_1; \cdots ; st_n \,\{\varphi_{\mathsf{post}}\}$$

Our approach is analogous to logical encodings of program paths in verification of non-probabilistic programs; there, each statement st_i is encoded as a formula φ_{st_i} in some appropriate first-order theory, e.g., the theories of linear arithmetic or arrays. Novel to our setting, we use *distribution axioms* to approximate the semantics of sampling statements in a first-order theory.

Assumption φ^{ax}	Upperbound e^{ub}	Parameters
Bernoulli: $v \sim \textsc{Bern}(v')$		
$(f(V^{\mathsf{in}}) = 1 \land v) \lor (f(V^{\mathsf{in}}) = 2 \land \neg v)$	$\begin{cases} v' & \text{if } f(V^{\mathsf{in}}) = 1 \\ 1 - v' & \text{if } f(V^{\mathsf{in}}) = 2 \\ 0 & \text{otherwise} \end{cases}$	$f(V^{\mathsf{in}}) \in \{1, 2, 3\}$
Uniform: $v \sim \textsc{Unif}(v')$		
$v \in f(V^{\mathsf{in}})$	$\lvert f(V^{\mathsf{in}}) \rvert / \lvert v' \rvert$	$f(V^{\mathsf{in}}) \subseteq v'$
Laplace: $v \sim \mathrm{Lap}\,(v_1, v_2)$		
$\lvert v - v_1 \rvert > v_2 \log \left(\frac{1}{f(V^{\mathsf{in}})} \right)$	$f(V^{\mathsf{in}})$	$f(V^{\mathsf{in}}) \in (0, 1]$
Exponential: $v \sim \textsc{Exp}(v_1, v_2)$		
$v < v_1 \lor v - v_1 > v_2 \log \left(\frac{2}{f(V^{\mathsf{in}})} \right)$	$f(V^{\mathsf{in}})$	$f(V^{\mathsf{in}}) \in (0, 1]$

Table 5.2: Example families of distribution axioms (v is always free in the distribution expression)

Logical Theory We assume that deterministic program expressions correspond to a first-order theory, like linear arithmetic. Given a formula φ, a model M of φ, denoted $M \models \varphi$, is a valuation of its free variables $fv(\varphi)$ satisfying the formula—e.g., $M \models x + y > 0$ where $M = \{x \mapsto 0, y \mapsto 1\}$. We use $M(\varphi)$ to denote φ with all free variables replaced by their interpretation in M. A formula φ is *satisfiable* if there exists M such that $M \models \varphi$; a formula is *valid* if $M \models \varphi$ for all models M.

Distribution Axioms Given a sampling statement $v \sim d$, an axiom is of the form

$$\Pr_{v \sim d}\,[\varphi^{\mathsf{ax}}] \leqslant e^{\mathsf{ub}}$$

where e^{ub} is a [0,1]-valued expression over V and φ^{ax} is a formula over V. The axiom must be true for all possible valuations of the program variables $V \setminus \{v\}$. We will use the axioms as follows: When encoding the effect of a sampling statement $v \sim d$, we can assume that $\neg\varphi^{\mathsf{ax}}$ is true, with a failure probability of at most e^{ub}. This allows us to sidestep probabilistic

reasoning and encode program semantics in our first-order theory.

Since axioms are approximations of primitive distributions, there are many possible axioms for any given distribution. In some cases, axioms may be parameterized, e.g., by the failure probability. We call parameterized axioms *axiom families*; Table 5.2 collects example axiom families for the distributions in Section 5.2.

Definition 5.12 (Laplace Axiom Family)**.** *Recall that the (discrete) Laplace distribution expression* $\text{Lap}\,(\nu_1, \nu_2)$ *is parameterized by two parameters, the* mean $\nu_1 \in \mathbb{Z}$ *and the* scale $\nu_2 \in \mathbb{R}$. *Sampling from* $\text{Lap}\,(\nu_1, \nu_2)$ *returns an integer* ν *with probability proportional to* $\exp(-|\nu - \nu_1|/\nu_2)$. *The Laplace distribution supports the following family of axioms, parameterized by a* $(0, 1]$*-valued function* f*:*

$$Pr_{\nu \sim \text{Lap}(\nu_1,\nu_2)}\left[|\nu - \nu_2| > \frac{1}{\nu_1}\log\left(\frac{1}{f(V^{\text{in}})}\right)\right] \leqslant f(V^{\text{in}})$$

Different instantiations of f *yield different axioms.*

The exponential distribution's axiom family is similar; note $\text{Exp}(\nu_1, \nu_2)$ has zero probability of returning elements smaller than ν_1, and this information is incorporated into the axiom. The Bernoulli distribution's family is parameterized by a function $f(V^{\text{in}})$ which decides whether to assume ν is *true*, *false* or treat it non-deterministically. The uniform distribution's axiom family is parameterized by a function $f(V^{\text{in}})$ returning a subset of the set defined by ν'.

Example 5.13. *Recall trace* τ_2 *(from program* `ex2`*) in Section 5.1 and Figure 5.4, which contains the statement* $a[i] \sim \text{Lap}\,(q[i], 1/\epsilon)$. *To prove correctness of* τ_2*, we instantiated the Laplace axiom family with* $f(V^{\text{in}}) = p$ *where* $p \in V^{\text{in}}$*, yielding the axiom*

$$Pr_{a[i] \sim \text{Lap}(q[i], 1/\epsilon)}\left[|a[i] - q[i]| > \frac{1}{\epsilon}\log\left(\frac{1}{p}\right)\right] \leqslant p$$

$$\begin{aligned} enc(i, v \leftarrow e) &\coloneqq v = e \wedge \omega_i = \omega_{i-1} \wedge h_i = h_{i-1} \\ enc(i, \mathsf{assume}(b)) &\coloneqq \omega_i = \omega_{i-1} \wedge h_i = (h_{i-1} \vee \neg b) \\ enc(i, v \sim d) &\coloneqq \omega_i = \omega_{i-1} + e^{\mathsf{ub}} \wedge h_i = (h_{i-1} \vee \varphi^{\mathsf{ax}}) \\ &\quad \text{given: } \Pr_{v\sim d}[\varphi^{\mathsf{ax}}] \leqslant e^{\mathsf{ub}} \end{aligned}$$

Figure 5.8: Logical encoding of statement semantics

Theorem 5.14. *Each axiom in Table 5.2 is sound: given any input state* s *and well-typed distribution expression* d*, the probability that* φ^{ax} *holds in* $s(d)$ *is at most* $s(e^{\mathsf{ub}})$*.*

The proof of Theorem 5.14 is given in Appendix C.2.

Logical Encoding

We now present our encoding for checking $\vdash_\beta \{\varphi_{\mathsf{pre}}\}\, \tau\, \{\varphi_{\mathsf{post}}\}$. First, without loss of generality, we assume that τ is in *static single assignment* (SSA) form; this ensures that variables are not assigned to more than once, simplifying our encoding. We also assume that φ_{pre} and φ_{post} are logical formulas over program variables. Our encoding explicitly maintains failure probability using a special set of real-valued variables ω_i, which encode failure probability after statement st_i along τ. In order to encode failure probability on unsatisfiable subtraces, we also use a special set of Boolean variables h_i to track if an execution was blocked by an assume statement.

The function *enc*, defined in Figure 5.8, is used to encode assignment, assume, and sampling statements; it maintains the variables ω_i, h_i and axiomatizes sampling statements using the aforementioned distribution axioms.

Consider, for instance, the encoding for assignment statements: it constrains v to e, while maintaining the same failure probability and blocked

status, ω_i and h_i. Intuitively, the semantics of assignment statements is precisely captured by our logical encoding, so assignment statements do not increase the probability of failure. In contrast, the probability of failure increases when an axiom is applied for a sampling statement. Concretely, if the axiom family $\Pr_{\nu \sim d}[\varphi^{\mathsf{ax}}] \leqslant e^{\mathsf{ub}}$ is applied, we assume that $\neg\varphi^{\mathsf{ax}}$ is true while accumulating probability of failure e^{ub}, as encoded in the constraint $\omega_i = \omega_{i-1} + e^{\mathsf{ub}}$.

The following theorem formalizes the encoding of $\vdash_\beta \{\varphi_{\mathsf{pre}}\}\, \tau \,\{\varphi_{\mathsf{post}}\}$ and states its correctness The proof is in Appendix C.2.

Theorem 5.15 (Soundness of Logical Encoding)**.** *The specification*

$$\vdash_\beta \{\varphi_{\mathsf{pre}}\}\, st_1, \dots, st_n \,\{\varphi_{\mathsf{post}}\}$$

is valid if the following formula is satisfiable:

$$\forall V, \omega_i, h_i.\ \left(\varphi_{\mathsf{pre}} \wedge \omega_0 = 0 \wedge h_0 = \mathit{false} \wedge \bigwedge_{i=1}^{n} \mathit{enc}(i, st_i)\right) \Rightarrow \varphi_p \tag{5.2}$$

where

$$\varphi_p \coloneqq (\omega_n \leqslant \beta \wedge (\neg h_n \Rightarrow \varphi_{\mathsf{post}}))$$

Observe that in the above encoding the only free symbols are the uninterpreted functions $f_1, \dots, f_m$ introduced by the axiom families used in the encoding of sampling statements. Thus, checking satisfiability involves synthesizing interpretations for $f_1, \dots, f_m$. (Equivalently, we can think of $f_1, \dots, f_m$ as existentially quantified so that we check validity of $\exists f_1, \dots, f_m \forall V \dots$.)

Example 5.16. *Recall the trace τ_1 from Section 5.1 and Figure 5.3 (program*

`ex1`), where we proved:

$$\vdash_p \{true\}\ \underbrace{x \sim \textsc{Bern}(0.5);}_{st_1}\ \underbrace{\mathsf{assume}(x);}_{st_2}\ \underbrace{y \sim \textsc{Bern}(p)}_{st_3}\ \{\neg y\}$$

Using the encoding in Theorem 5.15, we get the following formula:

$$\forall x, y, p, \omega_i, h_i.\ \left(\omega_0 = 0 \wedge h_0 = \mathit{false} \wedge \bigwedge_{i=1}^{3} \mathit{enc}(i, st_i)\right) \Rightarrow \varphi_p$$

where

$$\varphi_p \coloneqq (\omega_3 \leqslant p \wedge (\neg h_3 \Rightarrow \neg y))$$

To illustrate, $\mathit{enc}(1, x \sim \textsc{Bern}(0.5))$ *is the following constraint, using the axiom family in Table 5.2:*

$$\omega_1 = \omega_0 + \underbrace{\left\{\begin{array}{ll} 0.5 & \mathit{if}\ f_x(p) = 1 \\ 0.5 & \mathit{if}\ f_x(p) = 2 \\ 0 & \mathit{otherwise} \end{array}\right\}}_{e^{\mathsf{ub}}} \wedge (h_1 = h_0 \vee \neg\varphi^{\mathsf{ax}})$$

where

$$\neg\varphi^{\mathsf{ax}} \coloneqq (f_x(p) = 1 \wedge x) \vee (f_x(p) = 2 \wedge \neg x)$$

The proof in Section 5.1 used the interpretation $f_x(p) = 3$*, allowing* x *to take any value.*

From Synthesis to Craig Interpolation

Now that we have defined our logical constraints, we can apply *Craig interpolation* on the above encoding in Theorem 5.15 to construct the labeling functions, λ and κ, for an automaton accepting τ.

The standard notion of *sequence interpolants* (McMillan, 2006) generalizes Craig interpolants between two formulas to a sequence of unsatisfiable

formulas in first-order logic.

Definition 5.17 (Sequence Interpolants)**.** *Let $\bigwedge_{i=1}^{n} \varphi_i$ be unsatisfiable. There exists a sequence of formulas $\psi_1, \ldots, \psi_n$ such that:*

1. $\varphi_1 \Rightarrow \psi_1$ *and* $\psi_n \Rightarrow$ *false are valid,*
2. *for all* $i \in (1, n)$, $\psi_i \wedge \varphi_{i+1} \Rightarrow \psi_{i+1}$ *is valid, and*
3. $\mathit{fv}(\psi_i) \subseteq \mathit{fv}(\varphi_1, \ldots, \varphi_i) \cap \mathit{fv}(\varphi_{i+1}, \ldots, \varphi_n)$.

Note that sequence interpolation is equivalent to solving a form of recursion-free Horn clauses *(Rümmer et al., 2013); we use an interpolation-based presentation to reduce notational overhead.*

Labeling Automata via Interpolation

Suppose that we have discovered interpretations for $f_1, \ldots, f_m$ that satisfy Equation (C.1) from Theorem 5.15. This implies that the following formula, which is Equation (C.1) after negating it and instantiating $f_1, \ldots, f_m$ with their interpretations, is unsatisfiable:

$$\left(\varphi_{\mathsf{pre}} \wedge \omega_0 = 0 \wedge \bigwedge_{i=1}^{n} \mathit{enc}(i, \mathit{st}_i)\right) \wedge \neg(\omega_n \leqslant \beta \wedge (\neg h_n \Rightarrow \varphi_{\mathsf{post}}))$$

It follows that we can construct a *sequence of Craig interpolants* for the following problem:

$$\underbrace{\varphi_{\mathsf{pre}} \wedge \omega_0 = 0}_{\varphi_0} \wedge \bigwedge_{i=1}^{n} \underbrace{\mathit{enc}(i, \mathit{st}_i)}_{\varphi_i} \wedge \underbrace{\neg(\omega_n \leqslant \beta \wedge (\neg h_n \Rightarrow \varphi_{\mathsf{post}}))}_{\varphi_{n+1}}$$

Every interpolant ψ_i encodes the set of reachable states and the failure probability after executing the first i program statements beginning from a state in φ_{pre}. The free-variable condition for interpolants implies that the

only free variables in ψ_i are h_i, ω_i, and live program variables after the ith statement. The challenge is that interpolants describe both the program state invariants and the failure probability invariants, corresponding to the λ and κ needed to label the failure automaton. Fortunately, these labels can be extracted from the interpolants. The following theorem formalizes the transformation and states its correctness. The proof is in Appendix C.3.

Theorem 5.18 (Well-Labelings from Interpolants)**.** *Let $\{\psi_i\}_i$ be the interpolants computed as shown above. Let $A_\tau = \langle Q, \delta, \lambda, \kappa \rangle$ be the failure automaton that accepts only the trace $\tau = \mathit{st}_1, \ldots, \mathit{st}_n$, i.e.,*

$$\delta = \left\{ q^{\mathsf{in}} \xrightarrow{\mathit{st}_1} q_1, q_1 \xrightarrow{\mathit{st}_2} q_2, \ldots q_{n-1} \xrightarrow{\mathit{st}_n} q^{\mathsf{ac}} \right\}$$

Set the labeling functions as follows:

1. $\lambda(q^{\mathsf{in}}) \coloneqq \varphi_{\mathsf{pre}}$ *and* $\kappa(q^{\mathsf{in}}) \coloneqq 0$.
2. $\lambda(q_i) \coloneqq \exists \omega_i.\, \psi_i[h_i \mapsto \mathit{false}]$ *and* $\lambda(q^{\mathsf{ac}}) \coloneqq \exists \omega_n.\, \psi_n[h_n \mapsto \mathit{false}]$.
3. $\kappa(q_i) \coloneqq f(V^{\mathsf{det}})$, *where* $f(V^{\mathsf{det}})$ *is the function that returns, for any valuation of* V^{det}*, the largest value of* ω_i *that satisfies* $\exists V \setminus V^{\mathsf{det}}.\, \exists h_i.\, \psi_i$. *For* $\kappa(q^{\mathsf{ac}})$*, we use* $\exists V \setminus V^{\mathsf{in}}.\, \exists h_n.\, \psi_n$.

Then, A_τ *is well-labeled and implies* $\vdash_\beta \{\varphi_{\mathsf{pre}}\}\, \tau\, \{\varphi_{\mathsf{post}}\}$.

Notice that for λ we set h_i to be *false*, since we are only interested in states that pass assume statements (reachable states). We existentially quantify ω_i, as it is not a program variable. Also notice the technicality in constructing κ; this arises because the interpolant is a *relation* over values of ω_i and V^{det}, while the label of $\kappa(q_i)$ is technically a *function* from V^{det} to [0,1]. In practice, we need not construct the function f; we can perform all needed checks using relations.

5.6 Implementation and Case Studies

We have implemented our approach atop the Z3 SMT solver (de Moura and Bjørner, 2008). We encode statements using the following first-order theories: linear arithmetic, uninterpreted functions, and arrays. Below, we describe our implementation; we refer to Appendix C.6 for further details.

Algorithmic Strategy Our implementation is a determinization of the algorithm presented in Section 5.4. To ensure that we get tight upper bounds on failure probability, our implementation aggressively tries to apply the MERGE rule—recall that the MERGE rule allows us take the maximum failure probability across two automata, instead of the sum. Specifically, we modify the rule TRACE to return a set of traces $\tau_1, \ldots, \tau_n \in \mathcal{L}(P) \cap \overline{\mathcal{L}(\mathcal{A})}$. Then, we attempt to simultaneously label all traces with the same interpolants at nodes pertaining to the same control location. To ensure that we compute similar interpolants across traces, we use the same distribution axiom for the same sampling instruction in all traces it appears in. Finally, we attempt to apply the rule GENERALIZE to create cycles in the resulting automaton.

Discovering Axioms Given a formula of the form $\exists f.\, \forall X.\, \varphi$, we check its validity using a propose-and-check loop: (*i*) we propose an interpretation of f and then (*ii*) check if $\forall X.\, \varphi$ is valid with that interpretation using the SMT solver (more on this below). The first step proposes interpretations of f of increasing size, e.g., for a unary function $f(x)$, it would try $0, 1, x, x+1$, etc. As we shall see, even for complex randomized algorithms from the literature, the required axioms are syntactically simple, so this simple strategy works rather well.

Checking Validity The case studies to follow make heavy use of nonlinear arithmetic (e.g., $xy/z+u > 0$) and transcendental functions (namely,

log). Non-linear theories are generally undecidable. To work around this fact, we implement an incomplete formula validity checker using an eager version of the *theorem enumeration* technique recently proposed by Srikanth et al. (2017). First, we treat non-linear operations as uninterpreted functions, thus overapproximating their semantics. Second, we strengthen formulas by instantiating *theorems* about those non-linear operations. For instance, the following theorem relates division and multiplication: $\forall x, y.\, y > 0 \Rightarrow xy/y = x$. We then instantiate x and y with terms over variables in the formula. Since there are infinitely many possible instantiations of x and y, we restrict instantiations to terms of size 1, i.e., variables/constants.

Our implementation uses a fixed set of theorems about multiplication, division, and logarithms. These are instantiated for every given formula, typically resulting in ~1000 additional conjuncts.

Interpolation Technique Given the richness of the theories we use, we found that existing proof-based interpolation techniques either do not support the theories we require (e.g., the MathSAT solver (Cimatti et al., 2013)) or fail to find generalizable interpolants, e.g., cannot discover quantified interpolants (e.g., Z3). As such, we implement a *template-guided* interpolation technique (Albarghouthi and McMillan, 2013; Rummer and Subotic, 2013), where we force interpolants to follow syntactic forms that appear in the program. Specifically, for every Boolean predicate φ appearing in the program, the specification, or the axioms, we create a template φ^t by replacing its variables with placeholders, denoted $\bullet_i$. For instance, given $x > y$, we generate the template $\bullet_1 > \bullet_2$.

Given a set of templates, our interpolation technique *synthesizes* a conjunction of instantiations of those templates, where each wildcard is replaced by a well-typed term over formula variables. We bound the size of terms instantiating wildcards (e.g., to size 1), and proceed by finding the

smallest possible interpolant in terms of number of conjuncts. If no such interpolant can be found, we increase the bound on term size and repeat.

Case Studies in Privacy-Preserving Algorithms

Differential privacy (Dwork et al., 2006) is a strong probabilistic property modeling statistical data privacy that adds random noise at key points in the computation. Sophisticated differentially private algorithms are known for a wide variety of common data analyses, and differential privacy is starting to see deployments in both industry (Erlingsson et al., 2014; Johnson et al., 2018a) and government (Abowd and Schmutte, 2017; Haney et al., 2017).

Intuitively, more random noise yields stronger privacy guarantees at the expense of accuracy—the noisy answers may be too far from the exact answers to be of any practical use. Therefore, the designer of a differentially private algorithm aims to maximize accuracy of the computed results while achieving some target level of privacy. We now consider a number of algorithms from the differential privacy literature and demonstrate our technique's ability to automatically prove their accuracy guarantees. The algorithms and their specifications are shown in Figures 5.9 to 5.14 and described below; Table 5.3 provides runtime and other statistics, which we discuss later in this section.

Randomized Response (randResp)

One of the oldest randomized schemes for protecting privacy is *randomized response*, proposed by Warner (1965) decades before the formulation of differential privacy. In the typical setting, an individual has a single bit (0 or 1) as their private data, representing e.g. the presence of some disease or genetic marker. Under randomized response, the individual flips two fair coins: if the first result is heads, the individual reports their bit honestly,

```
⊢0.25 {true}
let randResp priv = do {
    r ~ Unif({00,01,10,11})
    if (fst r == 1) then
        ans = priv
    else
        ans = snd r
    return ans
} {ans ⟺ priv}
```

Figure 5.9: Randomized Response (randResp) algorithm

Algorithm	Axiom(s) synthesized	PA	TI	Time
randResp	$\mathtt{priv} \iff \mathtt{snd}(\mathtt{r})$	162	0	2
noisySum	$\lvert Q\rvert/p$	5	5496	98
noisyMax	$\lvert Q\rvert/p$	4	1768	33
expMech	$\lvert R\rvert/p$	3	1768	27
aboveT	$2/p$ and $2\lvert Q\rvert/p$	22	752	23
sparseVec	$3/p$, $3\lvert Q\rvert/p$, and $3/p$	941	1330	97

Table 5.3: Results on private algs. PA: # of proposed axioms; TI: # of theorem instantiations; time is in sec.

otherwise they ignore their private bit and report the result of the second flip. In this way, randomized response guarantees a degree of privacy by introducing plausible deniability—an individual's reported bit could have been the result of chance. At the same time, randomized response guarantees a weak notion of accuracy, as the output is biased towards the true private bit with probability $3/4$.

We encode randomized response as in Figure 5.9 and prove the accuracy guarantee. In the code, *priv* is the individual's private bit. The program draws two bits uniformly and then decides what to return; *fst* and *snd* extract the results of the first and second bits, respectively. The accuracy guarantee states that the returned answer is equal to the true private bit, except with probability at most $1/4$. Our implementation syn-

$\vdash_p \{\epsilon > 0\}$
```
let noisySum Q d ϵ = do {
    s, i = 0, 1
    while (i < |Q|) do {
        q = Q[i](d)
        a[i] ~ Lap(q, 1/ϵ)
        s = s + a[i]
        i = i + 1
    }
    return s
}
```
$\{|s - s^*| \leqslant |Q|/\epsilon \cdot \log(|Q|/p)\}$

Figure 5.10: Noisy Sum (noisySum) algorithm; $s^* \coloneqq \sum_{j=1}^{|Q|} Q[j](d)$

thesizes the axiom *priv* $\iff$ *snd*(r); this ensures that the second bit has the same value as *priv*, so if the first bit is 0 and the else branch is taken, the algorithm is forced to return the right result, with a failure probability of 1/4.

Noisy Sum (noisySum)

Our next algorithm computes the sum of a set of numeric queries, adding noise to the answer of each query in order to ensure differential privacy. This is a simplified version of the private counters by Chan et al. (2011) and Dwork et al. (2010), which are used to publish aggregate statistics privately, e.g., total number of website visitors.

The noisySum program (Figure 5.10) takes three inputs: a set Q of integer-valued queries (encoded as an integer array where index i holds the result of $Q_i(d)$), a database d holding the private data, and a parameter $\epsilon \in \mathbb{R}$ representing the desired level of privacy. The program populates an integer array a with answers to each query, with noise drawn from the Laplace distribution with scale controlled by ϵ; smaller ϵ is more private, but requires more noise. Finally, the output is the sum of all noisy answers.

The accuracy guarantee bounds how far the noisy sum deviates from

```
⊢p {ϵ > 0}
let noisyMax Q d ϵ = do {
    b, max, i = ⊥, ⊥, 1
    while (i <= Q) do {
        q = Q[i](d)
        a[i] ~ Lap(q, 2/ϵ)
        if (a[i] > max || b = ⊥) then
            b = i
            max = a_i
        i = i + 1
    }
    return best
} {∀j ∈ [1,|Q|]. Q[b](d) ⩾ Q[j](d) − 4/ϵ log(|Q| /p)}
```

Figure 5.11: Report Noisy Max (noisyMax) algorithm

the true sum with failure probability p, where p is a parameter. Our implementation synthesizes an axiom for each Laplace sampling, setting the failure probability to be $p/|Q|$ each time. Therefore, at step i,

$$|a_i - Q_i(d)| \leqslant \frac{1}{\epsilon} \log(|Q| / p)$$

Since there are $|Q|$ iterations, after the loop exits we have

$$\left| s - \sum_{j}^{|Q|} Q_j(d) \right| \leqslant \frac{|Q|}{\epsilon} \log(|Q| / p)$$

with a failure probability of at most $|Q| \cdot \frac{p}{|Q|} = p$.

Report Noisy Max (noisyMax) and Exponential Mechanism (expMech)

Our next pair of algorithms select an approximate maximum element from a set of private data.

In Report Noisy Max (Dwork and Roth, 2014), the algorithm is presented with a set Q of integer queries, a private database d, and a privacy level ϵ (Figure 5.11). The algorithm then evaluates each query on d and

```
⊢p {ϵ > 0}
let expMech R u D ϵ = do {
    b, max = ⊥, 0
    for (r in R) do {
        util = u(r, d)
        nu = Exp(util, 2/ϵ)
        if (nu > max || b = ⊥) then
            b, max = r, nu
    }
    return b
} {∀j ∈ R. u(b, d) ⩾ u(j, d) − 2/ϵ · log(2 |R| /p)}
```

Figure 5.12: Discrete Exponential mechanism (expMech)

adds Laplace random noise to protect privacy. Finally, the index of the query with the largest noisy value is returned. For example, if each query counts the number of patients with a certain disease, then Report Noisy Max will report a disease that may not be true most prevalent disease, but whose count is not too far from the true maximum count.

The postcondition states that the answer of the returned query Q_b is not too far below the answer of the actual maximum query. To achieve failure probability p, our implementation synthesizes an axiom for the Laplace sampling statement with failure probability $p/|Q|$. Since the loop executes $|Q|$ times, we establish that the postcondition holds with probability p. To do so, the interpolation engine discovers a number of key facts; we outline two of them:

$$\forall j \in [1, i).\, |a_j - Q_j(d)| \leqslant \frac{2}{\epsilon} \log \frac{|Q|}{p} \quad \text{and} \quad \forall j \in [1, i).\, a_b \geqslant a_j$$

The first formula specifies that, for every element of j of a, its distance from the corresponding valuation of $Q_j(d)$ is bounded above by $2/\epsilon \log |Q|/p$—this follows directly from the choice of distribution axiom. The second formula states that the best element is indeed larger than all previously seen ones. Upon loop exit, these facts, along with others, are sufficient to

imply the postcondition. Notice that the $2/\epsilon \log |Q|/p$ in the first formula weakens to $4/\epsilon \log |Q|/p$ in the postcondition. This is due to the two-sided error introduced by the absolute value in the Laplace axiom. The proof computed for noisyMax is presented in detail in Appendix C.6.

The algorithm expMech is a discrete version of the seminal Exponential mechanism (McSherry and Talwar, 2007), a fundamental algorithm in differential privacy (Figure 5.12). This algorithm is used to achieve differentially privacy in non-numerical queries, as well as a mechanism for achieving certain notions of fairness in decision-making algorithms(Dwork et al., 2012). expMech takes a set R of possible output elements, a *utility function u* mapping each element of R and private database to a numeric score, a private database d, and privacy parameter ϵ. The algorithm aims to return an element of R that has large utility on the given database. expMech differs from noisyMax through the use of the exponential distribution; because the exponential distribution never produces results lower than the shift, the accuracy bound for the expMech is better. The distance to the true maximum is at most $\frac{2}{\epsilon} \log(2|R|/p)$ instead of $\frac{4}{\epsilon} \log(|Q|/p)$, with failure probability at most p. To prove this, our implementation synthesizes an axiom analogous to that used for noisyMax.

Above Threshold (aboveT) and Sparse Vector Mechanism (sparseVec)

A useful differential privacy primitive is to return the first query in a list with a numeric answer (approximately) above some given threshold, ignoring queries with small answers. Our final two privacy examples do just this. The Above Threshold algorithm (Dwork and Roth, 2014) takes a list Q of queries, a private database d, a numeric threshold T, and the target privacy level ϵ (Figure 5.13). First, the program computes a noisy threshold t by adding noise to the true threshold T. The program loops through the queries, comparing the noisy answer of each query to the noisy threshold. If the noisy answer is above the noisy threshold,

```
⊢p {ϵ > 0}
let aboveT Q d T ϵ = do {
    i, done = 1, false
    t ~ Lap(T, 1/ϵ)
    while (i <= |Q| && !done) do {
        q = Q[i](d)
        a ~ Lap(q, 2/ϵ)
        if (a > t) then
            done = true
        i = i + 1
    }
    ans = done ? i - 1 : ⊥
    return ans
} {ans ≠ ⊥ ⇒ φ⊤) ∧ (ans = ⊥ ⇒ φ⊥)}
```

$$\varphi_\top := \begin{cases} \forall j \in [1, ans).\, Q[j](d) \leqslant T + 2/\epsilon \cdot \log(2\,|Q|\,/p) + 1/\epsilon \cdot \log(2/p) \\ Q[ans](d) \geqslant T - 2/\epsilon \cdot \log(2\,|Q|\,/p) - 1/\epsilon \cdot \log(2/p) \end{cases}$$

$$\varphi_\perp := \forall j \in [1, |Q|].\, Q[j](d) \leqslant T + 2/\epsilon \cdot \log(2\,|Q|\,/p) + 1/\epsilon \log(2/p)$$

Figure 5.13: Above Threshold (aboveT) algorithm

the program sets the flag *done* and exits the loop. Finally, the algorithm returns the index of the approximately above threshold query, or a default value $\perp$ if no such query was found.

The accuracy guarantee requires some care. There are two cases: the returned value is either a query index, or $\perp$. In the first case, q_{ans} should have true value not too far below the exact threshold T, and all prior queries should have true value not too far above T. In the second case, no query was found to be above threshold after adding noise, so no true answer should be too far above T. To prove this property, we synthesize axioms for the Laplace sampling instructions with different failure probabilities: $\frac{p}{2}$ for the threshold sampling, and $\frac{p}{2|Q|}$ for each loop sampling. There is one threshold sampling and at most $|Q|$ loop iterations, so the total failure probability is at most $\frac{p}{2} + |Q| \cdot \frac{p}{2|Q|} = p$.

A slightly more involved variant of this algorithm, called Numeric

```
⊢p {ϵ > 0}
let sparseVec Q d T ϵ = do {
    i, done = 1, false
    t ~ Lap(T, 1/ϵ)
    while (i <= |Q| && !done) do {
       q = Q[i](d)
       a ~ Lap(q, 2/ϵ)
       if (a > t) then
          noisy ~ Lap(q, 1/ϵ)
          done = true
       i = i + 1
    }
    ans = done ? (i - 1, noisy) : ⊥
} {(ans ≠ ⊥ ⇒ φ'⊤) ∧ (ans = ⊥ ⇒ φ'⊥)}
```

Figure 5.14: Sparse Vector (sparseVec) algorithm; the definitions of $\varphi'_\top$ and $\varphi'_\bot$ mirror that of $\varphi_\top$ and $\varphi_\bot$ from Figure 5.13, and so are elided here

Sparse Vector (Dwork and Roth, 2014), also returns a noisy answer to the above threshold query along with the query's index (Figure 5.14). Again, the accuracy property describes the two cases—above threshold query found, and no above threshold queries. In both cases, the noisy query answer should be close to the true answer. The proof proceeds much like in the simpler variant, adjusting the failure probabilities when applying axioms in order to take the additional noisy answer sampling into account.

Discussion of Results

Table 5.3 summarizes the results of applying our implementation to the above algorithms. The table lists the synthesized axiom per sampling statement—recall that our implementation strategy forces different instances of a sampling statement to use the same axiom. Additionally, we list the number of proposed and checked axioms (PA), the largest number of theorem instantiations for dealing with non-linear arithmetic (TI), and the total time in seconds.

Consider the aboveT algorithm. The implementation attempts 22 differ-

ent pairs (because there are two sampling statements) of axioms. Table 5.3 lists the synthesized interpretation of the function $f(V^{in})$ for the first and second sampling statements. The implementation discovers the axiom that assigns a failure probability $p/2$ for the first sampling statement and $p/(2|Q|)$ for the second sampling statement. Proving accuracy of aboveT takes 23 seconds and 752 theorems are instantiated to interpret non-linear arithmetic. Notice that noisySum takes the longest amount of time, even though it only attempts 5 axioms. This is due to the large number (~5500) of theorem instantiations. For sparseVec, the implementation proposes 941 axioms before discovering the shown axioms.

To the best of our knowledge, no existing tools can automatically reason about the algorithms and accuracy properties we have discussed here. The algorithms we considered are small yet sophisticated. As the number of sampling statements increases, the space of possible axioms grows combinatorially, impacting synthesis performance. As research into constraint-based program synthesis progresses, our approach can directly benefit from these developments.

Case Study in Unreliable Hardware

To demonstrate our approach's versatility, we consider another possible application: analyzing programs executing on approximate hardware, which is unreliable but efficient.

We use the program searchRef from the *Rely* system by Carbin et al. (2013), shown in Figure 5.15, which implements a pixel-block search algorithm from x264 video encoders. The program receives a constant number of pixel blocks ($nblocks = 20$) of size 16×16 ($height \times width$).

This program is expected to provide adequate video encoding despite potential hardware failures. Rely's programming model exposes unreliable arithmetic operations, denoted with a U (e.g. $x +_U y$), which may fail with small probability (say, 10^{-7}). Reading from variables typed as

```
let nblocks, height, width = 20, 16, 16 in
let searchRef pblocks, cblock =
    let reliable = i, j, k in
    let unreliable = minssd, minblock, ssd, t, t1, t2 in do {
        i = 0
        while (i < nblocks) do {
            ssd, j = 0, 0
            while (j < height) do {
                k = 0
                while (k < width) do {
                    t1, t2, = pblocks[i, j], cblock[j]
                    t = t1 −u t2
                    ssd = ssd +u (t ×u t)
                    k = k + 1
                }
                j = j + 1
            }
            if (ssd <u minssd) then
                minssd, minblock = ssd, i
            i = i + 1
        }
        return minblock
    }
```

Figure 5.15: Reliable computing example (Carbin et al., 2013)

unreliable may also fail with a small probability. Rely assumes loops over unreliable data have a constant bound on the number of iterations, so these loops can be unrolled.

Our goal is to prove the probability of a reliable execution is at least 0.99, where reliability implies no failures along the execution (note that Rely multiplies this probability by the reliability of the inputs (*pblocks*, *cblock*)—this does not impact the analysis). To do so, we analyze a version of the program instrumented with a Boolean flag *unrel*, which is initialized to *false*. We model each unreliable operation by adding a sampling from the Bernoulli distribution to determine whether the operation fails. For

instance, a read $y \leftarrow x$ from an unreliable x is transformed into

$$y \leftarrow x;\ \mathit{unrel} \leftarrow \mathit{unrel} \vee \textsc{Bern}(10^{-7})$$

We then use to prove $\vdash_{0.01} \{\mathit{true}\}$ searchRef $\{\mathit{unrel} = \mathit{false}\}$.

Unlike Rely, we do not assume independent failures. Our analysis thus gives a more conservative estimate of failure probability, but, as a benefit, retains soundness even if failures are correlated. Nevertheless, we are able to prove that the program is reliable with probability $\geqslant 0.992832$, compared to the 0.994885 computed by Rely. Moreover, since our approach is symbolic, we can prove a *symbolic* reliability bound as a function of the number of blocks and their size. This allows us to ask: *how many blocks can we use, and how large, and still be reliable?* We automatically establish the parameterized failure probability $1.4 \cdot 10^{-6} \cdot \texttt{nblocks} \cdot \texttt{height} \cdot \texttt{width}$, describing how program parameters affect reliability. For instance, we can increase the number of blocks to 25 and still maintain $\geqslant 0.99$ reliability, or quadruple the size of each block to 32^2 pixels and get $\geqslant .97$ reliability. In both settings, our approach completes the proof in less than 2 seconds.

5.7 Related Work

Interpolation & Trace Abstraction In software verification, interpolants were first used for constructing predicate abstract domains in *counterexample-guided abstraction refinement* (CEGAR) (Henzinger et al., 2004). McMillan's work on lazy abstraction with interpolants (McMillan, 2006) used proofs of correctness of program traces to directly construct Hoare-style annotations. By unrolling the program's CFG into a tree and adding annotations, he showed how to generalize a tree of paths into an automaton by adding back edges, proving the correctness of infinitely many traces.

Our approach is inspired by work by Heizmann et al. (2010, 2013, 2009), which provided an insightful and general view of interpolation-based ver-

ification through the lens of automata. Elegance aside, the automata view better suits our probabilistic setting: in McMillan's original formulation, a program can be unrolled into a tree, as paths with common prefixes can be combined. In our setting this combining is more subtle—our $\mathbb{\Lambda}$ is a restricted version. Further, the automata view allows us to maintain sets of traces separately and sum up their probabilities of failure.

Deductive Probabilistic Verification Deductive verification techniques for probabilistic programs include probabilistic Hoare logics (Rand and Zdancewic, 2015; Barthe et al., 2018; den Hartog, 2002; Chadha et al., 2007) and the lightweight probabilistic logic of Barthe et al. (2016c) our technique is closely related to, just as the classical interpolation-based techniques mirror Hoare-style proofs. Deductive techniques are highly expressive, but the complex proofs typically must be constructed manually or in an interactive setting. In contrast, our approach has the advantage of automation.

Pre-Expectation Calculus The pre-expectation calculus and associated predicate transformers (Kozen, 1985; Morgan et al., 1996) can prove properties of probabilistic programs, but have practical obstacles to full automation. Computing pre-expectations across sampling instructions yields an integral over the distribution being sampled from. Complex distributions, like the infinite-support Laplace distribution, yield correspondingly complex integrals that are difficult to reason about. Any automation of the pre-expectation calculus will need to establish algebraic properties about these mathematical expressions. Our use of distribution axioms (Section 5.5) obviates the need to reason directly about integrals via a reduction to synthesis.

Martingales Martingales—probabilistic analogues of loop invariants—are used in automated tools to prove termination conditions (Chakarov

and Sankaranarayanan, 2013; Chatterjee et al., 2016a,b; McIver et al., 2018) and properties of expected values (Barthe et al., 2016a). Automated martingale synthesis techniques are restricted to linear or polynomial invariants, which alone are unable to prove the accuracy properties we are interested in.

Probabilistic Model Checking Probabilistic model checking is perhaps the most well-developed technique for automated reasoning about probabilistic systems. Traditionally, it focused on temporal properties of *Markov Decision Processes* (MDP)—surveys by Kwiatkowska et al. (2010) and Katoen (2016) overview the current state of the art.

Our program model can be cast as an infinite-state MDP, with nondeterminism at program entry to pick an initial state. There have been a number of abstraction-based techniques for reducing the size of large (or infinite) MDP (Kattenbelt et al., 2009, 2010; Hermanns et al., 2008). To our knowledge, existing works cannot handle the programs and properties we consider here. The general limitation is the inability of existing model checking techniques to handle distribution expressions—e.g., a Laplace whose scale is a parameter—and failure probabilities that are expressions. Probabilistic CEGAR (Hermanns et al., 2008) uses a guarded-command language where probabilistic choice is a real-value determining the probability of executing each command. Other techniques limit distribution expressions to finite distributions with constant parameters (Kattenbelt et al., 2009).

Teige and Fränzle (2011) consider interpolation in *stochastic Boolean satisfiability* (Littman et al., 2001), where formulas contain existential and probabilistic quantifiers. The approach has been used for generalizing bounded encodings of finite-state MDP, in an analogous fashion to the original work on interpolation-based model checking (McMillan, 2003).

Other Analyses Probabilistic abstract interpretation (Cousot and Monerau, 2012) generalizes the abstract interpretation framework to a probabilistic setting; other techniques can be cast in this framework (Monniaux, 2000, 2001, 2005; Claret et al., 2013). Recently, Wang et al. (2018) presented PMAF, an elegant algebraic framework for constructing analyses of probabilistic programs. The approach is rather general, accepting recursive programs and supporting interprocedural analyses. Unlike PMAF, whose results depend highly on the expressiveness of the chosen abstract domain, our technique constructs abstractions on demand, *à la* interpolation-based verification, at the risk of never generalizing. PMAF instantiations considered by Wang et al. (2018) cannot prove our target accuracy properties, but alternative instantiations might achieve something similar.

Another line of work reduces probabilistic verification to a form of counting (Albarghouthi, 2017; Chistikov et al., 2015; Belle et al., 2015; Mardziel et al., 2011). To compute the probability that a formula is SAT, these techniques count the number of satisfying assignments—or perform numerical volume estimation in the infinite-state case. While these techniques can compute very precise—often exact—probabilities, they target simpler program models. Specifically, programs have no inputs, probability distribution are not parameterized, and loops are handled via unrolling.

Our technique is related to works verifying relational probabilistic properties, including differential privacy and uniformity (Albarghouthi and Hsu, 2018b,a). These systems encode the space of *coupling proofs* as a constraint-based synthesis problem. Our technique handles different properties, but shares the high-level design principle of reducing probabilistic reasoning to logical reasoning.

Computer algebra and symbolic inference methods (e.g., (Gehr et al., 2016; Cusumano-Towner et al., 2018; Narayanan et al., 2016)) have been applied to probabilistic programs in different domains (e.g., (Gehr et al.,

2018)). While these tools can automatically generate symbolic representations of output distributions, proving properties about these distributions remains challenging. Modern implementations use a variety of custom heuristics and reduction strategies to try to simplify complex algebraic terms, a computationally-expensive task.

6 EXPLOITING SYMMETRIES IN TERM ALGEBRAS

Program synthesis techniques are powerful and useful tools. In this dissertation alone, we have seen how *effective* program synthesis can be at lowering the burden-of-knowledge for data access by generating useful programs satisfying scalability and privacy constraints (Chapters 3 and 4); even the proof technique in Chapter 5 is only made possible by the use of program synthesis to generate distribution axioms.

While we frame synthesis as the enumeration of simple syntactic terms, we often have more information about the function symbols than just their input and output sorts: knowledge of their partial semantics can be encoded into equations over terms to describe properties such as commutativity, distributivity, or idempotence. Fundamentally, these properties relate two candidate terms together—e.g., $x + y$ and $y + x$ via commutativity—and form a semantic symmetry that can be exploited to limit the synthesis search domain.

In this chapter, we present the *synthesis modulo equations problem*, and explore how equivalence reduction uses equations to avoid exploring redundant program terms. To conclude, we explore the implementation and performance implications of several versions of equivalence reduction as applied to bottom-up and top-down synthesis techniques. The contents of this chapter are based on the work of Smith and Albarghouthi (2019a).

6.1 Introduction to Equivalence Reduction

Let us frame program synthesis as a teacher-learner model, where the learner (the synthesizer) proposes a program p and the teacher (the verifier) answers with *yes*/*no*, indicating whether $p \models \phi$ or not. Our goal is to make the learner smarter: we want to reduce the number of questions the learner needs to ask before arriving at the right answer.

Consider the following string-manipulation programs:

```
let p_1 x = swap (lower x)
let p_2 x = upper x
```

where `swap` turns all uppercase characters lowercase (and vice versa), and where `upper` and `lower` turn all characters uppercase or lowercase, respectively. A smart learner would know that turning all characters lowercase and then applying `swap` is the same as simply applying `upper`. Therefore, the learner would only need to ask about *one* of the programs `p_1` and `p_2`. Formally, the learner is equipped with the following piece of information connecting the three functions:

$$\forall x. \, \texttt{swap}(\texttt{lower}(x)) = \texttt{upper}(x)$$

One also imagines a variety of other semantic knowledge that a learner can leverage, such as properties of specific functions (e.g., idempotence) or relational properties over combinations of functions (e.g., distributivity). Such properties can be supplied by the developer of the synthesis domain, or discovered automatically using tools like QuickSpec (Claessen et al., 2010) or Bach (Smith et al., 2017).

Equivalence Reduction

Universally quantified formulas like the one above form *equational specifications*: they define some (but not all) of the intended behaviors of the function symbols, as well as relations between them. The equations partition the space of programs into equivalence classes, where each equivalence class contains all equationally equivalent programs. The learner needs to detect when two programs are in the same equivalence class and only ask the teacher about one *representative* per equivalence class. To do so, we utilize the equations to define a *normal form* on programs, where equivalent programs all simplify to the same normal form. By structuring

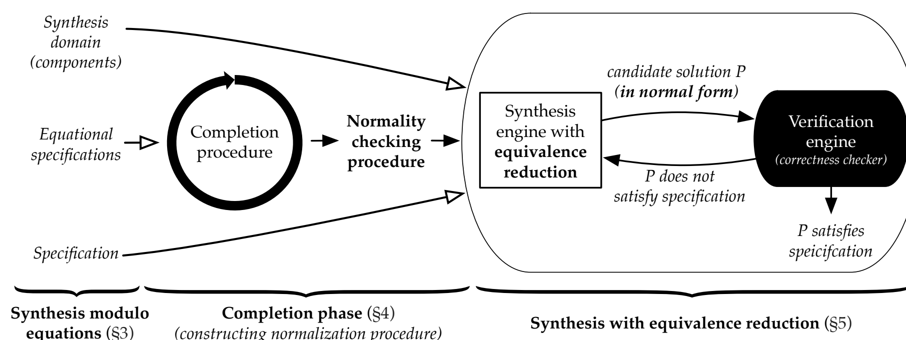


Figure 6.1: Overview of synthesis with equivalence reduction

the learner to only consider programs in normal form, we ensure that no redundant programs are explored, potentially creating drastic reductions in the search space. We call this process *program synthesis with equivalence reduction*.

By restricting the partial semantic information in the form of equations, we can leverage standard completion algorithms, e.g. Knuth-Bendix completion (Knuth and Bendix, 1983), to construct a term-rewriting system (TRS) that is confluent, terminating, and equivalent to the set of equations. The result of completion is a decision procedure that checks whether a program term p is a representative of an equivalence class—i.e., whether p is in normal form. The difficulty in completion is that constructing such a decision procedure is *undecidable*, as equations are rich enough to encode Turing machines. Nonetheless, significant progress has been made in completion algorithms and *termination proving* (e.g., (Wehrman et al., 2006; Giesl et al., 2006; Winkler and Middeldorp, 2010)), which is used for completion.

Given a normalizing TRS resulting from completion, we will show in Section 6.5 how to incorporate it into existing synthesis techniques in order to prune away redundant fragments of the search space and accelerate synthesis.

6.2 Overview of Synthesis Modulo Equations

Figure 6.1 provides an overview of our proposed synthesis approach. A *synthesis modulo equations* problem is defined by two inputs. First, we are given a synthesis problem (Definition 2.17), which contains a *synthesis domain* described as a term algebra T_Σ. Second, we expect *equational specifications*, which are equations over terms in T_Σ. For example, an equation might specify that a function symbol f is associative ($f(x, y) = f(y, x)$), or that two function symbols f and g are inverses of each other ($f(g(x)) = g(f(x)) = x$). Below, we describe the various components in Figure 6.1 in detail.

Synthesis modulo equations problem

Consider the function symbols in Table 6.1. The symbols, with their default interpretations, describe basic integer operations as well as a number of functions over strings and *byte arrays* that form a subset of Python 3.7's string API. We provide the canonical interpretation of some of the non-standard symbols. `split(x, y)` splits the string `x` into a list of strings using the *delimiter* `y`, e.g.:

```
split "hizuwzmadison" "z" = ["hi", "uw", "madison"]
```

The function `join(x, y)` concatenates a list of strings `x` using the delimiter `y`. Functions `encode`/`decode` convert between strings and UTF-8 byte arrays.

Equational Specifications

Even for a simple set of components, there is a considerable amount of latent domain knowledge that we can exploit during synthesis. Table 6.2 provides a partial view of the equations that we can utilize for the components in Table 6.1. The variables x, y, z are implicitly universally quantified.

Component name	Description
Integers	
$+$: `int` $\to$ `int` $\to$ `int`	integer addition
$-$: `int` $\to$ `int` $\to$ `int`	integer subtraction
$*$: `int` $\to$ `int` $\to$ `int`	integer multiplication
`abs` : `int` $\to$ `int`	absolute value
Strings and byte arrays	
$++$: `string` $\to$ `string` $\to$ `string`	str concatenation
`len` : `string` $\to$ `int`	str length
`swap` : `string` $\to$ `string`	swap upper/lowercase
`split` : `string` $\to$ `string` $\to$ [`string`]	split str w/ delimiter
`join` : [`string`] $\to$ `string` $\to$ `string`	concat. list w/ delimiter
`encode` : `string` $\to$ `utf`	encode str as UTF-8
`decode` : `utf` $\to$ `string`	decode UTF-8 into str

Table 6.1: Example synthesis domain over integers and strings

Equational specifications	
$x + y = y + x$	$\texttt{len}(x ++ y) = \texttt{len}(x) + \texttt{len}(y)$
$(x + y) + z = x + (y + z)$	$\texttt{swap}(\texttt{swap}(x)) = x$
$x * (y + z) = (x * y) + (x * z)$	$\texttt{join}(\texttt{split}(x, y), y) = x$
$\texttt{abs}(\texttt{abs}(x)) = \texttt{abs}(x)$	$\texttt{decode}(\texttt{encode}(x)) = \texttt{encode}(\texttt{decode}(x)) = x$

Table 6.2: Partial list of equations for integer and string components

Consider, for instance, the following equation:

$$\forall x, y.\, \texttt{join}(\texttt{split}(x, y), y) = x$$

This equation connects `split` and `join`: splitting a string x with delimiter y, then joining the result using the same delimiter y, produces the string$x$. In other words, `split` and `join` are *inverses*, assuming a fixed delimiter y.

Other equations specify, e.g., that `abs` is idempotent ($\forall x.\, \texttt{abs}(\texttt{abs}(x)) = \texttt{abs}(x)$) or that the function `swap` is an involution ($\forall x.\, \texttt{swap}(\texttt{swap}(x)) = x$).

Completion Phase

Two programs are equivalent with respect to the equations if we can use the equations to *rewrite* one into the other. Given the set of equations,

we would like to be able to partition the space of programs into equivalence classes, where two programs are in the same equivalence class if and only if they are equivalent with respect to the equations. By partitioning the space into equivalence classes, we can ensure that we only consider one representative program per equivalence class. The more equations we add—i.e., the more domain knowledge we have—the larger our equivalence classes are.

Given the set of equations, the completion phase generates a TRS that transforms any program term into its normal form—the representative of its equivalence class. The process of determining term equality modulo equations is generally undecidable (Novikov, 1955), since equations are rich enough to encode the transitions of a Turing machine. Completion attempts to generate a decision procedure for equality modulo equations, and as such can fail to terminate. Nevertheless, advances in automatic termination proving have resulted in powerful completion tools (e.g., Winkler and Middeldorp (2010); Wehrman et al. (2006)). Note that completion is a one-time phase for a given synthesis domain, and therefore can be employed offline, where it won't effect synthesis performance.

The Term Rewriting System

The TRS generated by completion is a set of *rewrite rules* of the form $l \to r$, which specify that if a (sub)program matches the *pattern* l, then it can be transformed using the pattern r. For instance, completion of the equations in our running example might result in a system that includes the rule

$$\texttt{swap}(\texttt{swap}(x)) \to x$$

That is, for any program containing the pattern $\texttt{swap}(\texttt{swap}(x))$, where x is a variable indicating any complete sub-term, we can rewrite it to x.

The above rule is a simple syntactic transformation (which we call

an *orientation*) of the corresponding equation. However, some equations result in rules that require more delicacy. Consider, for instance, commutativity of addition. The completion procedure will generate an *ordered* rewrite system to deal with such *unorientable* rules. For example, one rule that might be generated is

$$x + y \rightarrow_{>} y + x$$

which specifies that a program of the form $x + y$ can be rewritten into $y + x$ if and only if $x + y > y + x$, where $>$ is a well-founded *reduction ordering* on program terms. Often, the difficulty in completion is finding an appropriate reduction order, just as finding a *ranking function* is the key for proving program termination.

Given the TRS generated by the completion procedure, checking if a program p is in normal form is a simple process: if any of the rewrite rules in the TRS can be applied to p, then we know p is reducible and therefore not in normal form.

Synthesis with Equivalence Reduction

Let us now discuss how a synthesis procedure might utilize the TRS generated by completion to prune the search space. For the sake of illustration, suppose our synthesis technique constructs programs in a bottom-up fashion—combining small programs to generate larger programs—a strategy that is employed by a number of existing synthesis algorithms (Albarghouthi et al., 2013; Menon et al., 2013; Alur et al., 2013).

Consider the following simple program:

```
let f s count = (len (s ++ "012")) + count
```

where `s` is a string variable and `count` is an integer variable. The synthesizer constructs this program by applying integer addition to the two smaller expressions: `len (s ++ "012")` and `count`. To check if a program

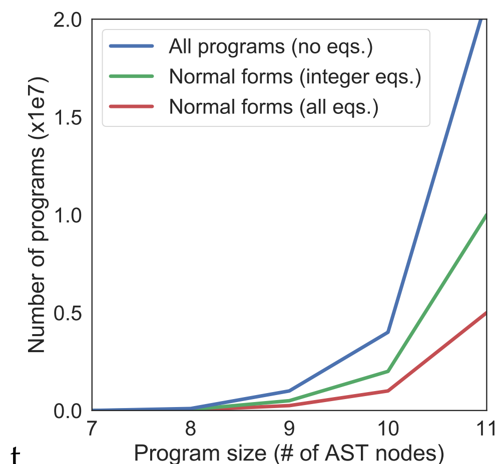


Figure 6.2: Number of terms vs. term size with varying levels of equivalence reduction

is in normal form, the synthesizer attempts to apply all the rules in the TRS generated by completion. If none of the rules apply, the program is *irreducible*, or in normal form. If any rule applies, we know the program is not in normal form, and we can completely discard this program from the search space. *But what if the final solution uses this discarded program as a sub-term?* By construction of the TRS, if a program p is not in normal form, then all programs where p is a sub-program are *also* not in normal form—intuitively, we can apply the same rewrite rule to p as a sub-program.

By ensuring that we only construct and maintain programs in normal form, we drastically prune the search space. Figure 6.2 shows the number of well-typed programs per program size in our running synthesis domain (augmented with two integer and two string variables). The blue line shows the number of programs per size of the abstract syntax tree. When we include the equations in Table 6.2 that only deal with integer components, the number of programs per size shrinks, as shown by the

green line. Incorporating the full set of equations over integer and string components shrinks the number of programs further, as shown by the red line. For instance, at 11 AST nodes, there are approximately 21 million syntactically distinct programs, but only about 20% of them are in normal form.

In all but the simplest of synthesis domains, the number of programs explodes as we increase the size. Utilizing the equations allows us to delay this explosion, and to peer deeper into the space of programs. In Section 6.7, we will experimentally demonstrate the utility of equations on practical synthesis applications.

6.3 Defining Synthesis Modulo Equations

Recall that $S = (\Sigma, \phi, h)$ is a *synthesis problem* (Definition 2.17): Σ induces a synthesis domain T_Σ of program terms, while ϕ and h encode a correctness constraint and a qualitative objective, respectively. We will extend this definition (and the corresponding solutions) with sets of equations.

Definition 6.1 (Equations over Terms)**.** *Let Σ be a signature, and X be a countable set of variables. A Σ-equation is a pair $(t_1, t_2) \in T_\Sigma(X) \times T_\Sigma(X)$, which denotes the universally quantified formula $\forall X.\, t_1 = t_2$.*

Example 6.2 (Matrix Operations)**.** *Suppose that the synthesis problem provides the signature $(\{m\}, \Sigma)$, where:*

$$\Sigma = \{t : m \to m, +_m : mm \to m, i : \epsilon \to m\}$$

where t computes the matrix transpose, $+_m$ is matrix addition, and i is an input

matrix. A possible set of Σ-equations is:

$$t(t(x)) = x \qquad (s_1)$$

$$t(x +_m y) = t(x) +_m t(y) \qquad (s_2)$$

where x and y are variables in X. Equation s_1 specifies that transposing a matrix twice returns the original matrix, and equation s_2 specifies that transposition distributes over matrix addition.

Using the set of equations $\{s_1, s_2\}$, we can infer that the following programs are semantically equivalent:

$$t(t(i) +_m t(i)) =_{s_2} t(t(i)) +_m t(t(i)) =_{s_1} i +_m i$$

Given a synthesis problem S, the set of Σ-equations $\mathcal{E}$ induce an equivalence relation on terms in T_Σ. We shall use $t_1 =_\mathcal{E} t_2$ to denote that two program terms are equivalent modulo $\mathcal{E}$ (formally defined in Section 6.4). Using $=_\mathcal{E}$, we can partition T_Σ into a union of disjoint equivalence classes:

$$T_\Sigma = \bigsqcup_{i \in I} P_i$$

where for all $t_1, t_2 \in T_\Sigma$, $t_1 =_\mathcal{E} t_2$ if and only if $\exists i \in I.\, t_1, t_2 \in P_i$. For each equivalence class P_i, we shall designate a single program $p_i \in P_i$, called the representative of P_i. A program term $t \in T_\Sigma$ is in *normal form* (denoted $\textsc{Norm}(t)$) if and only if it is a representative of some equivalence class P_i.

Definition 6.3 (Synthesis Modulo Equations Problem)**.** *Let $S = (\Sigma, \phi, h)$ be a synthesis problem (Definition 2.17), and let $\mathcal{E}$ be a set of Σ-equations (Definition 6.1). The pair $(S, \mathcal{E})$ is a* synthesis modulo equations problem*.*

Let $t \in T_\Sigma$. The term t is a solution *to $(S, \mathcal{E})$ if and only if*

$$t \in \operatorname{argmin}_{t \in T_\Sigma, t \models \phi, \textsc{Norm}(t)} h(t)$$

Definition 6.3 is much like Definition 2.17, except a solution only need be optimal with respect to other normal forms.

6.4 Term Rewriting and Completion

We will know extend our use of terms with the theory of *term rewriting systems* and discuss the use of *completion* to transform our equations $\mathcal{E}$ into a decision procedure that detects if a program is in normal form. For an in-depth view of term rewriting, we refer the reader to Baader and Nipkow (1998).

Rewrite Rules

A *rewrite system* R is a set of *rewrite rules* over some signature.

Definition 6.4 (Rewrite Rules and Systems). *Let Σ be a signature, and let X be a countable set of variables. A* rewrite rule *is a pair of the form:*

$$(l, r) \in T_\Sigma(X) \times T_\Sigma(X)$$

where $\mathcal{V}l \subseteq \mathcal{V}r$; the pair is written $l \to r$.

A rewrite rule $l \to r$ rewrites *t_1 into t_2 (written $t_1 \xrightarrow{l \to r} t_2$) if and only if there exists (i) a context C, a variable assignment $\nu : X \to T_\Sigma$, and a term $s \in T_\Sigma$ such that*

$$t_1 = C[s] \quad \textit{and} \quad s = \nu^*(l) \quad \textit{and} \quad t_2 = C[\nu^*(r)]$$

A set of rewrite rules R is called a term rewriting system, *and induces a relation $\to_R$ over $T_\Sigma \times T_\Sigma$ as follows:*

$$t_1 \to_R t_2 \Leftrightarrow \exists (l \to r) \in R.\, t_1 \xrightarrow{l \to r} t_2$$

We illustrate rewrite rules with an example.

Example 6.5. *Consider the rewrite rule* $f(x, x) \to g(x)$, *and consider the program term* $t = f(f(a, a), b)$, *where* a *and* b *are two constant function symbols. We can apply the rewrite rule to rewrite* t *into* $t' = f(g(a), b)$ *by replacing the subprogram* $f(a, a)$ *with* $g(a)$.

We will use $\to_R^*$ to denote the reflexive transitive closure of the rewrite relation. The symmetric closure of $\to_R^*$, denoted $\leftrightarrow_R^*$, forms an equivalence relation. When clear from context, we shall drop the subscript R.

Normal Forms

For a given TRS R, a program term t is R-*irreducible* if and only if there is no program term t' such that $t \to_R t'$. For a fixed t, the set of R-irreducible programs reachable from p via $\to_R$ is its set of *normal forms*. We write $N_R(t) = \{t' \mid t \to_R^* t',\ t' \text{ R-irreducible}\}$ for the normal forms of t.

A TRS R is *normalizing* if and only if, for every program term t, $|N_R(t)| \geqslant 1$. R is *terminating* if and only if the relation $\to_R$ is *well-founded*; that is, for every term t, there exists $n \in \mathbb{N}$ such that there is no $t' \in T_\Sigma$ where $t \to_R^n t'$ (i.e., there is no t' reachable from t through n rewrites).

To synthesize exclusively normal forms, one might rewrite the synthesis domain so that all terms are normal by construction. This can be done by (*i*) interpreting Σ and R as *regular tree languages*, (*ii*) computing their intersection, which is again regular, and (*iii*) converting the intersection into a Σ'. However, expressing R as a regular tree grammar requires the rules to be unordered (see below) and left-linear (Otto, 1999); these conditions are too strong to be a general solution.

Completion Procedures

Equations take a similar form to rewrite rules, and as such it is often common to treat an equation (t_1, t_2) as a *pair* of rewrite rules, $t_1 \to t_2$ and $t_2 \to t_1$ (assuming $Vt_1 = Vt_2$). This naíve construction, however, produces a TRS that is not terminating, and so cannot easily be used for computing determining unique normal forms. To generate a terminating TRS equivalent to a set of equations $\mathcal{E}$, we turn to *completion procedures*.

For our purposes, we only need a declarative view of completion procedures. A completion procedure provides a term rewriting system R_c such that $t \leftrightarrow^*_R t' \Leftrightarrow t =_{\mathcal{E}} t'$ and, for any program term t, applying rules in R_c will always lead to a unique normal form in finitely many rewrites, regardless of the order in which rewrites are applied. Formally, R_c is *terminating and confluent*.

Completion is generally undecidable. Knuth and Bendix (1983) are responsible for the first completion procedure; it repeatedly attempts to *orient* equations—turn them into rewrite rules by making one side l and one side r—through syntactic transformations. As such, Knuth-Bendix completion can fail to produce a result, even if it terminates, as not all equations are orientable. Bachmair et al. (1989) neatly side-step this weakness by presenting a completion procedure that cannot terminate and still fail, called *unfailing completion*. In order to handle unorientable rules, unfailing completion introduces *ordered rules*: let $>$ be a *reduction order*—a well-founded order that ensures termination of the rewrite system—and let $r : u \to_> v$ be an ordered rule. Then $t_1 \to t_2$ by rule r if and only if $t_1 \xrightarrow{u \to v} t_2$ and $t_1 > t_2$.

Recall our matrix domain from Example 6.2, and suppose we have the equation $x +_m y = y +_m x$. Knuth-Bendix completion will fail to orient this rule, but unfailing completion, when provided with a suitable reduction order $>$, would generate the ordered rule $x +_m y \to_> y +_m x$. Modern completion tools are able to simultaneously complete a set of rules and

derive an appropriate reduction order (Winkler and Middeldorp, 2010; Wehrman et al., 2006).

Reduction Orders

The *Knuth-Bendix order* (KBO) is a standard family of reduction orders that we will use in our implementation and evaluation. The formal definition of KBO is not important for our exposition, and we thus relegate it to Appendix D.1. We will denote a KBO as $>_{\text{KBO}}$, and note that naïvely computing KBO following its standard definition is polynomial in the size of the compared terms. We discuss our use of (and the importance of) a linear-time implementation in Section 6.7.

Another prominent class of reduction orders are the *path orderings*, such as the *lexicographic path order* (LPO) and *multiset path order* (MPO). A disadvantage of LPO and MPO is that naive implementations are exponential in the size of the terms, and even optimized implementations are only polynomial Löchner (2006). Furthermore, while there exist equations that path orders can orient that KBO fails to, they sometimes orient equations in ways that massively expand the size. The weights of KBO provide a check to this behavior, limiting how large (with respect to size) the right-hand side of a rule can be. This property makes KBO a more natural fit for synthesis tasks.

6.5 Synthesis Modulo Equations

Bottom-up synthesis techniques explore the space of programs in a bottom-up, dynamic-programming fashion, building larger programs from smaller ones. Examples include Escher (Albarghouthi et al., 2013), the enumerative solver of SyGuS (Alur et al., 2013), and the probabilistic search of Menon et al. (Menon et al., 2013).

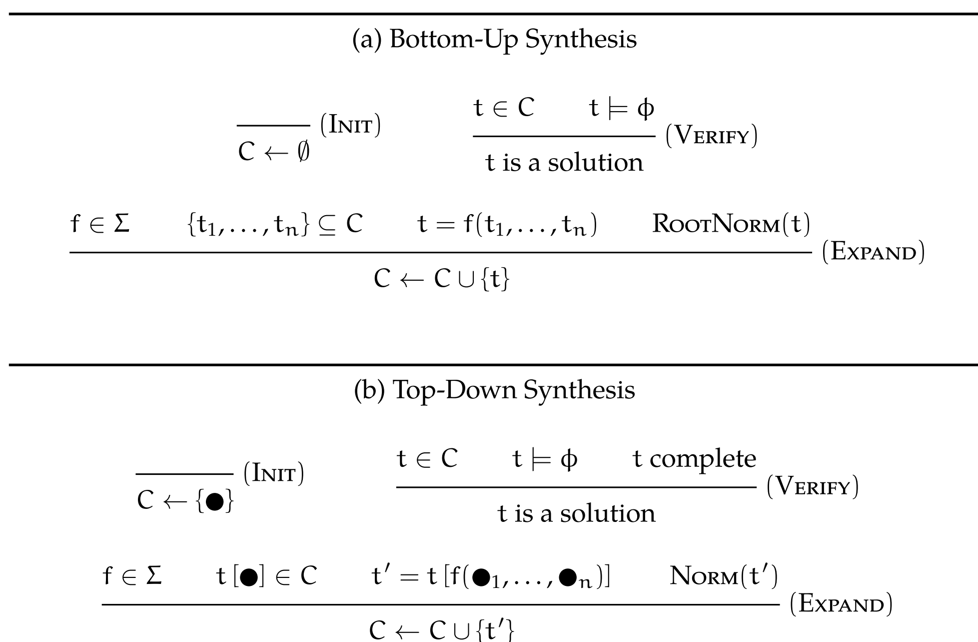


Figure 6.3: Algorithms for synthesis with equivalence reduction

Top-down synthesis techniques explore the space of programs in a top-down fashion, effectively, by unrolling the signature parameterizing the search space. A number of recent synthesis algorithms, particularly for functional programs, employ this methodology, e.g, the approach in Chapter 3, Myth (Osera and Zdancewic, 2015), Myth2 (Frankle et al., 2016), λ^2 (Feser et al., 2015), and SynQuid (Polikarpova et al., 2016).

We now present abstract algorithms for these techniques and show how to augment them with equivalence reduction.

Bottom-Up Synthesis Modulo Equations

Figure 6.3(a) shows a bottom-up synthesis algorithm as a set of guarded inference rules that can be applied non-deterministically. The only state maintained is a set C of explored programs, which is initialized to the empty set using the rule (Init). The algorithm terminates whenever the

rule (Verify) applies, in which case a solution to $(\mathcal{S}, \mathcal{E})$ has been found, modulo optimization of the qualitative objective h.

The rule (Expand) creates a new program term t by applying an n-ary function f to n program terms from the set C. Observe, however, that (Expand) maintains the invariant that all programs in C are in normal form. This invariant can be used to simplify checking Norm(t) during the (Expand) step. In synthesizing $t = f(t_1, \ldots, t_n)$, we already know that terms $t_1, \ldots, t_n$ are normal. Therefore, if t is not normal, it must be due to a rule applying at the *root*. Checking this property, called *root-normality*, simplifies rule application—instead of examining all sub-terms of t to see if the rule $l \to r$ applies, it suffices to check whether there exists an assignment $\nu : X \to T_\Sigma$ such that $\nu^*(t) = \nu^*(l)$.

Top-Down Synthesis Modulo Equations

Figure 6.3(b) shows the top-down synthesis algorithm, a simplified version of the algorithm in Chapter 3. The algorithm maintains a set C of incomplete program terms. C is initialized with the term $\bullet$, using (Init). The rule (Expand) picks an incomplete term $t \in C$ and substitutes one of its wildcards with a new program term. The algorithm terminates when a complete program in C satisfies the specification ϕ, as per rule (Verify).

The rule (Expand) checks whether t is in normal form before adding it to C. However, note that the algorithm maintains incomplete programs in C, and even if an incomplete term is normal, not all complete terms derivable from it through (Expand) need be normal. Therefore, the algorithm may end up exploring sub-trees of the search space that are redundant. Deciding if an incomplete program has complete instances in normal form is known as checking R-*ground reducibility*, which is decidable in exponential time (Comon and Jacquemard, 1997). Our formulation avoids exponential checks at the cost of exploring redundant sub-trees.

6.6 Properties of Equivalence Reduction

Synthesis modulo equations enjoys several nice properties. We begin by presenting a notion of consistency between our correctness constraint ϕ and equations:

Definition 6.6. *A correctness constraint* ϕ *is* $\mathcal{E}$-consistent *if and only if* $t_1 =_{\mathcal{E}} t_2 \Rightarrow (t_1 \models \phi \iff t_2 \models \phi)$.

This definition lifts naturally to synthesis modulo equations problems:

Definition 6.7. *A synthesis modulo equations problem* $(S, \mathcal{E})$ *is* consistent *if and only if* ϕ *is* $\mathcal{E}$*-consistent.*

In practice, it is straightforward to ensure a synthesis modulo equations problem $(S, \mathcal{E})$ is consistent. A common approach is ensuring that $\mathcal{E}$ is *semantics-preserving*, and that ϕ cares only about the *semantics* of candidate programs—e.g., not their runtime or memory overhead. When we have such guarantees, we are free to prune unnecessary programs safe in the knowledge that another version still exists in the search space. This intuition leads very naturally to a guarantee of soundness:

Theorem 6.8 (Soundness)**.** *Let* $(S, \mathcal{E})$ *be a consistent synthesis modulo equations problem. Then, if bottom-up or top-down synthesis returns* t, t *is a solution to* $(S, \mathcal{E})$.

The proof—which is similar to the proofs of Theorem 3.5—proceeds through structural induction on the inference rules defining bottom-up and top-down synthesis, and the observation that when we prune we always preserve at least one program term t such that $t \models \phi$. Furthermore, consistent synthesis problems satisfy a weak notion of completeness:

Theorem 6.9 (Relative Completeness)**.** *Given a consistent synthesis modulo equations problem* $(S, \mathcal{E})$*, if* t *is a solution to* $(S, \mathcal{E})$ *then bottom-up synthesis*

(equivalently, top-down synthesis) will return a solution to $(S, \mathcal{E})$ *in finitely-many steps.*

Again, the proof mirrors that of Theorem 3.6.

The above theorems, when combined with the ease of guaranteeing $(S, \mathcal{E})$ is consistent, are the central strength of synthesis modulo equations. When provided with a synthesis framework built around a fair implementation of bottom-up or top-down synthesis, we are free to guide the search by providing equations consistent with the specification, reducing the search space dramatically and—as we will see in Section 6.7—improving the performance noticeably.

Incomplete Completion

A useful property of Theorem 6.8 and Theorem 6.9 is that—when ϕ is dependent only on semantics—we only require our equations $\mathcal{E}$ to be *consistent* with the semantics of candidate programs, not fully explain them. We can use this fact to preserve soundness and completeness in the face of undecidability of completion procedures.

If, in the process of constructing a procedure for $\textsc{Norm}(\cdot)$, we are unable to complete the equations $\mathcal{E}$, we will often be able to salvage an *under-approximation* R of a normalizing TRS for $\mathcal{E}$. The following definitions characterize this behavior:

Definition 6.10. *A* TRS R *under-approximates equations* $\mathcal{E}$ *if and only if* $t_1 \leftrightarrow_R^* t_2 \Rightarrow t_1 =_{\mathcal{E}} t_2$ *for all program terms* t_1 *and* t_2*.*

A consistency constraint ϕ *is* R-consistent *if and only if* $t_1 \leftrightarrow_R^* t_2 \Rightarrow (t_1 \models \psi \iff t_2 \models \phi)$ *for all program terms* t_1 *and* t_2*.*

Under-approximating rewrite systems are not useless. The following theorem shows that by under-approximating, we maintain consistency.

Theorem 6.11. *Let* TRS $\mathcal{R}$ *under-approximate* $\mathcal{E}$*, and* ϕ *be* $\mathcal{E}$*-consistent. Then* ϕ *is* $\mathcal{R}$*-consistent.*

The proof follows immediately from the definitions. However, Theorem 6.11 allows us to use an under-approximation of $\mathcal{E}$ to construct $\textsc{Norm}(\cdot)$ and still maintain soundness and relative completeness of our search while gaining the performance benefits of equivalence reduction.

Salvaging an under-approximation from a failed completion run is straightforward. The completion procedures mentioned in this paper (Wehrman et al., 2006; Knuth and Bendix, 1983; Winkler and Middeldorp, 2010) all grow a TRS $\mathcal{R}$ by adding rules that are consistent with the provided set of equations $\mathcal{E}$ until $\mathcal{R}$ is able to fully describe a set of normal programs for $\mathcal{E}$. Therefore, stopping the completion procedure *at any time*, and for any reason, will yield an under-approximation and preserve correctness of the synthesis procedure.

6.7 Implementation and Evaluation

We implemented our technique in an existing efficient synthesis tool, written in OCaml, that employs bottom-up and top-down search strategies. Our tool accepts a signature Σ, defined as typed OCaml functions, along with a set of equations $\mathcal{E}$ over the provided OCaml functions. For the correctness constraint ϕ and the qualitative objective h, we utilize input-output examples (see Section 6.7 below) and a simple size function.

The implementation of bottom-up and top-down synthesis augment the abstract algorithms in Section 6.5 with a deterministic search strategy that utilizes types, as in Chapter 3 and Chapter 4, so that only well-typed programs are enumerated.

Completion and Reduction Orders

Completions of equations were found using the omkbTT tool (Winkler and Middeldorp, 2010), which employs termination provers for generating a reduction order. All completions used the KBO reduction order.

During synthesis, the reduction order can be a performance bottleneck, as we need to compute it for every candidate program. If we were to implement KBO directly from its formal definition, evaluating $s >_{KBO} t$ would be quadratic in $|s| + |t|$. However, program transformation techniques have given us an algorithm linear in the sizes of the terms (Löchner, 2006). In our tool, we implement Löchner's linear-time KBO algorithm. The performance impacts of the reduction order are discussed in Section 6.7.

Data Structures for Normalization

Every time a candidate program is considered, we check if it is in normal form using $\textsc{Norm}(\cdot)$ (recall algorithms in Figure 6.3). More precisely, given a candidate program term t, $\textsc{Norm}(t)$ attempts to find an assignment $\nu : X \to T_\Sigma$ and a rule $l \to r$ such that $t = C[\nu^*(l)]$. This is a *generalization* problem, which has been studied for years in the field of automated theorem proving. A naïve implementation of $\textsc{Norm}(t)$ might keep a list of rules in the TRS, and match candidate programs against one rule at a time. Instead, we borrow from the existing literature and use *perfect discrimination trees* (McCune, 1992) to represent our list of rules. Perfect discrimination trees are used in the Waldmeister theorem prover (Hillenbrand et al., 1997) to great effect; *the tree representation lets us match multiple rules at once, and ignore rules that are inapplicable*.

A perfect discrimination tree can be thought of as a *trie*. Figure 6.4 illustrates the construction for a set of unordered rules—ordered rules can be added analogously. First, rules are rewritten using De Bruijn-like indices (De Bruijn, 1972). Second, the left-hand side of every rule is

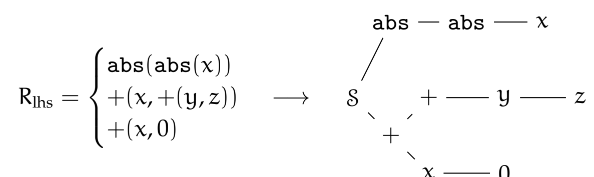
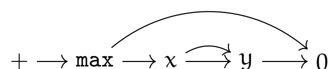


Figure 6.4: Building a perfect discrimination tree from left-hand side of rules

converted into a string through a pre-order traversal. Finally, all string representations are inserted into the trie.

To match a candidate program term t against the trie, we first convert t to a *flat-term*, which is a linked-list representation of t in pre-order, with forward pointers to jump over sub-terms. For example, the term $\mathtt{max}(x, y) + 0$ is converted to:



Now, matching the program against the trie is done using a simple backtracking algorithm, which returns a substitution (if one exists) that converts the left-hand side of a rule in our set to the query program. See McCune (1992) for details.

Using perfect discrimination trees in our normalization procedure has several immediate benefits, the most important of which is that unused rules do not impact the performance, as their paths are never followed. Section 6.7 will evaluate the performance overhead of normalization.

Synthesis Domain and Benchmarks

A primary inspiration for this work came from applying synthesis to the domain of large-scale, data-parallel programming, as in Chapter 3. Here, we will focus on synthesizing *reducers*, and so will be applying the verification technique from Section 3.4.

```
let add-c q1 q2 =
    let real = (fst q1) + (fst q2) in
    let imaginary = (snd q1) + (snd q2) in
    pair real imaginary
```

Figure 6.5: Addition of two complex numbers of the form $a + bi$, where a and b are represented as a `pair`

Our synthesis domain comprises four primary sets of components, each consisting of 10+ components, that focus on different types. These types—integers, tuples, strings, and lists—are standard, and appear as the subject of many synthesis works. The full list of components appears in Appendix D.2.

We manually gathered a set of 50 equations for our synthesis domain. Alternatively, this process can be automated using tools like QuickSpec (Claessen et al., 2010) and Bach (Smith et al., 2017). Each class of components has between 3 (lists) and 21 (integers) equations, with a few equations correlating functions over multiple domains (e.g., strings and integers interacting through `length`). Completions of the equations are a mix of ordered and unordered rules describing the interaction of the components. Some equations are described below, but the full list of equations is in Appendix D.2.

1. **Strings**: In addition to the equations relating `upper`, `swap`, and `lower` (as defined in Section 6.1), we include equations encoding, e.g., idempotence of `trim`, and the fact that many string operations distribute over concatenation. For instance, we have the equation

$$\forall x, y.\, \texttt{len}(x) + \texttt{len}(y) = \texttt{len}(x \mathbin{+\!\!+} y)$$

2. **Lists**: We provide equations specifying that operations distribute over list concatenation, as in $\forall x, y.\texttt{sum}(x) + \texttt{sum}(y) = \texttt{sum}(\texttt{cat}(x, y))$. In addition,

we relate constructors/destructors, as in $\forall x, y.\texttt{head}(\texttt{cons}(x, y)) = x$.

Benchmarks Our benchmarks were selected to model common reducers over our domain, and typically require solutions with 10–12 AST nodes—large enough to be a challenge for state-of-the-art synthesizers, as we see later. A few examples are given below, and the full list is in Appendix D.2.

1. **Tuples and integers**: The tuple benchmarks expose several different uses for pairs in reducers—as an encoding for rational numbers (such as in `mult-q`), for complex numbers (in `add-c`), and for points on the plane (as in `distances`). We also treat pairs as intervals over integers (e.g., `intervals` synthesizes *join* in the lattice of intervals (Cousot and Cousot, 1977)). Figure 6.5 shows the synthesized program for one of those benchmarks.

2. **Lists and integers**: Lists are also an interesting target for aggregation, e.g., if we are aggregating values from different scientific experiments, where each item is a list of readings from one sensor. List benchmarks compute a value from two lists and emit the result as a singleton list. For example, `ls-sum-abs` computes absolute value of the sums of two lists, and then adds the two, returning the value as a singleton list.

Like many synthesis tools, we use input–output examples to characterize the desired solution. Examples are used to ensure that the solution (*i*) matches user expectations and (*ii*) forms a CSG.

Experimental Evaluation

Our experiments investigate the following questions:

RQ1 Does equivalence reduction increase the efficiency of synthesis algorithms on the domain described above?

RQ2 What is the overhead of equivalence reduction?

RQ3 How does the performance change with different numbers of equations?

RQ4 Are the data structures used in theorem provers a good fit for synthesis?

To address these questions, we developed a set of 30 synthesis benchmarks. Each benchmark consists of: (*i*) a specification, in the form of input–output examples (typically no more than 4 examples are sufficient to fully specify the solution); (*ii*) a set of components from the appropriate domain; (*iii*) a set of ordered and unordered rewrite rules generated from equations over the provided components.

For each synthesis algorithm, bottom-up (BU) and top-down (TD), we use three different levels of equivalence reduction:

1. BU and TD: equivalence reduction disabled.
2. $BU_{\mathtt{n}}$ and $TD_{\mathtt{n}}$: equivalence reduction enabled.
3. $BU_{\tilde{\mathtt{n}}}$ and $TD_{\tilde{\mathtt{n}}}$: equivalence reduction without ordered rules. By dropping ordered rules from the generated TRS, we get more normal forms (less pruning).

See Table 6.3 for the full results. For each experiment, we measure total time taken in seconds. Grey boxes indicate the best-in-category strategy for each benchmark—e.g., the winner of the `sub-c` benchmark is $BU_{\mathtt{n}}$ in the bottom-up category, and $TD_{\tilde{\mathtt{n}}}$ in top-down. Values reported are the average across 10 runs.

RQ1: Effects of equivalence reduction on performance In 2 out of the 3 benchmarks where BU and TD do not terminate, adding equivalence reduction allows the synthesizer to find a solution in the allotted time. For bottom-up, in all benchmarks where BU terminates in under 1s, both $BU_{\mathtt{n}}$ and $BU_{\tilde{\mathtt{n}}}$ outperform the naive BU, often quite dramatically: in `sum-to-second`, BU takes over 60s, while $BU_{\mathtt{n}}$ and $BU_{\tilde{\mathtt{n}}}$ finish in under 2s.

Benchmark	Bottom-up variations BU	$BU_{\tilde{n}}$	BU_n	Top-down variations TD	$TD_{\tilde{n}}$	TD_n	Other tools λ^2	SynQuid
Integers								
`add`	0.00	0.00	0.00	0.00	0.00	0.00	0.01	0.18
`max`	0.01	0.01	0.01	0.03	0.05	0.02	0.4	0.6
`min`	0.01	0.01	0.01	0.05	0.02	0.04	0.01	0.62
Tuples & integers								
`add-4`	0.05	0.05	0.11	0.12	0.14	1.29	5.35	8.86
`mult-q`	7.38	0.48	0.44	15.63	1.95	3.03	✗	✗
`div-q`	7.38	0.48	0.44	14.22	2.12	3.82	✗	✗
`add-c`	7.39	0.48	0.44	13.51	3.68	8.11	✗	✗
`sub-c`	7.35	0.48	0.43	31.63	4.51	9.28	✗	✗
`add-q-long`	✗	44.65	49.51	✗	57.43	95.55	✗	✗
`max-pair`	32.79	4.02	4.92	40.25	21.53	56.21	✗	✗
`intervals`	32.76	4.00	4.92	74.77	25.25	21.80	✗	✗
`min-pair`	32.72	4.03	4.95	81.88	19.55	67.49	✗	✗
`sum-to-first`	52.74	1.18	0.62	110.35	5.47	15.06	✗	✗
`sum-to-second`	68.93	1.51	0.70	107.88	5.27	11.22	✗	✗
`add-and-mult`	7.39	0.48	0.45	26.12	1.51	9.34	✗	✗
`distances`	✗	7.36	5.58	✗	50.31	100.66	✗	✗
Strings & integers								
`str-len`	1.40	0.42	0.52	4.47	1.58	2.06	-	-
`str-trim-len`	26.29	6.79	7.14	219.50	52.21	218.61	-	-
`str-upper-len`	5.70	1.78	1.81	26.93	5.64	8.01	-	-
`str-lower-len`	3.86	1.23	1.26	22.48	6.93	4.63	-	-
`str-add`	0.05	0.02	0.03	0.26	0.08	0.07	-	-
`str-mult`	0.05	0.02	0.03	0.17	0.05	0.06	-	-
`str-max`	1.79	0.45	0.54	8.10	1.32	2.99	-	-
`str-split`	✗	✗	✗	✗	✗	✗	-	-
Lists & integers								
`ls-sum`	0.00	0.02	0.03	0.01	0.02	0.11	0.01	10.43
`ls-sum2`	147.91	88.33	107.02	229.66	✗	254.87	✗	✗
`ls-sum-abs`	0.08	0.07	0.10	0.22	0.19	0.37	63.00	✗
`ls-min`	0.00	0.02	0.03	0.01	0.02	0.10	0.01	-
`ls-max`	0.00	0.02	0.03	0.01	0.02	0.10	0.01	-
`ls-stutter`	27.68	5.12	8.01	50.42	15.90	93.84	✗	-

Table 6.3: Experimental equivalence reduction results; we impose a CPU timeout of 300 seconds (✗ denotes a timeout) and a memory limit of 10GBs per benchmark

For top-down, $TD_{\tilde{n}}$ outperforms TD in nearly all benchmarks that take TD more than 1 second (the exception being `ls-sum2`). With ordered rules, the exceptions are more numerous. The most egregious is `ls-stutter`, going from 50s with TD to 94s with TD_n. There is still potential for large performance gains: in `sum-to-second`, we decrease the time from 108s in

TD to under 12s for TD_{n} and under 6s for $TD_{\tilde{n}}$.

Equivalence reduction appears to drastically improve the performance of bottom-up and top-down synthesis. In general, the unordered rules outperform the full ordered rules. In the bottom-up case, this performance gap is smaller than 5s: while the ordered rules are more costly to check, bottom-up synthesis only requires that we check them at the root of a program. In top-down, we must check rule application at all sub-programs. This magnifies the cost of the ordered rules and leads to significant performance differences between TD_{n} and $TD_{\tilde{n}}$.

RQ2-a: Overhead of equivalence reduction Figure 6.6 provides a different look at the benchmarks in Table 6.3: for each benchmark where BU and TD do not terminate in less that 1 second, we compute (*i*) the *overhead*, the percentage of time spent in the normalization procedure $\textsc{Norm}(\cdot)$; and (*ii*) the *reduction*, the percentage of programs visited compared to the un-normalized equivalent, BU or TD. The results are shown as density plots.

Figure 6.6a and Figure 6.6c show the performance characteristics of $BU_{\tilde{n}}$ and $TD_{\tilde{n}}$, respectively. Both have consistent overhead—40% for $BU_{\tilde{n}}$ and 25% for $TD_{\tilde{n}}$—although $TD_{\tilde{n}}$ has a more reliable reduction of over 85%, while $BU_{\tilde{n}}$ ranges from 60% to 90% reduction. Both strategies boast large reductions in the number of candidate programs visited for reasonable overhead, although $TD_{\tilde{n}}$ is the clear winner—$BU_{\tilde{n}}$ dominates $TD_{\tilde{n}}$ in Table 6.3, suggesting that normalization isn't enough to fully close the gap between BU and TD. In Figure 6.6b and Figure 6.6d, we see the performance characteristics of BU_{n} and TD_{n}, respectively. Compared to Figure 6.6a and Figure 6.6c, we see a higher overhead with less consistent normalization. Both figures have secondary clusters of benchmarks outside the region of highest density: these contain the benchmarks from the strings and integers domain.

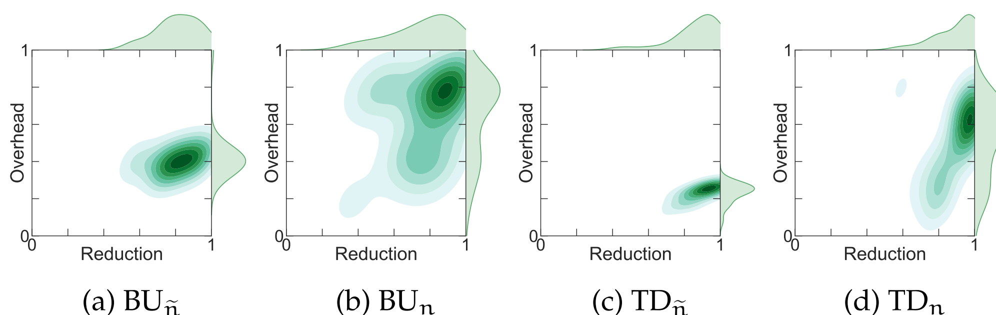
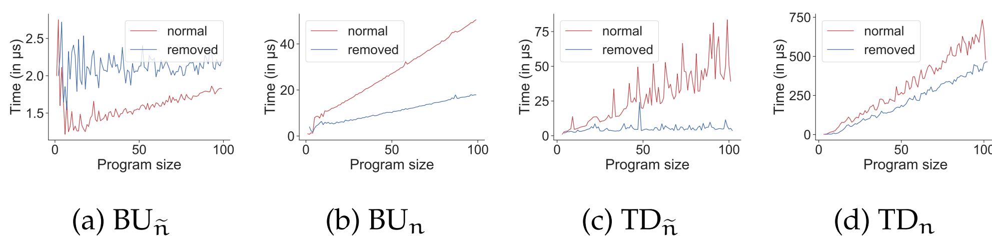


Figure 6.6: Equivalence reduction overhead—benchmarks are converted into (overhead, reduction) pairs and plotted using kernel density estimation (KDE), with marginal distributions projected on to the side.

This view of the data supports the conclusion of Table 6.3 that unordered rules outperform ordered rules. While our implementation of KBO is optimized, evaluating the reduction order is still a bottleneck. Our implementation verifies candidate solutions quickly, but the benefits of high reduction outweigh the large overhead as verification time increases. For instance, when providing more input-output examples, the verification time increases but not the overhead. In the `ls-stutter` benchmark, $BU_{\tilde{n}}$ visits 1,288,565 programs with an average overhead of 1.07 seconds, while BU_{n} visits 792,662 programs with an average overhead of 5.6 seconds. Increasing the verification cost per program by only 0.0001 seconds will raise $BU_{\tilde{n}}$'s time by 129s, while BU_{n}'s time is only raised by 80s—easily a large enough gap to out-scale the overhead. Indeed, when we instrument our tool with a synthetic delay, this behavior is visible.

RQ2-b: Normalization overhead Experience holds that normalization procedures don't scale as candidate programs become large. To explore how this behavior might impact the effectiveness of equivalence reduction, we instrumented our tool to ignore solutions and explore the space of programs depth-first, during which we record the average overhead of $\textsc{Norm}(\cdot)$ at all program sizes. Figure 6.7 presents the data for the



(a) $\mathrm{BU}_{\tilde{n}}$ (b) BU_{n} (c) $\mathrm{TD}_{\tilde{n}}$ (d) TD_{n}

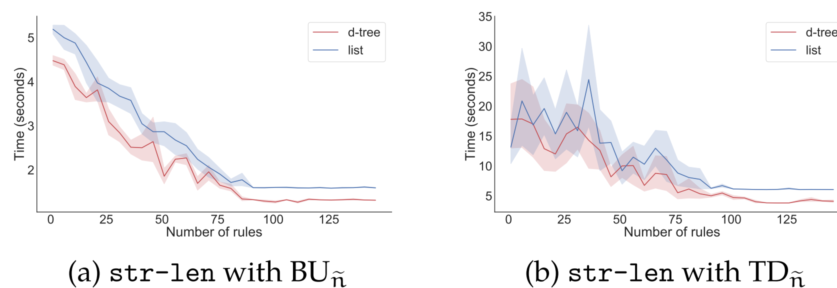
Figure 6.7: Average performance of $\textsc{Norm}(\cdot)$ on `sum-to-first`; *normal* graph represents executions of $\textsc{Norm}(\cdot)$ that return `true`; *removed* represents executions that return `false`. Time is in microseconds—note the difference in scale between graphs.

`sum-to-first` benchmark, although the figures are representative of the other benchmarks.

Unsurprisingly, $\textsc{Norm}(\cdot)$ scales linearly with program size. This linear growth appears quite sustainable. Solutions with 100 AST nodes are beyond modern-day synthesis tools, and a 3x slowdown compared to programs of size 40 is manageable.

When we compare the performance of $\mathrm{BU}_{\tilde{n}}$ in Figure 6.7a to that of BU_{n} in Figure 6.7b, we observe an order of magnitude loss in performance. This holds as well for $\mathrm{TD}_{\tilde{n}}$ and TD_{n} in Figure 6.7c and 6.7d, respectively. Checking KBO is clearly expensive, and so the observed performance in Table 6.3 of BU_{n} and TD_{n} indicate a large amount of search-space reduction occurring.

RQ3 & RQ4: Impact of rules & perfect discrimination trees To determine how the number of rules impacts our tool's performance, we completed our entire set of 50 equations to produce 83 unordered rules that we randomly sample subsets from (the results from ordered results are similar). To test the effectiveness of perfect discrimination trees, we compare performance against a naïve algorithm that maintains a list of rules it checks against one by one on a representative benchmark: `str-len`.



(a) `str-len` with $\mathrm{BU}_{\tilde{n}}$ (b) `str-len` with $\mathrm{TD}_{\tilde{n}}$

Figure 6.8: Performance per number of rules sampled for `d-tree` and `list` over 2 benchmarks; the line is the average of 10 samples per x-value, and the lighter band is a 95% confidence interval

Not all rules apply to the components used—only 47 out of 83 describe components used for `str-len`. We plot the time taken for synthesis per number of randomly sampled rules, from 0 rules to 150 rules (to clearly show optimal performance). Results are presented in Figure 6.8.

We see, for both benchmarks, nearly continuously decreasing graphs; the only exceptions are with low numbers of rules sampled, where it is likely we have mostly unusable rules. The performance levels off at 83 rules, when we are guaranteed to sample all applicable rules. These results are promising: completion is undecidable, and so it is impossible to predict the rules that will be included from a given set of equations. However, the results in Figure 6.8 indicate that—on average—the more rules we provide the better the algorithm's performance, even when the rules might not be relevant. Furthermore, we see immediately and clearly that perfect discrimination trees outperform our list-based implementation. Performance differences are magnified in the $\mathrm{TD}_{\tilde{n}}$ benchmarks, where checking normality includes checks on every sub-term. On the rest of the benchmarks, the naive implementation results in an average of an 11% increase in time for $\mathrm{BU}_{\tilde{n}}$ and a 144% increase for $\mathrm{TD}_{\tilde{n}}$, which strongly indicates that perfect discrimination trees are an important implementation

choice.

Gauging benchmark difficulty We considered related tools as a gauge of benchmark difficulty and a baseline for evaluation. The most similar tool—λ^2 (Feser et al., 2015)—is top-down, type-directed, uses input-output examples, and searches for programs from smallest to largest. SynQuid (Polikarpova et al., 2016) synthesizes Haskell programs from refinement types, using SMT-driven type-directed synthesis. When able, we encoded specifications of our benchmarks as refinement types.

As seen in Table 6.3, λ^2 is either not applicable (strings are not supported, and so were ignored) or unable to solve most benchmarks. SynQuid exhibits similar behavior and performance. We stress that these results are meant as a indication of the difficulty of the benchmarks, and not a head-to-head comparison between our algorithms and those of λ^2 and SynQuid.

Threats to validity We identify two primary threats to the validity of our evaluation. First, we base our evaluation on a single tool in order to evaluate various algorithms and data structures. However, since our bottom-up and top-down strategies are (*i*) instances of standard synthesis techniques and (*ii*) comparable to existing implementations (as seen in Table 6.3), we believe our results can be beneficial to tools like Myth, SynQuid, and λ^2, modulo technical details.

Second, the domains considered in our evaluation—integers, lists, etc.—operate over well-behaved algebraic structures. These domains form the core search space of many modern synthesis tools, but one could imagine domains that do not induce many equational specifications, e.g., GUI manipulation and stateful domains.

6.8 Related Work

Program synthesis We are not the first to use normal forms for pruning in synthesis. In type-directed synthesis, Osera and Zdancewic (2015) and Frankle et al. (2016) restrict the space by only traversing programs in β-*normal form*. Equivalence reduction can be used to augment such techniques with further pruning, by exploiting the semantics of the abstract data types defined. Feser et al. (2015) mention that their enumeration uses a fixed set of standard rewrites, e.g., $x + 0 \rightarrow x$, to avoid generating redundant expressions. In contrast, our work presents a general methodology for incorporating equational systems into the search by exploiting completion algorithms.

Superoptimization techniques that search for fast programs (Schkufza et al., 2013; Phothilimthana et al., 2016) may not be able to directly benefit from equivalence reduction, as it may impose inefficient normal forms. It would be interesting to incorporate a cost model into completion and coerce it into producing minimal-cost normal forms.

In SyGuS (Alur et al., 2013; Udupa et al., 2013), the synthesizer generates a program encodable in a decidable first-order theory and equivalent to some logical specification. A number of solvers in this category employ a *counter-example-guided synthesis loop* (CEGIS) (Solar-Lezama et al., 2006): they prune the search space using a set of input-output examples, which impose a coarse over-approximation of the true equivalence relation on programs. In the CEGIS setting, equivalence reduction can be beneficial when, for instance, (*i*) evaluating a program to check if it satisfies the examples is expensive, e.g., if one has to compile the program, simulate it, evaluate a large number of examples; or (*ii*) the verification procedure does not produce counterexamples, e.g., if we are synthesizing separation logic invariants.

A number of works sample programs from a probabilistic grammar that imposes a probability distribution on programs (Menon et al., 2013;

Liang et al., 2010; Dechter et al., 2013). It would be interesting to investigate incorporating equivalence reduction in that context, for instance, by *truncating* the distribution so as to only sample irreducible programs.

Recently, Wang et al. (2017b) introduced SYNGAR, where abstract transition relations are provided for each component of a synthesis domain. The synthesis algorithm over-approximates equivalence classes by treating two programs equivalent if they are equivalent in the abstract semantics. The abstraction is refined when incorrect programs are found.

Term-rewriting systems A number of classic works (Reddy, 1989; Dershowitz, 1985) use completion procedures to transform an equational specification into a program—a terminating rewrite system. Our setting is different: we use completion in order to prune the search space in modern inductive synthesis tools.

Kurihara and Kondo's multi-completion (Kurihara and Kondo, 1999) sidesteps the issue of picking a reduction order by allowing completion procedures to consider a class of reduction orders simultaneously. Klein and Hirokawa's maximal completion algorithm (Klein and Hirokawa, 2011) takes advantage of SMT encodings of reduction orders (such as Zankl et al.'s KBO encoding (Zankl et al., 2009)) to reduce completion to a series of MaxSMT problems in which the parameters of the reduction order are left free. Completion tools like omkbTT (Winkler and Middeldorp, 2010) and Slothrop (Wehrman et al., 2006) rely on external termination provers (Alarcón et al., 2010; Korp et al., 2009).

7 CONCLUSION

This dissertation is focused on the application of program synthesis to improve the availability of data in the face of burdens-of-knowledge restricting access. In Chapters 3 and 4, we demonstrated how effective synthesis can be at providing an interface to meaningful analysis of real-world data, and in Chapter 5 we presented a technique for automatically proving high-probability guarantees that relied on synthesis to generate axioms for probability distributions. Lastly, in Chapter 6, we illustrated how incorporating domain knowledge can improve the performance of synthesis across a variety of domains. There is, however, more to be said and much more to be done. In this chapter we will outline several promising lines for future work, before providing some closing remarks.

7.1 Future Directions and Extensions

Program synthesis for data access is a natural arms race—for every paper titled *Program Synthesis for* X published, three new protocols to automate come to light. Instead of enumerating all possible values for X, we will frame two concrete challenges extending the work in this dissertation: (*i*) improving the data efficiency of privacy-preserving queries, and (*ii*) synthesizing high utility programs.

Improving Data Efficiency of Privacy-Preserving Queries

Recall from Definition 2.1 that our notion of differential privacy includes a parameter δ that we immediately assume is 0; this case is called *pure* differential privacy. If we relax our guarantee to *approximate* differential privacy, when $\delta > 0$, we can more efficiently utilize our privacy budget through more advanced composition mechanisms (Winograd-Cort et al.,

2017; Near et al., 2019) that answer adaptive *sequences* of queries. We would, therefore, like to synthesize programs in these frameworks.

But adapting the approach of Chapter 4 to systems with adaptive composition is not straightforward, in part because tools like Duet utilize two-tiered languages with (*i*) a private component and (*ii*) a non-private component. This complicates the type system, and introduces challenges in *decomposing* the synthesis problem into a sequence of adaptive subproblems. A templated approach, such as the one we introduce in Chapter 3, provides a weak form of decomposition, while a framework like Prose (Polozov and Gulwani, 2015) provides a slightly stronger one. Neither, however, are suited for the adaptive case. Therefore, the construction of a notion of adaptive decomposition for program synthesis is necessary to enable the use of more efficient privacy systems.

Synthesizing High Utility Programs

It is easy to construct a differentially private query (through the gratuitous application of noise), but it is not easy to construct a highly useful query—one needs to reason about the interaction of control flow and random sampling as it pertains to a high-probability guarantee. Further, high-probability guarantees with symbolic failure probabilities define a Pareto frontier with the accuracy-reliability tradeoff, it is not even clear how to determine if one query is *more useful* than another on an arbitrary workload.

Assuming a high-probability guarantee is provided, we might reasonably want to synthesize a program satisfying it. Our automatic proof technique, however, is difficult to use during synthesis, as (*i*) every execution takes on the order of tens of seconds, and (*ii*) we have no means of applying it to a partial program. The first point is potentially addressable with optimizations and clever engineering, but the second point requires we reason about *partial proofs*.

Yet, the Hoare-style logic used in Chapter 5 is inherently *compositional*, and that is promising. Polikarpova and Sergey (2019) have already demonstrated how one can convert a Hoare-style program logic into an efficient synthesis algorithm. In particular, the existence of separating conjunction provides a strong modality that can be manipulated to direct the search. It is not clear if such a modality exists for an inversion of probabilistic trace abstraction, but if one can be found and exploited, it would enable program synthesis directed by high-probability guarantees.

7.2 Closing Remarks, or: a Promise

Lack of privacy and data availability are real problems that have real impacts on real people, while the work presented in this dissertation only exists in journal articles and code repositories. Bridging that gap between research and practice takes time, effort, and focused communication. In the case of program synthesis for data accessibiilty, all the techniques in the world mean little if we don't reach out and directly apply them towards making that digital panopticon public use. This is not a sermon; these words are a promise to ourself that we will put in the work to make our work meaningful. May our promise be your motivation to do the same.

A PROPERTIES OF DATA-PARALLEL SYNTHESIS

A.1 Proofs of Soundness and Completeness

Proof of Theorem 3.5. Trivially from the application of the rules (VERIFY) and (CONS).

Specifically, let p be returned by (VERIFY). (CONS) ensures condition 1 of Definition 3.2 holds (by explicitly constructing p from an $h \in H$ and a map $\nu : W \to T_{\Sigma^C}$), while (VERIFY) ensures conditions 2 and 3 of Definition 3.2 holds by explicitly checking if $p \models E$ and $\textsc{Determ}(p)$ hold. □

To prove Theorem 3.6, we first present two supporting lemmas.

Lemma A.1. *Let τ and Γ by a type and context, respectively, and let $p \in \tau^\Gamma$ be a program that is a (partial) completion of an incomplete program q. Then $q \in \tau^\Gamma$.*

Proof of Lemma A.1. If p is a completion of q, then there exists a witness $\nu : W \to T_{\Sigma^C}$ such that $p = \nu^*(q)$. Further, there exists a type map σ such that $\sigma\Gamma \vdash p : \sigma\tau$. Now, construct the type map δ as follows:

$$\forall \bullet \in \textsc{Dom}(\nu), \delta(\bullet) = \tau', \text{ where } \tau', \Gamma' = \textsc{Infer}(\bullet, p)$$

Note $\Gamma' \subseteq \Gamma$, as $\nu(\bullet)$ is a sub-term of p and is therefore well-typed in Γ, and that $\textsc{Dom}(\delta) \cap \textsc{Dom}(\sigma) = \emptyset$. Note also that, by construction,

$$\sigma\delta\Gamma \vdash q : \sigma\delta\tau$$

as δ maps the wildcards no longer in p to the type of their replacement in p. Therefore, $\sigma \circ \delta$ is a witness to $q \in \tau^\Gamma$. □

Lemma A.2. *Suppose τ^Γ is in the domain of M on a fair execution of the algorithm in which the rule (VERIFY) is never applied. Then, every complete program $p \in \tau^\Gamma$ will eventually appear in $M(\tau^\Gamma)$.*

Proof of Lemma A.2. We proceed by induction on the depth n of p.

Base Let $n = 1$. If p is a variable v or constant c, then by fairness it must eventually be added to $M(\tau^\Gamma)$.

App Let $p = f(e_1, \dots, e_n) \in \tau^\Gamma$ be complete. Construct $q = f(\bullet, \dots, \bullet)$. By Lemma A.1, $q \in \tau^\Gamma$, and by the inductive hypothesis, each e_i eventually appears in M in the appropriate context. This enables the application of (PApp) using q and $e_1, \dots, e_n$, which by fairness eventually applies to produce $p \in \tau^\Gamma$.

Abs Let $p = \lambda x.\, e$. The proof is similar to the case for abstraction.

□

We can now prove Theorem 3.6.

Proof of Theorem 3.6. Let p satisfy the correctness constraint ϕ_S for a synthesis task $S = (E, C, H)$, and let $h \in H$ and $v : W \to T_{\Sigma^C}$ be chosen such that $p = v^*(h)$. Let $\bullet \in \mathit{wild}(h)$. By construction of $\textsc{Infer}(\cdot, \cdot)$, $v(\bullet) \in \tau^\Gamma$, where $\tau, \Gamma = \textsc{Infer}(\bullet, h)$.

Therefore, by Lemma A.2, $v(\bullet)$ (or some α-equivalent program) will eventually appear in M. Eventually, $v(\bullet)$ will appear in M for all $\bullet \in \mathit{wild}(h)$, enabling the application of (Cons). By fairness, the execution will eventually construct p from h using the assignment v, and again, by fairness, (Verify) will eventually apply on p and succeed. □

A.2 Proof of Validity of CSG-Checking

Proof of Theorem 3.8. We will prove each direction by contradiction.

(⇒) Let r be a binary function, and R its encoding. Suppose that VC_R is not valid: the formula

$$\varphi_{com} \wedge \varphi_{assoc} \wedge \neg\psi_{CSG}$$

is therefore satisfiable. Let M be the witnessing model. By construction of ψ_{CSG}, M must satisfy at least one of the formulas (*i*) $o_1 \neq o_2$ or (*ii*) $o_3 \neq o_5$.

1. Suppose $M \models o_1 \neq o_2$. Then, by construction of R, $M[i_1]$ and $M[i_2]$ are witnesses to the non-commutativity of r.
2. Suppose $M \models o_3 \neq o_5$. Then, by construction of R, $M[i_1]$, $M[i_2]$, and $M[i_3]$ are witnesses to the non-associativity of r.

($\Leftarrow$) Suppose (τ, r) is not a CSG. Then one of the following cases must hold:

1. Suppose r is not commutative. Then, by construction of R, there must be some values $c_1, c_2 \in \tau$ such that the formula

$$R(c_1, c_2, o_1) \wedge R(c_2, c_1, o_2) \wedge o_1 \neq o_2$$

   where o_1, o_2 are free variables, is satisfiable. Therefore, VC_R cannot be valid.

2. Suppose r is not associative. Then, by construction of R, there must be some values $c_1, c_2, c_3 \in \tau$ such that the formula

$$R(c_1, c_2, o_1) \wedge R(o_1, c_3, o_3) \wedge R(c_2, c_3, o_4) \wedge R(c_1, o_4, o_5) \wedge o_3 \neq o_5$$

   where $o_1, \ldots, o_5$ are free variables, is satisfiable. Therefore, VC_R cannot be valid.

$\square$

B PROPERTIES OF PRIVACY-AWARE SYNTHESIS

B.1 Multisets and Metric Preservation

In our slight variant of DFuzz , we introduce the following subtyping rule for multisets:

$$\frac{\Phi, \Gamma \models \sigma \sqsubseteq \tau}{\Phi \wedge S \leqslant T, \Gamma \models \texttt{mset}\,[\sigma]\,[S] \sqsubseteq \texttt{mset}\,[\tau]\,[T]}\;(\sqsubseteq.\textsc{mset})$$

and state without elaboration that such a rule can integrate with DFuzz's proof of metric preservation. Here, we clarify this point through an example and inspection of the proof.

Example Multiset Comparison

As we have introduced a new type, so must we introduce an appropriate *metric rule* for values of the chosen type. We will add the following rule:

$$\frac{\emptyset \vdash v_1 : \texttt{mset}\,[\sigma]\,[S] \quad \emptyset \vdash v_2 : \texttt{mset}\,[\sigma]\,[S] \quad r = |v_1 \triangle v_2|}{\vdash v_1 \sim_r v_2 : \texttt{mset}\,[\sigma]\,[S]}$$

where $|v_1 \triangle v_2|$ is the size of the *symmetric difference* between v_1 and v_2. This rule is similar to the one originally presented for databases: the only distinction is the presence of a *precise type*.

Consider two multisets, v_1 and v_2, with elements of type σ and size bounds of S and T, respectively. Now assume S and T are such that

$$\emptyset \vdash v_1 : \texttt{mset}\,[\sigma]\,[S], \quad \emptyset \vdash v_2 : \texttt{mset}\,[\sigma]\,[T].$$

For this to be possible, S and T must be concrete values in $\mathbb{N} \cup \{\infty\}$. Therefore, using the subtyping rule ($\sqsubseteq$.MSET), we can arrive at the judgements

$$\emptyset \vdash v_1 : \texttt{mset}\,[\sigma]\,[K], \quad \emptyset \vdash v_2 : \texttt{mset}\,[\sigma]\,[K],$$

where $K = \max(S, T)$. Applying the metric rule introduced above, we can derive

$$\vdash v_1 \sim_{|v_1 \,\triangle\, v_2|} v_2 : \texttt{mset}\,[\sigma]\,[K].$$

Now we can compare multisets, regardless of size bounds, by losing precision in the dependent indices.

Proof of Metric Preservation

The original presentation of DFuzz contains a chain of lemmas that culminate in a proof of metric preservation. To account for our new subtyping and metric rule, we need only modify one of these lemmas, whose critical statement is given below.

Lemma B.1. *Suppose* $\Gamma \vdash e : \tau$. *If* $\vdash \delta_1 \sim_\gamma \delta_2 : \Gamma^\circ$ *and* $\delta_1 e$ *is a value, then* $\delta_2 e$ *must also be a value and* $\vdash \delta_1 e \sim_{\gamma [\![\Gamma]\!]} \delta_2 e : \tau$.

Proof of Lemma B.1. By induction on the judgement deriving $\Gamma \vdash e : \tau$. We only need to consider an extension to the subtyping case

$$\frac{\Gamma \vdash e : \sigma \quad \emptyset \models \sigma \sqsubseteq \tau}{\Gamma \vdash e : \tau}\ (\sqsubseteq .R)$$

By the inductive hypothesis, $\delta_2 e$ is a value and

$$\vdash \delta_1 e \sim_{\gamma [\![\Gamma]\!]} \delta_2 e : \sigma.$$

We now consider the new case that σ is $\texttt{mset}\,[\alpha]\,[S]$ for some type α and size term S. By inversion, we have

$$\frac{\vdash \delta_1 e \sim_{\gamma [\![\Gamma]\!]} \delta_2 e : \texttt{mset}\,[\alpha]\,[S]}{\emptyset \vdash \delta_1 e : \texttt{mset}\,[\alpha]\,[S] \quad \emptyset \vdash \delta_2 e : \texttt{mset}\,[\alpha]\,[S] \quad \gamma\,[\![\Gamma]\!] = |\delta_1 e \,\triangle\, \delta_2 e|}$$

Note τ must be $\texttt{mset}\,[\alpha]\,[T]$ for some size term T. By hypothesis, we have

$$\emptyset \models \texttt{mset}\,[\alpha]\,[S] \sqsubseteq \texttt{mset}\,[\alpha]\,[T].$$

We immediately derive the judgements (replacing $\texttt{mset}\,[\alpha]\,[T]$ by τ)

$$\emptyset \vdash \delta_1 e : \tau, \quad \emptyset \vdash \delta_2 e : \tau,$$

and consequently, by application of our new metric rule,

$$\vdash \delta_1 e \sim_{\gamma[\![\Gamma]\!]} \delta_2 e : \tau.$$

$\square$

This lemma allows us to extend our new metric rule to the metric relation over expressions (and states) when said expressions are also values (and final states). Our proof above is a necessary and sufficient addition to prove the rest of the chain leading to metric preservation.

B.2 Proof of Soundness

Our search enjoys guarantees of soundness. To aid in the discussion, we first introduce some notation to capture the process of using the inference rules defining the search:

1. Let $\rightsquigarrow_I$ be the smallest relation between synthesis problems and synthesis states consistent with the rule (Init). That is, if $S = \langle \sigma, C, \phi_k, \phi_s \rangle$ is a synthesis task (Definition 4.4), and $s = \langle \phi,\ e \rangle$ is a synthesis state, we write $S \rightsquigarrow_I s$ if and only if we can infer s from S using rule (Init).

2. Let $\rightsquigarrow_S$ be the smallest relation between two synthesis states consistent with all inference rules except (Init) and (Finish). So if s and s' are two synthesis states, we write $s \rightsquigarrow_S s'$ if and only if a single application of an allowed inference rule lets us derive s' from s.

3. Let $\rightsquigarrow_F$ be the smallest relation between synthesis states and programs consistent with the rule (FINISH). We write $s \rightsquigarrow_F p$ if and only if an application of (FINISH) lets us derive p from s.

When clear from context, we will drop the subscript and write $\rightsquigarrow$. Furthermore, we will abuse notation and treat the above non-subscripted relations as a single relation (referred to as $\rightsquigarrow$), as in $S \rightsquigarrow^* p$, where $\rightsquigarrow^*$ is the transitive-reflexive closure of $\rightsquigarrow$.

The following definitions and lemmas will aid us in proving our intuitive notion of *soundness*: the inference rules defining the synthesis cannot produce a program that fails to be a solution to the provided synthesis problem.

Definition B.2. *Let* M *be an assignment of variables in the standard model of symbolic context constraints, expressions-as-terms, and types-as-terms.* M *induces an interpretation of* symbolic contexts $[\![\Omega]\!]$, *and an interpretation of expressions and types —* $[\![e]\!]$ *and* $[\![\sigma]\!]$, *respectively — that replaces all free type and sensitivity variables with the assignment given by* M.

We will write $\Omega \vdash_M e : \sigma$ *to mean that the typing judgement* $\Omega \vdash e : \sigma$ *is valid with respect to the assignment* M, *or more formally:* $[\![\Omega]\!] \vdash [\![e]\!] : [\![\sigma]\!]$.

Lemma B.3. *Let* $\langle \sigma_s, C, \phi_k, \phi_s \rangle \rightsquigarrow^* \langle \phi,\ p \rangle$. *For all* $M \models \phi$, *for all wildcards* $\bullet_{\sigma_i}^{\Omega_i} \in \textsc{Wild}(p)$, *and for all closed expressions* e_i *such that* $\Omega_i \vdash_M e_i : \sigma_i$, *the following judgement holds:*

$$\Gamma_C \vdash_M p \left[\bullet_{\sigma_i}^{\Omega_i} / e_i\right]_i : \sigma$$

Proof of Lemma B.3. The proof is by induction over the inference rules defining $\rightsquigarrow$. The rule (INIT) represents the base case, while every other rule is an inductive step. We prove the illustrative cases below: the rest are structurally similar.

(Init) Assume $\langle \sigma_s, \Sigma, \phi_k, \phi_s \rangle \rightsquigarrow_I \langle \phi,\ p \rangle$ via the (Init) inference rule. Then $\phi = \phi_k$ and $p = \bullet^{\Sigma}_{\sigma_s}$. Let M be a model such that $M \models \phi_k$, and let e be a closed term such that $[\![\Sigma]\!] \vdash [\![e]\!] : [\![\sigma_s]\!]$. Clearly, it must be the case that

$$[\![\Sigma]\!] \vdash \left[\!\left[p \left[\bullet^{\Sigma}_{\sigma_s} / e \right] \right]\!\right] : [\![\sigma_s]\!] ,$$

and the base case is complete.

(Abs) Consider the derivation of synthesis states

$$\left\langle \phi,\ p \left[\bullet^{\Omega}_{\tau \multimap_r \sigma} \right] \right\rangle \rightsquigarrow_S \left\langle \phi \wedge \Omega' = \Omega, \{x :_r \tau\},\ p \left[\lambda x : \tau.\, \bullet^{\Omega'}_{\sigma} \right] \right\rangle$$

via the inference rule (Abs), and assume the inductive hypothesis holds for the initial state. Then let $M \models \phi \wedge \Omega' = \Omega, \{x :_r \tau\}$, and let e be a concrete term such that $\Omega' \vdash_M e : \sigma$. As M is an assignment in the standard model of symbolic context constraints, we can derive the typing judgement

$$\Omega, \{x :_r \tau\} \vdash_M e : \sigma,$$

and by application of the DFuzz typing rule for $\multimap$-introduction, we derive the typing judgement

$$\Omega \vdash_M \lambda x : \tau.\, e : \tau \multimap_r \sigma.$$

Now, for every wildcard

$$\bullet^{\Omega_i}_{\sigma_i} \in \textsc{Wild}(p \left[\lambda x : \tau.\, \bullet^{\Omega'}_{\sigma} \right]) / \bullet^{\Omega'}_{\sigma},$$

let e_i be a concrete term such that $\Omega_i \vdash_M e_i : \sigma_i$. As

$$p \left[\lambda x : \tau.\, \bullet^{\Omega'}_{\sigma} \right] \left[\bullet^{\Omega'}_{\sigma} / e, \bullet^{\Omega_i}_{\sigma_i} / e_i \right]_i = p \left[\bullet^{\Omega}_{\tau \multimap_r \sigma} \right] \left[\bullet^{\Omega}_{\tau \multimap_r \sigma} / \lambda x : \tau.\, e, \bullet^{\Omega_i}_{\sigma_i} / e_i \right]_i ,$$

by application of the inductive hypothesis we arrive at the conclusion that

the following typing judgement holds:

$$\Sigma \vdash_M p \left[\lambda x : \tau.\, \bullet_{\sigma}^{\Omega'}\right] \left[\bullet_{\sigma}^{\Omega'}/e, \bullet_{\sigma_i}^{\Omega_i}/e_i\right]_i : \sigma_s$$

(App) Consider the derivation of synthesis states

$$\left\langle \phi,\ p\left[\bullet_{\sigma}^{\Omega}\right]\right\rangle \rightsquigarrow_S \left\langle \phi \wedge \Omega = \Omega_1 + r \cdot \Omega_2,\ p\left[\bullet_{\tau \multimap_r \sigma}^{\Omega_1}\ \bullet_{\tau}^{\Omega_2}\right]\right\rangle$$

via the inference rule (App), and assume the inductive hypothesis holds for the initial state. Then let $M \models \phi \wedge \Omega = \Omega_1 + r \cdot \Omega_2$, and let e_1 and e_2 be concrete terms such that $\Omega_1 \vdash_M e_1 : \tau \multimap_r \sigma$ and $\Omega_2 \vdash_M e_2 : \tau$. As M is an assignment in the standard model of symbolic context constraints, via application of the DFuzz rule for $\multimap$-elimination we derive the typing judgement

$$\Omega_1 + r \cdot \Omega_2 \vdash_M e_1\ e_2 : \sigma$$

(TApp) Consider the derivation of synthesis states

$$\left\langle \phi,\ p\left[\bullet_{\sigma}^{\Omega}\right]\right\rangle \rightsquigarrow_S \left\langle \phi,\ p\left[\bullet_{\forall t.\tau}^{\Omega}[t']\right]\right\rangle$$

via the inference rule (TApp), and assume the inductive hypothesis holds for the initial state. Then let $M \models \phi$, and let e be a concrete term such that $\Omega \vdash_M e : \forall t.\, \tau$. By assumption we have $\sigma \prec_{[t/t']} \tau$, so applying $\forall$-elimination we derive the typing judgement

$$\Omega \vdash_M e[t'] : \sigma$$

(Id) Consider the derivation of synthesis states

$$\left\langle \phi,\ p\left[\bullet_{\sigma}^{\Omega}\right]\right\rangle \rightsquigarrow_S \langle \phi \wedge \gamma \wedge \psi \wedge \Omega = \{v :_1 \tau\},\ p\,[v]\rangle$$

via the inference rule (Id), and assume the inductive hypothesis holds

for the initial state. Then let $M \models \phi \wedge \gamma \wedge \psi \wedge \Omega = \{v :_1 \tau\}$. As M is an assignment in the standard model of symbolic context constraints, $[\![\Omega]\!]$ contains a use of v, and so we derive the judgement

$$\Omega \vdash_M v : \tau.$$

By assumption, $\gamma; \psi \vdash \tau \leftarrow \sigma$, so the fact that $M \models \gamma \wedge \psi$ is sufficient to give

$$\top \models [\![\tau]\!] \sqsubseteq [\![\sigma]\!],$$

which combined with DFuzz 's typing rule $\sqsubseteq$ .R immediately implies the desired judgement

$$\Omega \vdash_M v : \sigma$$

The rest of the inductive cases are proven in the same manner: assume the inductive hypothesis to get a model M, use derived synthesis state constraints and the fact that M is an assignment in the standard model of symbolic context constraints to apply the appropriate DFuzz typing rule and derive the required conclusion. □

Now we can prove soundness.

Theorem B.4 (Soundness, Reformalized from Theorem 4.18)**.** *Let* $S = \langle \sigma, C, \phi_k, \phi_s \rangle$ *be a synthesis problem. If* $S \rightsquigarrow^* p$*, then* p *is a solution to* S*.*

Proof of Theorem B.4. Let $s = \langle \phi, p \rangle$ be a subproblem such that $S \rightsquigarrow^* s \rightsquigarrow_F p$. We will show that $p \models \phi_S$, the *correctness constraint* induced by S (Definition 4.4):

1. p is derived from s by an application of (Finish), and so $p \models_d \phi_s$.

2. The solution p is a complete term, so $wild(p) = \emptyset$. Further, we know $\textsc{Sat}(\phi)$, so let M be a witness and apply Lemma B.3 to derive the desired judgement

$$\Gamma_C \vdash_M p : \sigma$$

A natural consequence of $M \models \phi$ is that $M \models \phi_k$.

Since p satisfies the two conditions necessary for $p \models \phi_S$, p is a solution of S. □

B.3 Proof of Relative Completeness

We cannot derive *all* possible programs p from a given synthesis problem. Our inference rules place the following restrictions on p:

1. p must contain only primitives from C, and
2. p cannot contain type-, size-, or sensitivity-abstractions.

These restrictions are necessary to ensure the search is well-defined and to avoid undecidability in the type system. An algorithm derived from our inference rules should be able (given reasonable assumptions about fair application of inference rules) to find any such p that is a solution to the given synthesis problem. This formalizes our notion of relative completeness:

Theorem B.5 (Relative Completeness, Reformalized from Theorem 4.19)**.** *Let* $S = \langle \sigma, C, \phi_k, \phi_s \rangle$ *be a synthesis problem, and let* p *be a solution to* S*. Then* $S \rightsquigarrow^* \langle \phi,\ p \rangle$ *where* p *is complete and* $\textsc{Sat}(\phi)$*.*

Proof of Theorem B.5. Let p be a solution. Then there is an application of rules that — if we ignore constraints and focus just on the types and structure — will generate p. Since p is a solution it is clearly a complete term, so we just need to show $\textsc{Sat}(\phi)$.

First, note that p a solution to S implies there is some assignment M such that $\Gamma_C \vdash_M p : \sigma$ and $M \models \phi_k$.

Clearly p is closed, as it is a solution - our goal is to show $\textsc{Sat}(\phi)$. Let φ be a conjunct from ϕ. There are two kinds of conjuncts that can be added to ϕ:

1. *A symbolic context constraint*: Our inference rules are inversions of DFuzz typing rules, and the symbolic context constraints produced are the most general restriction to still allow type-checking. As such, since $\Gamma_C \vdash_M p : \sigma$, it must be that $M \models \varphi$.

2. *An abduction constraint*: Abduction constraints encode constraints to allow subtyping. However, since the subtyping relation is encoded in the type judgement, and $\Gamma_C \vdash_M p : \sigma$, by abduction most-generality, $M \models \varphi$.

Therefore, $M \models \phi$ and $\textsc{Sat}(\phi)$. □

B.4 Pruning the Search

Observe that each inference rule that generates constraints does so by conjoining them to the old constraints. This forms the basis of a pruning strategy:

Theorem B.6 (Pruning). *Let $s = \langle \phi, p \rangle$ be a subproblem, and let $s' = \langle \phi', p' \rangle$ be a subproblem such that $s \rightsquigarrow^* s'$. Then $\textsc{Unsat}(\phi) \Rightarrow \textsc{Unsat}(\phi')$.*

Proof of Theorem B.6. Since ϕ' is derived from ϕ by conjoining new obligations, $\phi' \Rightarrow \phi$. So $\textsc{Sat}(\phi') \Rightarrow \textsc{Sat}(\phi)$, which is the contrapositive of our claim. □

Using Theorem B.6, it is easy to see that we never need to explore a branch of the search that has an unsatisfiable proof obligation. Any synthesis state derivable in that branch will never have a satisfiable obligation, and consequently we will never be able to apply the inference rule (Finish) to return a synthesis solution.

B.5 Subtyping Constraint Abduction

To complete the inference rules defining our abduction judgement, we must add two rules excluded from the paper. The first rule is that for precise real numbers:

$$\frac{}{\top; S = S' \vdash_{\emptyset} \mathbb{R}[S] \leftarrow \mathbb{R}[S']} \text{ (REAL)}$$

And the second rule is that for tuples:

$$\frac{\gamma; \psi \vdash_A \sigma_1 \leftarrow \sigma_2 \quad \delta; \phi \vdash_B \tau_1 \leftarrow \tau_2}{\gamma \wedge \delta; \psi \wedge \phi \vdash_{A \cup B} \langle \sigma_1, \tau_1 \rangle \leftarrow \langle \sigma_2, \tau_2 \rangle} \text{ (TUPLE)}$$

B.6 Proof of Abduction Most-Generality

Proof of Theorem 4.11. The proof proceeds by induction over the inference rules defining abduction. Assume $\gamma; \psi \vdash \sigma \leftarrow \tau$, and let δ and ϕ be constraints such that $\delta; \phi \vdash \sigma \leftarrow \tau$:

(REFL) By hypothesis, $\gamma = \top$ and $\psi = \top$, so the claim is trivially true.

(LVAR) By assumption, t has no free sensitivity variables. The only way to subtype, then, is to make $\nu = t$. Therefore, for all assignments $M \models \delta \wedge \phi$, it must be the case that $M \models \nu = t$, and so the claim holds.

(RVAR) Symmetric to the above case.

(NAT) If $\sigma = \mathbb{N}[S]$ and $\tau = \mathbb{N}[S']$, the only way to subtype is to ensure $S = S'$. Therefore, for all assignments $M \models \delta \wedge \phi$, it must be the case that $M \models S = S'$, and so the claim holds.

(REAL) Symmetric to the above case.

(List) By assumption, $\sigma = L(\sigma')[S]$ and $\tau = L(\tau')[S']$, and $\gamma;\psi \vdash \sigma' \leftarrow \tau'$. Now, by the induction hypothesis, $\delta \wedge \varphi \Rightarrow \gamma \wedge \psi$. Of course, since $\delta \wedge \varphi$ is enough to ensure subtyping, by a symmetric argument to the previous case for any assignment $M \models \delta \wedge \varphi$, it must hold that $M \models S = S'$. So $\delta \wedge \varphi \Rightarrow \gamma \wedge \psi \wedge S = S'$, and the claim holds.

(MSet) By assumption, $\sigma = \texttt{mset}\,[\sigma']\,[S]$ and $\tau = \texttt{mset}\,[\tau']\,[S']$, and $\gamma;\psi \vdash \sigma' \leftarrow \tau'$. By the induction hypothesis, $\delta \wedge \varphi \Rightarrow \gamma \wedge \psi$, and by a symmetric argument to the previous case, using the subtyping rule for precise multisets, we arrive at the fact that for any assignment $M \models \delta \wedge \varphi$, it must be true that $M \models S' \leqslant S$, and so the claim holds.

(Modal) Symmetric to the above case.

(Arrow) By assumption, $\sigma = \sigma_1 \multimap \tau_1$ and $\tau = \sigma_2 \multimap \tau_2$, and $\gamma;\psi \vdash \sigma_2 \leftarrow \sigma_1$ and $\gamma';\psi' \vdash \tau_1 \leftarrow \tau_2$. By induction, $\delta \wedge \varphi \Rightarrow \gamma \wedge \psi$ and $\delta \wedge \varphi \Rightarrow \gamma' \wedge \psi'$. Therefore, $\delta \wedge \varphi \Rightarrow \gamma \wedge \psi \wedge \gamma' \wedge \psi'$, and the claim holds.

(Monad) Symmetric to the proof for the codomain in the above case.

(Tuple) Symmetric to two copies of the above case.

(ForAll) By assumption, $\sigma = \forall\alpha.\,\sigma'$ and $\tau = \forall\beta.\,\tau'$, and $\gamma;\psi \vdash_A \sigma'\,[\alpha/\rho] \leftarrow \tau'\,[\beta/\rho]$ with $\rho \notin A$. As variables are only put in A via application of (LVar) or (RVar), we know that ρ is unconstrained by γ and ψ. By induction $\delta \wedge \varphi \Rightarrow \gamma \wedge \psi$, and as ρ is unconstrained this continues to hold when ρ is abstracted out of σ and τ.

As all cases are covered, the proof is complete. □

B.7 Proof of Equisatisfiability

Proof of Theorem 4.17. One direction is straightforward: any model of $\varphi_d \wedge \varphi_c$ must also choose values for the sensitivity variables present in the

symbolic contexts defined in ϕ_c. So if $\mathfrak{A} \models \phi_d \wedge \phi_c$, it must also be that $\mathfrak{A} \models \phi_d \wedge \Delta(\phi_c)$.

For the other direction, let $\mathfrak{A}$ be a model such that $\mathfrak{A} \models \phi_d \wedge \Delta(\phi_c)$ and—without loss of generality—contains only variables that appear in $\phi_d \wedge \Delta(\phi_c)$. By construction,

$$\phi_c = \bigwedge_{i=1}^{n} C^{l,i} = C^{r,i},$$

and

$$\Delta(\phi_c) = \bigwedge_{x \in \textsc{Support}(\phi_c)} \bigwedge_{i=1}^{n} S_x^{l,i} = S_x^{r,i} \wedge \phi_x^{l,i} \wedge \phi_x^{r_i}.$$

Let $\psi_{x,i}$ be a conjunct in $\Delta(\phi_c)$ of the form that is, for fixed x and i,

$$\psi_{x,i} \coloneqq S_x^{l,i} = S_x^{r,i} \wedge \phi_x^{l,i} \wedge \phi_x^{r_i}$$

for some fixed x and i.

Note that, by construction, $\psi_{x,i}$ is an equality constraint in the theory of non-linear real arithmetic. Therefore, we can rewrite $\psi_{x,i}$ into the following form:

$$\psi_{x,i} \coloneqq p(s_1, \ldots, s_n, r_x^{\Omega_1}, \ldots, r_x^{\Omega_m}) = 0 \wedge \phi_x^{l,i} \wedge \phi_x^{r_i},$$

where p is some polynomial in $n + m$ variables, $s_1, \ldots, s_n$ are sensitivity variables that appear explicitly in the *untransformed* ϕ_c, and $r_x^{\Omega_1}, \ldots, r_x^{\Omega_m}$ are fresh sensitivity variables introduced in the construction of the symbolic sensitivity constraint $\Delta(\phi_c)$.

We will use $\psi_{x,i}$ to construct an extension of $\mathfrak{A}$—which we will call $\mathfrak{A}_{x,i}$—as follows:

$$\mathfrak{A}_{x,i}(\Omega_k) \coloneqq \begin{cases} \left\{ x :_{\mathfrak{A}\left(r_x^{\Omega_k}\right)} \tau_x \right\} & \text{if } \Omega_k \text{ is not bound} \\ \mathfrak{A}(\Omega_k) \cup \left\{ x :_{\mathfrak{A}\left(r_x^{\Omega_k}\right)} \tau_x \right\} & \text{if } \Omega_k \text{ is already bound} \end{cases}$$

for $1 \leqslant k \leqslant m$, and $\mathfrak{A}_{x,i} = \mathfrak{A}$ everywhere else. The type τ_x can either be inferred from ϕ_c (if x has a provided type, uniqueness is guaranteed by preservation of type safety during synthesis), or *consistently* set to an arbitrary type.

Now, we can define the model $\mathfrak{A}_S$ as follows:

$$\mathfrak{A}_S \coloneqq (\dots(\mathfrak{A}_{x_1,i_1})_{x_2,i_2}\dots)_{x_s,i_s},$$

where $x_j, i_j \in \textsc{Support}(\phi_c) \times \{1, 2, \dots, n\}$ and $s = |\textsc{Support}(\phi_c)| \cdot n$. That is, we extend our initial model $\mathfrak{A}$ by every formula $\psi_{x,i}$ present in $\Delta(\phi_c)$.

As $\mathfrak{A}_S$ is an extension of $\mathfrak{A}$, we naturally have $\mathfrak{A}_S \models \phi_d$. It just remains to be seen that $\mathfrak{A}_S \models \phi_c$.

Let $\varphi \coloneqq C_i^l = C_i^r$ be a conjunct of ϕ_c. By construction, $\mathfrak{A}_S$ binds all variables that appear in φ, but it is not immediately the case that $\mathfrak{A}_S \models \varphi$. Assume, for the sake of contradiction, that $\mathfrak{A}_S \not\models \varphi$. As context equality is variable-wise, there are only three ways for $\mathfrak{A}_S$ to fail to satisfy φ:

1. There exists expression variable x such that $(x :_R \tau) \in \mathfrak{A}_S(C_i^l)$ and $(x :_{R'} \sigma) \in \mathfrak{A}_S(C_i^r)$, but $\sigma \neq \tau$. By construction, if σ or τ appear in a concrete context in C_i^l or C_i^r, then they appear uniquely and so all instances of x will be given that concrete type in $\mathfrak{A}_S$. If σ and τ do *not* appear in the symbolic contexts, then, again by construction, all instances of x are given the same type in $\mathfrak{A}_S$.

2. There exists expression variable x such that $(x :_R \tau) \in \mathfrak{A}_S(C_i^l)$ and $(x :_{R'} \sigma) \in \mathfrak{A}_S(C_i^r)$, but $\mathfrak{A}_S(R) \neq \mathfrak{A}_S(R')$. By choice of model, all sensitivity variables that might appear in R or R' were already defined in $\mathfrak{A}$. Since $\mathfrak{A} \models \Delta(\phi_c)$, $\mathfrak{A} \models R = R'$, giving us an immediate contradiction.

3. There exists some expression variable x such that (without loss of generality) $x \in \textsc{Dom}(\mathfrak{A}_S(C_i^l))$ but $x \notin \textsc{Dom}(\mathfrak{A}_S(C_i^r))$. Instead of treating finite contexts as *finite sets of bindings*, instead we can simply interpret finite con-

texts as if they bound *all expression variables*, only finitely-many of which have non-zero sensitivity. Then the domain of every context in the model $\mathfrak{A}_S$ is all possible expression variables, and this case effectively reduces down to the other two.

Consequently, $\mathfrak{A}_S \models \varphi$. The above argument applies to *all* conjuncts of ϕ_c, and so $\mathfrak{A}_S \models \phi_c$ and the desired result follows immediately. $\square$

C PROOFS FOR PROBABILISTIC TRACE ABSTRACTION

C.1 Proving Proof Rules

Proof of Theorem 5.1. By definition of the semantics, the output distribution of a program P on input state s is

$$\llbracket P \rrbracket (s) = \sum_{\tau \in \mathcal{L}(P)} \llbracket \tau \rrbracket (s)$$

Hence for any input state $s \in \varphi_{\mathsf{pre}}$, we have

$$\llbracket P \rrbracket (s)(\overline{\varphi_{\mathsf{post}}}) = \sum_{\tau \in \mathcal{L}(P)} \llbracket \tau \rrbracket (s)(\overline{\varphi_{\mathsf{post}}}) \leqslant \sum_{\tau \in \mathcal{L}(\mathcal{A})} \llbracket \tau \rrbracket (s)(\overline{\varphi_{\mathsf{post}}}) \leqslant s(\beta)$$

by the trace inclusion and failure probability upper bound conditions. □

Proof of Theorem 5.5. Let $s \in \varphi_{\mathsf{pre}}$ be any input state satisfying the pre-condition. For each automaton A_i, the pre-condition inclusion condition implies that $s \in \varphi_{\mathsf{pre}} \subseteq \lambda_i(q_i^{\mathsf{in}})$ and so Theorem 5.4 gives

$$\sum_{\tau \in \mathcal{L}(A_i)} \llbracket \tau \rrbracket (s)(\overline{\lambda_i(q_i^{\mathsf{ac}})}) \leqslant s(\kappa_i(q_i^{\mathsf{ac}}))$$

By the post-condition inclusion property, we also have $\overline{\varphi_{\mathsf{post}}} \subseteq \overline{\lambda_i(q_i^{\mathsf{ac}})}$ and so

$$\sum_{\tau \in \mathcal{L}(A_i)} \llbracket \tau \rrbracket (s)(\overline{\varphi_{\mathsf{post}}}) \leqslant s(\kappa_i(q_i^{\mathsf{ac}})).$$

Finally we can conclude by the trace inclusion and failure probability

upper bound conditions:

$$\begin{aligned}
[\![P]\!]\,(s)(\overline{\varphi_{\text{post}}}) &= \sum_{\tau \in \mathcal{L}(P)} [\![\tau]\!]\,(s)(\overline{\varphi_{\text{post}}}) \\
&\leqslant \sum_{i=1}^{n} \sum_{\tau \in \mathcal{L}(A_i)} [\![\tau]\!]\,(s)(\overline{\varphi_{\text{post}}}) \\
&\leqslant \sum_{i=1}^{n} s(\kappa_i(q_i^{\text{ac}})) \leqslant s(\beta)
\end{aligned}$$

□

Proof of Theorem 5.4. We first consider the simpler case when A has no directed loops. In such an automaton, the valuation of the deterministic variables V^{det} at any node q_i is the same for all execution traces starting at q^{in} with initial state s_0 and reaching q_i; we write ν_i for these valuations, and we write ν_{in} and ν_{ac} for these valuations at q^{in} and q^{ac}, respectively.

We need to work with a slightly more general version of well-labeled automata, where the initial and final nodes are labeled by a function of the deterministic variables V^{det}. We show that for any initial state $s_0 \in \lambda(q^{\text{in}})$, we have

$$\sum_{\tau \in \mathcal{L}(A)} \mu(\overline{\lambda(q^{\text{ac}})}) \leqslant \nu_{ac}(\kappa(q^{\text{ac}})) - \nu_{in}(\kappa(q^{\text{in}})).$$

where $\mu = [\![\tau]\!]\,(s_0)$ is the output distribution. Note that when $\kappa(q^{\text{in}}) = 0$ and $\kappa(q^{\text{ac}})$ is labeled by input variables V^{in} only, we recover:

$$\sum_{\tau \in \mathcal{L}(A)} \mu(\overline{\lambda(q^{\text{ac}})}) \leqslant s_0(\kappa(q^{\text{ac}})).$$

The proof is by induction on the number k of branches (i.e., nodes with two outgoing e

In the base case $k = 0$, the automaton represents a sequential composition $\not{s}_1; \cdots ; \not{s}_n$. Let the corresponding nodes be $q_0, \dots, q_n$, with $q_0 = q^{\text{in}}$

and $q_n = q^{ac}$. Since the probability labels $\lambda(q_i)$ depend on deterministic variables only, given any initial state $s_0 \in \kappa(q_0)$ there is a sequence of valuations $v_0, \dots, v_n$ for the deterministic variables such that the deterministic variables V^{det} of any state with non-zero probability in $[\![st_1; \cdots ; st_i]\!] (s_0)$ are set to v_i, with $v_0 = s_0(V^{det})$. By the well-labeled condition, we have:

$$\vdash_{v_i(\kappa(q_i)) - v_{i-1}(\kappa(q_{i-1}))} \{\lambda(q_{i-1}) \wedge V^{det} = v_{i-1}\}\, st_i \,\{\lambda(q_i) \wedge V^{det} = v_i\}$$

By the sequential composition rule of the union bound logic, we have

$$\vdash_{v_n(\kappa(q^{ac})) - v_0(\kappa(q^{in}))} \{\lambda(q^{in}) \wedge V^{det} = v_0\}\, st_1; \cdots ; st_n \,\{\lambda(q^{ac})\}$$

By definition, $v_n = v_{ac}$ and $v_0 = v_{in}$ so we have

$$\vdash_{v_{ac}(\kappa(q^{ac})) - v_{in}(\kappa(q^{in}))} \{\lambda(q^{in}) \wedge V^{det} = v_{in}\}\, st_1; \cdots ; st_n \,\{\lambda(q^{ac})\}$$

and we conclude by soundness of the union bound logic.

Now, suppose there are $k > 0$ branches in A. Starting from the initial node q^{in}, let the first branching node be q_r with outgoing edges to q_t and q_f, labeled by $\mathsf{assume}(b)$ and $\mathsf{assume}(\neg b)$ respectively. We let A_0 be the sub-automaton with initial node q^{in} and final node q_r; note that this automaton is a single path along nodes $q_0 = q^{in}, q_1, \dots, q_r$ with edge labels $st_1, \dots, st_r$. Letting $\mu_r = [\![A_0]\!] (s_0)$ be the output distribution of this automaton, the base case yields

$$\sum_{\tau \in \mathcal{L}(A_0)} \mu_r(\overline{\lambda(q_r)}) \leqslant v_r(\kappa(q_r)) - v_{in}(\kappa(q^{in})).$$

Now, we consider the rest of the automaton. Let A_t be the sub-automaton of all reachable nodes starting from q_t, and let A_f be the sub-automaton starting from q_f. Note that A_t and A_f are both well-labeled automata with entry nodes q_t and q_f respectively, and have at most $k - 1$ branching

nodes each. Since the assume statements do not modify this variables, ν is also the deterministic valuation of V^{det} at the entry nodes of q_t and q_f. By induction, for any state $s \in \lambda(q_b)$ such that $s(V^{det}) = \nu_r$ and $b \in \{t, f\}$ we have

$$\sum_{\tau \in \mathcal{L}(A_b)} [\![\tau]\!]\,(s)(\overline{\lambda(q^{ac})}) \leqslant \nu_{ac}(\kappa(q^{ac})) - \nu_r(\kappa(q_b))$$

To combine our bounds for A_0, A_t, A_f back together, we assume that the state labels at the branching node q_r satisfy

$$\lambda(q_r) \subseteq \lambda(q_t) \cap \{s \mid s(b)\} \qquad \text{and} \qquad \lambda(q_r) \subseteq \lambda(q_f) \cap \{s \mid s(\neg b)\}.$$

If either fails, then the edge condition for well-labeled automata ensures that $\nu_{ac}(q^{ac}) - \nu_r(q_r) \geqslant \nu_r(q_b) - \nu_r(q_r) \geqslant 1$ and so $\nu_{ac}(q^{ac}) - \nu_{in}(q^{in}) \geqslant 1$, and our target bound is trivial. Now, every trace in $\mathcal{L}(A)$ is of the form $q_0, \dots, q_r, q_b, \dots, q^{ac}$ for $b = t$ or $b = f$; since A has no loops, the trace after q_r is entirely contained in A_b.

Now, we decompose $\mu_r = \mu_t + \mu_f + \mu_{err}$ into three pieces:

1. μ_{err} is the restriction to states not in $\lambda(q_r)$;
2. μ_t is the restriction to states in $\lambda(q_r)$ with b is true;
3. μ_f is the restriction to states not in $\lambda(q_r)$ with b false.

Note that all states in the support of μ_t and μ_f lie in $\lambda(q_t)$ and $\lambda(q_f)$, respectively. Since A_0, A_t, A_f are all loop free with at most $k - 1$ branches, applying the induction hypothesis gives

$$\begin{aligned}
\sum_{\tau \in \mathcal{L}(A)} [\![\tau]\!]\,(s_0)(\overline{\lambda(q^{ac})}) &\leqslant \mathit{bind}(\mu_t, [\![A_t]\!])(\overline{\lambda(q^{ac})}) + \mathit{bind}(\mu_f, [\![A_f]\!])(\overline{\lambda(q^{ac})}) + |\mu_{err}| \\
&\leqslant |\mu_t| \cdot (\nu_{ac}(\kappa(q^{ac})) - \nu_r(\kappa(q_t))) + |\mu_f| \cdot (\nu_{ac}(\kappa(q^{ac})) - \nu_r(\kappa(q_f))) \\
&\quad + (\nu_r(\kappa(q_r)) - \nu_{in}(\kappa(q^{in}))) \\
&\leqslant |\mu_t| \cdot (\nu_{ac}(\kappa(q^{ac})) - \nu_r(\kappa(q_r))) + |\mu_f| \cdot (\nu_{ac}(\kappa(q^{ac})) - \nu_r(\kappa(q_r))) \\
&\quad + (\nu_r(\kappa(q_r)) - \nu_{in}(\kappa(q^{in}))) \\
&= (|\mu_t| + |\mu_f|) \cdot \nu_{ac}(\kappa(q^{ac})) + (1 - |\mu_t| - |\mu_f|) \cdot \nu_r(\kappa(q_r)) - \nu_{in}(\kappa(q^{in})) \\
&\leqslant \nu_{ac}(\kappa(q^{ac})) - \nu_{in}(\kappa(q^{in}))
\end{aligned}$$

where the first inequality is due to semantics, the second is the inductive hypothesis, the third uses $\kappa(q_r) \leqslant \kappa(q_b)$, and the fourth relies on $\kappa(q_r) \leqslant \kappa(q^{ac})$.

Finally, we consider the general case where A may have directed loops. The basic idea is to reduce to the acyclic case we have just considered by performing finite unrollings of A. The argument uses standard constructions on automata and regular expressions (see, e.g., prior work giving an algebraic view of program schemes (Angus and Kozen, 2001)); we just sketch the proof here. Let C be the set of all statements appearing in A. We can view A as a deterministic automaton D over the alphabet $\Sigma = Q \times Q \times C$ by viewing each transition $q_i \xrightarrow{\mathscr{I}} q_j$ as a transition on letter $(q_i, q_j, \mathscr{I})$. To make this a deterministic automaton, we can add a new dead node q_{dead} with a self loop on all letters, and add a transition from every existing node $q \in Q$ to q_{dead} on all letters that don't appear as outgoing transitions from q in A. Then, we mark q^{ac} as the sole accepting node in D. Now, the language $\mathcal{L}_D$ accepted by D is evidently equal to the language $\mathcal{L}(A)$ of all traces in A.

By Kleene's theorem, this language can also be represented as a regular expression R over Σ. Now, we can define finite unrollings in terms of R. For $n \in \mathbb{N}$, let R_n be the regular expression obtained by repeatedly replacing each sub-term r^* where r is star-free by the finite approximation $1 + r + \cdots + r^n$; the order of replacement will not matter for our purposes. Now $\mathcal{L}(R) = \cup_n \mathcal{L}(R_n)$, and $\mathcal{L}(R_i) \subseteq \mathcal{L}(R_j)$ for all $i \leqslant j$. Again by Kleene's theorem, the language of each R_n is recognized by a deterministic finite automaton; let D_n be a minimal automaton for each R_n.

Now since the language of R_n is finite and D_n is minimal, the only cycles in D_n must occur as self-loops on a single (non-accepting) dead node p_n. All transitions from the initial node to non-dead nodes must be labeled by $(q^{in}, -, -)$. There are at most two such transitions since there are at most two transitions out of q^{in} in the original automaton A, and if

there are two transitions they must be of the form $(q^{\mathsf{in}}, q_t, \mathsf{assume}(b))$ and $(q^{\mathsf{in}}, q_f, \mathsf{assume}(\neg b))$. By a similar inductive argument, each non-dead node has at most two outgoing transitions to non-dead nodes and if there are two transitions, they are of the form $(q, q_t, \mathsf{assume}(b))$ and $(q, q_f, \mathsf{assume}(\neg b))$. Thus, we can associate each node p in D_n with a node $a(p)$ in A and convert D_n to a well-labeled acyclic automaton A_n by labeling $\lambda(p) \coloneqq \lambda(a(p))$ and $\kappa(p) \coloneqq \kappa(a(p))$ and removing the dead node; note that $\mathcal{L}(A_n) = \mathcal{L}(D_n) = \mathcal{L}(R_n)$.

Finally, let s be any initial state in $\lambda(q^{\mathsf{in}})$. By reduction to the acyclic case, we have

$$\sum_{\tau \in \mathcal{L}(A_n)} [\![\tau]\!]\,(s)(\overline{\lambda(q^{\mathsf{ac}})}) \leqslant s(\kappa(q^{\mathsf{ac}}))$$

for every $n \in \mathbb{N}$. Since the left-hand side is increasing in n and bounded above by $s(\kappa(q^{\mathsf{ac}}))$, the limit exists and we have

$$\lim_{n\to\infty} \sum_{\tau \in \mathcal{L}(A_n)} [\![\tau]\!]\,(s)(\overline{\lambda(q^{\mathsf{ac}})}) \leqslant s(\kappa(q^{\mathsf{ac}})).$$

But since $\mathcal{L}(A_n)$ is increasing and $\cup_n \mathcal{L}(A_n) = \mathcal{L}(A)$, we conclude

$$\sum_{\tau \in \mathcal{L}(A)} [\![\tau]\!]\,(s)(\overline{\lambda(q^{\mathsf{ac}})}) \leqslant s(\kappa(q^{\mathsf{ac}})).$$

□

C.2 Proofs of Soundness

Proof of Theorem 5.11. We show by induction on the derivation of rules used by the algorithm that the automaton set $\mathcal{A}$ is always well-labeled, and each automaton in $\mathcal{A}$ satisfies the pre- and post-condition inclusion properties in Theorem 5.5. The base case, rule INIT, is trivial. Each trace added to the automaton set by TRACE is well-labeled by construction, and

the simplification rules GENERALIZE and MERGE keep the automaton set well-labeled by definition (Lemma 5.10). Finally, if the termination rule CORRECT fires, then the automata are well-labeled and satisfy pre- and post-condition inclusion properties by induction, and the side-conditions guarantee the trace inclusion and failure probability upper bound conditions. Therefore by Theorem 5.5, the accuracy judgment is valid. □

We first begin by proving the following lemma, which captures correctness of the encoding of τ. Specifically, the following lemma formalizes the correspondence between models of the encoding and the support of the output distribution of τ: we show that for any initial state s, the models of the logical encoding correspond to a set of states R_s and a failure probability c such that $[\![\tau]\!](s)(\overline{R_s}) \leqslant c$.

Lemma C.1 (Soundness of *enc*)**.** *Fix trace* $\tau = \not{s}t_1; \cdots; \not{s}t_n$. *Let*

$$\varphi \coloneqq \omega_0 = 0 \land h_0 = \mathit{false} \land \bigwedge_{i=1}^{n} \mathit{enc}(i, \not{s}t_i)$$

where all uninterpreted functions resulting from distribution axiom families have been given a fixed interpretation. Fix a state $s \in S$. *Let* $M1, \ldots, M_m$ *be the set of models of* φ *such that* $s(V^{\mathsf{in}}) = M_i(V^{\mathsf{in}})$ *and* $M_i(h_n) = \mathit{false}$*, for all* $i \in [1, m]$. *Let*

$$R_s = \{s' \mid s'(V) = M_i(V)\}$$

Then, for any $M_s \models \varphi$ *such that* $M_s(V^{\mathsf{in}}) = s(V^{\mathsf{in}})$*, we have* $[\![\tau]\!](s)(\overline{R_s}) \leqslant M_s(\omega_n)$.

Proof of Lemma C.1. Note that all models $M_s \models \varphi$ such that $M_s(V^{\mathsf{in}}) = s(V^{\mathsf{in}})$ agree on the value of ω_i. This is because the constraints ω_i are functions of V^{in}. Next, note that by construction, there is always $M_s \models \varphi$, so φ is never unsatisfiable.

We proceed by induction on the length of τ. For $n = 1$, we have three cases. Fix an s as in lemma statement.

Case 1 $\tau = v \leftarrow e$. We have $\varphi \coloneqq v = e \wedge \omega_1 = 0 \wedge h_1 = \mathit{false}$ (after simplification). From φ, $M_s(\omega_1) = 0$. Therefore, lemma states: $\llbracket v \leftarrow e \rrbracket(s)(\overline{R_s}) \leqslant 0$. Suppose this does not hold, then, by definition of $\llbracket v \leftarrow e \rrbracket$, there is a state $s' \in S \setminus R_s$ such that $s' = s[v \mapsto s(e)]$. However, by definition of φ, $s' \in R_s$, since $M_i(V) = s'(V)$, for some i.

Case 2 $\tau = \mathsf{assume}(b)$. We have $\varphi \coloneqq \omega_1 = 0 \wedge (h_1 = \neg b)$ (after simplification). From φ, $M_s(\omega_1) = 0$. Therefore, lemma states: $\llbracket \mathsf{assume}(b) \rrbracket(s)(\overline{R_s}) \leqslant 0$. Suppose this does not hold, then, by definition of $\llbracket \mathsf{assume}(b) \rrbracket$, we have $s \in S \setminus R_s$ and $s(b) = \mathit{true}$. However, by definition of φ, $s \in R_s$, iff $s(b) = \mathit{true}$.

Case 3 $\tau = v \sim d$. We have $\varphi \coloneqq \omega_1 = e^{\mathsf{ub}} \wedge h_1 = \varphi^{\mathsf{ax}}$ (after simplification). Lemma states that $\llbracket v \sim d \rrbracket(s)(\overline{R_s}) \leqslant M_s(e^{\mathsf{ub}})$. This follows from the definition of a distribution axiom: that $\Pr_{v \sim d}[\varphi^{\mathsf{ax}}] \leqslant e^{\mathsf{ub}}$ is true for any valuation of $V \setminus \{v\}$.

Assume that lemma holds for traces of length n. We show that it also holds for $n + 1$, where τ' is a trace of length n.

Case 1 $\tau = \tau'; v \leftarrow e$. The encoding is $\varphi \coloneqq \varphi' \wedge v = e \wedge \omega_{n+1} = \omega_n \wedge h_{n+1} = h_n$. Let $M'_1, \ldots, M'_m$ and R'_s be defined for φ' and τ', as per lemma statement. By hypothesis, $\llbracket \tau' \rrbracket(s)(\overline{R'_s}) \leqslant M_s(\omega_n)$. By semantics of assignment, we have $\llbracket \tau'; v \leftarrow e \rrbracket(s)(\overline{X}) \leqslant M_s(\omega_{n+1})$, where $X = \{s \mid s' \in R'_s, s = s'[v \leftarrow s'(e)]\}$. Observe that $X = R_s$: by definition of φ, its models are a subset of $\{M'_1, \ldots, M'_m\}$ such that $v = e$. It then follows that $\llbracket \tau \rrbracket(s)(\overline{R_s}) \leqslant M_s(\omega_{n+1})$.

Case 2 $\tau = \tau'; \mathsf{assume}(b)$. The encoding is $\varphi \coloneqq \varphi' \wedge \omega_n = \omega_{n-1} \wedge (h_n = (h_{n-1} \vee \neg b))$. Let $M'_1, \ldots, M'_m$ and R'_s be defined for φ' and τ', as per lemma statement. By hypothesis, $\llbracket \tau' \rrbracket(s)(\overline{R'_s}) \leqslant M_s(\omega_n)$. We know that models $\{M_i\}$ of φ are a subset of $\{M'_i\}$ where b is true. Therefore $\overline{R_s} \supseteq \overline{R'_s}$.

But we have that all states in $\overline{R_s} \setminus \overline{R'_s}$ are those were b is false. By definition of $[\![\mathsf{assume}(b)]\!]$, all those states are assigned probability 0. Therefore, $[\![\tau]\!](s)(\overline{R_s}) \leqslant M_s(\omega_{n+1})$.

Case 3 $\tau = \tau'; \nu \sim d$. The encoding is $\varphi \coloneqq \varphi' \wedge \omega_{n+1} = \omega_n + e^{\mathsf{ub}} \wedge h_{n+1} = (h_n \vee \varphi^{\mathsf{ax}})$. Let $M'_1, \dots, M'_m$ and R'_s be defined for φ' and τ', as per lemma statement. By hypothesis, $[\![\tau']\!](s)(\overline{R'_s}) \leqslant M_s(\omega_n)$. Let X be the set of all states that satisfy $\neg\varphi^{\mathsf{ax}}$. From φ, we know that $R_s = R'_s \cap X$. By the union bound and the distribution axiom, $[\![\tau'; \nu \sim d]\!](s)(\overline{R'_s} \cup \overline{X}) \leqslant M_s(\omega_{n+1})$

□

Now, correctness of Theorem 5.15 follows from Lemma C.1.

Proof of Theorem 5.14. Soundness of the Bernoulli and Uniform axioms is straightforward. The Laplace axiom is Barthe et al. (2016c, Lemma 5). The exponential axiom follows from the Laplace axiom, noting that

$$s(\textsc{Exp}(\nu_1, \nu_2))(z) \leqslant 2 \cdot s(\mathrm{Lap}\,(\nu_1, \nu_2))(z)$$

for all $z > s(\nu_1)$, so the failure probability for the exponential axiom is at most twice the failure probability for the Laplace axiom. □

C.3 Proof of Well-Labeling from Interpolants

Proof of Theorem 5.18. Notice that by construction we have $\lambda(q^{\mathsf{in}}) \coloneqq \mathit{true}$ and $\kappa(q^{\mathsf{in}}) = 0$.

We first show that $\lambda(q^{\mathsf{ac}}) \Rightarrow \varphi_{\mathsf{post}}$. By construction of encoding, the formula $\bigwedge_{i=1}^{n} \varphi_i \Rightarrow (\neg h_n \Rightarrow \varphi_{\mathsf{post}})$ is valid. By definition of sequence interpolants, the formula $\psi_n \Rightarrow (\neg h_n \Rightarrow \varphi_{\mathsf{post}})$ is valid. Therefore $\psi_n[h_n \mapsto \mathit{false}] \Rightarrow \varphi_{\mathsf{post}}$ is valid. Since ω_n does not appear in φ_{post}, $\exists \omega_n.\, \psi_n[h_n \mapsto \mathit{false}] \Rightarrow \varphi_{\mathsf{post}}$ is valid.

Second, we show that $\varphi_{\mathsf{pre}} \Rightarrow \kappa(q^{ac}) \leqslant \beta$. By construction, $\kappa(q^{ac}) \coloneqq f(V^{in})$, where $f(V^{in})$ is the function that returns, for any valuation of V^{in}, the largest value of ω_n that satisfies $\exists V \setminus V^{in}.\, \exists h_n.\, \psi_n$. By definition of sequence interpolants: $\psi_n \Rightarrow \omega_n \leqslant \beta$. Since β is over V^{in}, we have $\exists V \setminus V^{in}.\, \exists h_n.\, \psi_n \Rightarrow \omega_n \leqslant \beta$ is valid. Pick a model M of φ_{pre} with the largest possible ω_n interpretation that satisfies $\exists V \setminus V^{in}.\, \exists h_n.\, \psi_n$. By construction of the encoding this model exists, since any model satisfying φ_{pre} can be extended to a model of $\bigwedge_i \varphi_i$. It follows that this model satisfies $\omega_n \leqslant \beta$.

Finally, we need to show that for every edge $q_i \xrightarrow{st} q_j$, where $j = i + 1$, we have

$$\vdash_{wp^f(\kappa(q_j), st) - \kappa(q_i)} \{\lambda(q_i)\}\, st\, \{\lambda(q_j)\}$$

We break up the proof by statement type:

Assignment From definition of seq. interpolants, we know the following is valid

$$\psi_i \wedge v = e \wedge \omega_j = \omega_i \wedge h_j = h_i \Rightarrow \psi_j$$

Set h_i to *false* on left-hand side of implication. The following is valid:

$$\psi_i[h_i \mapsto \mathit{false}] \wedge v = e \wedge \omega_j = \omega_i \wedge h_j = \mathit{false} \Rightarrow \psi_j$$

It follows that we can set h_j to *false* on both sides, resulting in the following valid statement:

$$\psi_i[h_i \mapsto \mathit{false}] \wedge v = e \wedge \omega_j = \omega_i \Rightarrow \psi_j[h_j \mapsto \mathit{false}]$$

Weaken rhs by existentially quantifying ω_j. The following is valid:

$$\psi_i[h_i \mapsto \mathit{false}] \wedge v = e \wedge \omega_j = \omega_i \Rightarrow \exists \omega_j.\, \psi_j[h_j \mapsto \mathit{false}]$$

Since ω_i is, by encoding, a function of V^{in}, we can project it out on the lhs.

The following is valid:

$$(\exists \omega_i.\, \psi_i[h_i \mapsto \mathit{false}]) \land \nu = e \land \omega_j = \omega_i \Rightarrow \exists \omega_j.\, \psi_j[h_j \mapsto \mathit{false}]$$

As a result, we can drop the $\omega_j = \omega_i$ constraint, resulting in the following valid statement:

$$(\exists \omega_i.\, \psi_i[h_i \mapsto \mathit{false}]) \land \nu = e \Rightarrow \exists \omega_j.\, \psi_j[h_j \mapsto \mathit{false}]$$

This implies that the following Hoare triple, since $\lambda(q_i) \equiv \exists \omega_i.\, \psi_i[h_i \mapsto \mathit{false}]$ and $\lambda(q_j) \equiv \exists \omega_j.\, \psi_j[h_j \mapsto \mathit{false}]$:

$$\vdash_c \{\lambda(q_i)\}\, \mathit{st}\, \{\lambda(q_j)\}$$

for any $c \in [0, 1]$.

It now remains to show that $\mathit{wp}^f(\kappa(q_j), \mathit{st}) - \kappa(q_i) \geqslant 0$, for any state s in $\lambda(q_i)$. From our constraint, for any values of ω_i and V^{det} that satisfy $\exists V \setminus V^{det}.\, \exists h_i.\, \psi_i$, the same values where $\omega_j = \omega_i$ also satisfy $\exists V \setminus V^{det}.\, \exists h_j.\, \psi_j$. Therefore, it is always the case that $\mathit{wp}^f(\kappa(q_j), \mathit{st}) - \kappa(q_i) \geqslant 0$

Sample Following a similar simplification path to the one we used for assignment statements, we arrive at the following valid statement:

$$(\exists \omega_i.\, \psi_i[h_i \mapsto \mathit{false}]) \land \neg \varphi^{ax} \Rightarrow \exists \omega_j.\, \psi_j[h_j \mapsto \mathit{false}]$$

Since we know that $\Pr[[] \varphi^{ax}] \leqslant e^{ub}$, from the applied axiom, this implies that the following Hoare triple, since $\lambda(q_i) \equiv \exists \omega_i.\, \psi_i[h_i \mapsto \mathit{false}]$ and $\lambda(q_j) \equiv \exists \omega_j.\, \psi_j[h_j \mapsto \mathit{false}]$:

$$\vdash_{e^{ub}} \{\lambda(q_i)\}\, \mathit{st}\, \{\lambda(q_j)\}$$

It now remains to show that $\mathit{wp}^f(\kappa(q_j), \mathit{st}) - \kappa(q_i) \geqslant e^{ub}$, for any state s in

$$\begin{aligned} enc(i, v \leftarrow e) &\coloneqq v = e \wedge \omega_i = \omega_{i-1} \\ enc(i, \mathsf{assume}(b)) &\coloneqq b \wedge \omega_i = \omega_{i-1} \\ enc(i, v \sim d) &\coloneqq \neg\varphi^{\mathsf{ax}} \wedge \omega_i = \omega_{i-1} + e^{\mathsf{ub}} \\ &\quad \text{given: } \Pr[v \sim d]\ \varphi^{\mathsf{ax}} \leqslant e^{\mathsf{ub}} \end{aligned}$$

Figure C.1: Simplified logical encoding of statement semantics for feasible traces

$\lambda(q_i)$.

Following argument from base case of Lemma C.1, we establish the specification. From our constraint, for any values of ω_i and V^{det} that satisfy $\exists V \setminus V^{\mathsf{det}}.\ \exists h_i.\ \psi_i$, the same values where $\omega_j = \omega_i + e^{\mathsf{ub}}$ also satisfy $\exists V \setminus V^{\mathsf{det}}.\ \exists h_j.\ \psi_j$. Therefore, it is always the case that $wp^{\mathsf{f}}(\kappa(q_j), st) - \kappa(q_i) \geqslant e^{\mathsf{ub}}$

Assume Similar to sampling statements.

□

C.4 A Simplified Encoding

The encoding in Section 5.5 is designed for full generality: it assumes that a trace may be infeasible, which is why it introduces the auxiliary variables h_i to track states that cannot make it through the trace. In the case where the trace is feasible for some input states, the encoding and interpolation problems become much simpler by doing away with the auxiliary h_i variables. The simplified version of *enc* is shown in Figure C.1.

Henceforth we assume that for a trace τ, all Boolean expressions appearing in assume statements are over V^{det}. Second, we assume that there is a state s such that $\tau(s)$ is a distribution.

Theorem C.2 (Soundness of simplified encoding)**.** *The specification* $\vdash_\beta \{\varphi_{\mathsf{pre}}\}\ \mathit{st}_1, \ldots, \mathit{st}_n\ \{\varphi_{\mathsf{post}}\}$ *is valid if the following formula is satisfiable:*

$$\forall V, \omega_i.\ \left(\varphi_{\mathsf{pre}} \wedge \omega_0 = 0 \wedge \bigwedge_{i=1}^{n} enc(i, \mathit{st}_i)\right) \implies (\omega_n \leqslant \beta \wedge \varphi_{\mathsf{post}}) \qquad \text{(C.1)}$$

Theorem C.2 follows from the next lemma:

Lemma C.3 (Soundness of simplified *enc*)**.** *Fix trace* $\tau = \mathit{st}_1; \cdots; \mathit{st}_n$*. Let* $\varphi \coloneqq \omega_0 = 0 \wedge \bigwedge_{i=1}^{n} enc(i, \mathit{st}_i)$*, where all uninterpreted functions resulting from distribution axiom families have been given a fixed interpretation. Fix a state* $s \in S$*. Let* $M_1, \ldots, M_m$ *be the set of models of* φ *such that* $s(V^{\mathsf{in}}) = M_i(V^{\mathsf{in}})$*, for all* $i \in [1, m]$*. Let*

$$R_s = \{s' \mid s'(V) = M_i(V)\}$$

Then, for any M_i*, we have* $[\![\tau]\!]\,(s)(\overline{R_s}) \leqslant M_i(\omega_n)$*.*

Proof of Lemma C.3. First, we note that all models $M_i \models \varphi$ agree on the value of ω_i. This is because the constraints ω_i are functions of V^{in}. Second, note that by our assumption, there is always $M_i \models \varphi$—i.e., it is never unsatisfiable.

We proceed by induction on the length of τ. For $n = 1$, we have three cases. Fix an s as in lemma statement.

Case 1 $\tau = v \leftarrow e$. We have $\varphi \coloneqq v = e \wedge \omega_1 = 0$. From φ, $M_i(\omega_1) = 0$, for all i. Therefore, lemma states: $[\![v \leftarrow e]\!]\,(s)(\overline{R_s}) \leqslant 0$. Suppose this does not hold, then, by definition of $[\![v \leftarrow e]\!]$, there is a state $s' \in S \setminus R_s$ such that $s' = s[v \mapsto s(e)]$. However, by definition of φ, $s' \in R_s$, since $M_i(V) = s'(V)$, for some i.

Case 2 $\tau = \mathsf{assume}(b)$. We have $\varphi \coloneqq b \wedge \omega_1 = 0$. From φ, $M_i(\omega_1) = 0$, for all i. Therefore, lemma states: $[\![\mathsf{assume}(b)]\!]\,(s)(\overline{R_s}) \leqslant 0$. Suppose this does not hold, then, by definition of $[\![\mathsf{assume}(b)]\!]$, we have $s \in S \setminus R_s$ and $s(b) = \mathit{true}$. However, by definition of φ, $s \in R_s$, iff $s(b) = \mathit{true}$.

Case 3 $\tau = v \sim d$. We have $\varphi \coloneqq \neg\varphi^{\mathsf{ax}} \wedge \omega_1 = e^{\mathsf{ub}}$. Lemma states that $[\![v \sim d]\!]\,(s)(\overline{R_s}) \leqslant M_i(e^{\mathsf{ub}})$, for all i. This follows from the definition of a distribution axiom: that $\Pr_{v \sim d}[\varphi^{\mathsf{ax}}] \leqslant e^{\mathsf{ub}}$ is true for any valuation of $V \setminus \{v\}$.

Assume that lemma holds for traces of length n. We show that it also holds for $n+1$, where τ' is a trace of length n.

Case 1 $\tau = \tau'; v \leftarrow e$. The encoding is $\varphi \coloneqq \varphi' \wedge v = e \wedge \omega_{n+1} = \omega_n$.

Let $M'_1, \ldots, M'_m$ and R'_s be defined for φ' and τ', as per lemma statement. By hypothesis, $[\![\tau']\!]\,(s)(\overline{R'_s}) \leqslant M'_i(\omega_n)$. By semantics of assignment, we have $[\![\tau'; v \leftarrow e]\!]\,(s)(\overline{X}) \leqslant M'_i(\omega_{n+1})$, where $X = \{s \mid s' \in R'_s, s = s'[v \leftarrow s'(e)]\}$. Observe that $X = R_s$: by definition of φ, its models are a subset of $\{M'_1, \ldots, M'_m\}$ such that $v = e$. It then follows that $[\![\tau]\!]\,(s)(\overline{R_s}) \leqslant M_i(\omega_{n+1})$, for all i.

Case 2 $\tau = \tau'; \mathsf{assume}(b)$. The encoding is $\varphi \coloneqq \varphi' \wedge b \wedge \omega_n = \omega_{n-1}$. Let $M'_1, \ldots, M'_m$ and R'_s be defined for φ' and τ', as per lemma statement. By hypothesis, $[\![\tau']\!]\,(s)(\overline{R'_s}) \leqslant M'_i(\omega_n)$, for all i. We know that models $\{M_i\}$ of φ are a subset of $\{M'_i\}$ where b is *true*. Therefore $\overline{R_s} \supseteq \overline{R'_s}$. But we have that all states in $\overline{R_s} \setminus \overline{R'_s}$ are those were b is false. By definition of $[\![\mathsf{assume}(b)]\!]$, all those states are assigned probability 0. Therefore, $[\![\tau]\!]\,(s)(\overline{R_s}) \leqslant M_i(\omega_{n+1})$, for all i.

Case 3 $\tau = \tau'; v \sim d$. The encoding is $\varphi \coloneqq \varphi' \wedge \neg\varphi^{\mathsf{ax}} \wedge \omega_{n+1} = \omega_n + e^{\mathsf{ub}}$. Let $M'_1, \ldots, M'_m$ and R'_s be defined for φ' and τ', as per lemma statement. By hypothesis, $[\![\tau']\!]\,(s)(\overline{R'_s}) \leqslant M'_i(\omega_n)$, for all i. Let X be the set of all states that satisfy $\neg\varphi^{\mathsf{ax}}$. From φ, we know that $R_s = R'_s \cap X$. By the union bound and the distribution axiom, $[\![\tau'; v \sim d]\!]\,(s)(\overline{R'_s} \cup \overline{X}) \leqslant M_i(\omega_{n+1})$, for all i.

□

Assume we construct a sequence of interpolants for the above encoding as described in Section 5.5. Then, the following theorem holds, which is the same as Theorem 5.18, but without handling h_i variables.

Theorem C.4 (Well-Labelings from Interpolants)**.** *Let* $\{\psi_i\}_i$ *be the interpolants computed as shown above. Let* $A_\tau = \langle Q, \delta, \lambda, \kappa \rangle$ *be the failure automaton that accepts only the trace* $\tau = \mathit{st}_1, \ldots, \mathit{st}_n$*, i.e.,* $\delta = \{q^{\mathsf{in}} \xrightarrow{\mathit{st}_1} q_1, q_1 \xrightarrow{\mathit{st}_2} q_2, \ldots q_{n-1} \xrightarrow{\mathit{st}_n} q^{\mathsf{ac}}\}$*. Set the labeling functions as follows:*

1. $\lambda(q^{\mathsf{in}}) \coloneqq \varphi_{\mathsf{pre}}$ *and* $\kappa(q^{\mathsf{in}}) \coloneqq 0$*.*
2. $\lambda(q_i) \coloneqq \exists \omega_i . \psi_i$ *and* $\lambda(q^{\mathsf{ac}}) \coloneqq \exists \omega_n . \psi_n$*.*
3. $\kappa(q_i) \coloneqq f(V^{\mathsf{det}})$*, where* $f(V^{\mathsf{det}})$ *is the function that returns, for any valuation of* V^{det}*, the largest value of* ω_i *that satisfies* $\exists V \setminus V^{\mathsf{det}} . \psi_i$*. For* $\kappa(q^{\mathsf{ac}})$*, we use* $\exists V \setminus V^{\mathsf{in}} . \psi_n$*.*

Then, A_τ *is well-labeled and implies* $\vdash_\beta \{\varphi_{\mathsf{pre}}\}\, \tau \,\{\varphi_{\mathsf{post}}\}$*.*

Proof of Theorem C.4. Similar to the proof of Theorem 5.18. □

C.5 Capturing the Union Bound Logic

Our trace abstraction technique is inspired by the union bound logic (aHL), proposed by Barthe et al. (2016c). The core rules of this program logic are presented in Figure C.2; the only omitted rules are the ones for the skip command (trivial to add to our language) and the rules for external and internal procedure calls (we do not consider interprocedural analysis). We comment briefly on a few rules; the others are largely standard. The sampling rule [Rand] encodes distribution axioms. The most complicated rule is [While]—intuitively, the side-conditions ensure that there is a non-increasing integer variant e_v whose initial value bounds the maximum number of loop iterations. The program logic also features an interesting complement of structural rules. Along with the usual rule of consequence [Weak] and rule of constancy [Frame], the disjunction rule [Or] combines two pre-conditions (keeping the failure probability unchanged) and the conjunction rule [And] combines two post-conditions, while summing

$$\frac{}{\vdash_0 \{\Phi[v \mapsto e]\}\, v \leftarrow e\, \{\Phi\}}\ \textsc{Assn} \qquad \frac{\forall s.\ s(\Phi) \Rightarrow \Pr_{[\![v \sim d]\!](s)}(\neg\Psi) \leqslant s(\beta)}{\vdash_\beta \{\Phi\}\, v \sim d\, \{\Psi\}}\ \textsc{Rand}$$

$$\frac{\vdash_\beta \{\Phi\}\, P\, \{\Psi\} \qquad \vdash_{\beta'} \{\Psi\}\, P'\, \{\Theta\}}{\vdash_{\beta+\beta'} \{\Phi\}\, P\, ;\, P'\, \{\Theta\}}\ \textsc{Seq}$$

$$\frac{\vdash_\beta \{\Phi \wedge b\}\, P\, \{\Psi\} \qquad \vdash_\beta \{\Phi \wedge \neg b\}\, P'\, \{\Psi\}}{\vdash_\beta \{\Phi\}\, \textbf{if}\ b\ \textbf{then}\ P\ \textbf{else}\ P'\, \{\Psi\}}\ \textsc{If}$$

$$\frac{\begin{array}{c}\forall s, k.\ s(\Phi \wedge b \wedge e_v = k) \Rightarrow \Pr_{[\![P]\!](s)}(e_v \geqslant k) = 0 \\ e_v : \mathbb{N} \qquad \models \Phi \wedge e_v \leqslant 0 \Rightarrow \neg b \qquad \vdash_\beta \{\Phi \wedge b\}\, P\, \{\Phi\}\end{array}}{\vdash_{\rho \cdot \beta} \{\Phi \wedge e_v \leqslant \rho\}\, \textbf{while}\ b\ \textbf{do}\ P\, \{\Phi \wedge \neg b\}}\ \textsc{While}$$

$$\frac{\models (\Phi' \Rightarrow \Phi) \wedge (\Psi \Rightarrow \Psi') \wedge (\beta \leqslant \beta') \qquad \vdash_\beta \{\Phi\}\, P\, \{\Psi\}}{\vdash_{\beta'} \{\Phi'\}\, P\, \{\Psi'\}}\ \textsc{Weak}$$

$$\frac{MV(P) \cap FV(\Phi) = \emptyset}{\vdash_0 \{\Phi\}\, P\, \{\Phi\}}\ \textsc{Frame} \qquad \frac{\vdash_\beta \{\Phi\}\, P\, \{\Psi\} \qquad \vdash_{\beta'} \{\Phi\}\, P\, \{\Psi'\}}{\vdash_{\beta+\beta'} \{\Phi\}\, P\, \{\Psi \wedge \Psi'\}}\ \textsc{And}$$

$$\frac{\vdash_\beta \{\Phi\}\, P\, \{\Psi\} \qquad \vdash_\beta \{\Phi'\}\, P\, \{\Psi\}}{\vdash_\beta \{\Phi \vee \Phi'\}\, P\, \{\Psi\}}\ \textsc{Or} \qquad \frac{}{\vdash_1 \{\Phi\}\, P\, \{\bot\}}\ \textsc{False}$$

Figure C.2: The union bound logic, core rules (Barthe et al., 2016c)

failure probabilities. Finally, the rule [False] states that a judgment with failure probability at most 1 can prove any post-condition.

A minor but important difference between the setting in aHL and our setting is in the treatment of the failure probability expression β. In aHL, these expressions range over some fixed set of *logical variables*, which appear only in assertions and not in programs. In our setup, we would model these variables as *input variables* V^{in}, which may appear in programs by cannot be modified. We will assume that input variables V^{in} correspond precisely to the logical variables in aHL.

We will show that our proof technique is complete with respect to the logic aHL, subject to two restrictions on aHL proofs:

1. The rule [While] is applied to "for"-loops (this can be slightly generalized to loops with a deterministic variant e_v, but we make this restriction to simplify proofs).
2. The rule [Or] is not used.

Both of these restrictions stem from how our approach keeps track of the failure probability. Roughly speaking, the original aHL can analyze loops where the guard is probabilistic but there is a deterministic bound on the number of iterations. Since our failure probabilities must be deterministic along the trace, we cannot directly handle such loops. However, these programs still have a deterministic bound on the number of iterations and so they can be directly transformed to be of the following form:

$$\textbf{while } b \textbf{ do } P \equiv i \leftarrow 0; \textbf{while } i < \rho \textbf{ do } i \leftarrow i + 1; \textbf{if } b \textbf{ then } P \textbf{ else skip}$$

The situation with the rule [Or] is similar. If we have two well-labeled automata modeling the two proofs in the premise, we would like to combine them into a single automata but this is not possible—the labels on the edges would need to be of the form $\mathsf{assume}(\Phi)$ or $\mathsf{assume}(\Phi')$, but these

guards to not appear in the program P. While it does not appear possible to eliminate the [Or] rule, in our experience this rule is quite rarely used. The rule can also be avoided entirely by applying a program transformation to mark the logical cases:

$$\mathsf{P} \equiv \textbf{if } \Phi \textbf{ then } \mathsf{P} \textbf{ else } \mathsf{P}$$

and then applying the standard conditional rule [If].

We will prove completeness in two steps. First, we will show that for any derivable judgment in aHL, there exists a well-labeled automata modeling the judgment (i.e., satisfying the conditions of Theorem 5.5 for the given pre-condition, post-condition, failure probability, and program). Then, we show that well-labeled automata derived from programs can be found by a run of our algorithm, given some labeling oracle label.

Before we begin, we fix an automata representation of imperative programs once and for all. Each automaton will have one entry node and one exit node. The rest of the nodes, edge labels, and transition structure will be constructed inductively given a program P.

1. Basic statements $\mathit{st} \in \Sigma$. Automaton with single edge from entry to exit node labeled by st.

2. Sequential composition $\mathsf{P} ; \mathsf{P}'$. Identify the exit node for the automaton from P with the entry node for the automaton from P'.

3. Conditionals **if** b **then** P **else** P'. Make new entry node, add directed edges labeled by $\mathsf{assume}(b)$ and $\mathsf{assume}(\neg b)$ to the entry nodes of automata from P and P' respectively, and then identify the exit nodes of the two automata.

4. Loops **while** b **do** P. Make new entry node with an $\mathsf{assume}(b)$ edge to the entry node of the automaton for P, and an $\mathsf{assume}(\neg b)$ edge to a new exit node. From the exit node of P, add an edge back to the new entry node labeled $\mathsf{assume}(b)$ and an edge to the new exit node labeled $\mathsf{assume}(\neg b)$.

We call such automata derived from programs *well-structured*.

Theorem C.5 (Completeness of Well-Labeled Automata)**.** *Let* $\vdash_\beta \{\Phi\}\ P\ \{\Psi\}$ *be derivable in the fragment of* aHL *indicated above. Then there exists a well-structured and well-labeled automaton* A *satisfying the conditions of Theorem 5.5 for this accuracy specification.*

Proof of Theorem C.5. Let A be the well-structured automaton corresponding to P. We will show that the nodes of A can each be labeled by a predicate and a failure probability expression, such that the entire automaton is well-labeled and satisfies the conditions of Theorem 5.5. By induction on the proof derivation.

[Assn] Label the entry and exit nodes by the pre- and post-condition respectively, with failure probability 0.

[Rand] Label the entry and exit nodes by the pre- and post-condition respectively, with failure probability 0 and β.

[Seq] Take the well-labelings for P and P' by induction. Label the node at the join point with invariant Ψ. For each node in the P' automaton, add β to the failure probability label.

[If] Take the well-labelings for P and P' by induction. We may label the entry nodes $\Phi \wedge b$ and $\Phi \wedge \neg b$ while preserving the well-labeling. Label the new entry node by Φ with failure probability 0, and the new exit node by Ψ with failure probability β.

[While] Let ρ be the loop upper bound and let i be the loop counter. Take the well-labeling of the body P by induction. By assumption on the structure of the while loop, there is a single transition from the body entry node q_0 to another node q_1, and it is labeled by $i \leftarrow i + 1$. Furthermore, q_0 and q_1 are both labeled with failure probability 0. Add the deterministic expression $(i - 1) \cdot \beta$ to all failure probability labels except at node q_0, and

set the failure probability of q_0 to be $i \cdot \beta$. Label the new initial node by Φ and failure probability 0, and the new exit node by $\Phi \land \neg b$ and failure probability $\rho \cdot \beta$.

[Weak] Take the well-labeled automaton by induction.

[Frame] Label all nodes by Φ and failure probability 0.

[And] Take the two well-labelings (κ_1, λ_1) and (κ_2, λ_2) by induction. By assumption, both of these well-labeled automata have the same structure (given by the well-structured automaton corresponding to P). Set the new labeling functions to be $\kappa = \kappa_1 \land \kappa_2$, and $\lambda = \lambda_1 + \lambda_2$.

[Or] Not allowed.

[False] Label the entry node by Φ and failure probability 0. Label all other nodes by $\perp$ and failure probability 1.

□

Theorem C.6 (Completeness)**.** *Let A be a well-structured and well-labeled automaton. Then, there exists a run of our algorithm in Figure 5.6 given some labeling oracle label_A that produces A along its execution.*

Proof of Theorem C.6. We provide a sketch of the proof. First, our algorithm can recover any loop-free well-labeled automaton (possibly not well-structured). In a bit more detail, let $\mathcal{L}(A)$ be the set of all paths from entry to exit node; note that this set is finite for loop-free automata. By repeatedly applying trace, our algorithm can label each of these traces using the well-labeling in A, yielding a set of well-labeled traces. Then by repeatedly applying merge, our algorithm can merge all traces and recover the automaton A.

Now, suppose that A is well-structured but not loop-free. We can convert A to a loop-free automaton A_{lf} by simply deleting each back edge

from the exit node of each while loop back to its corresponding entry node; dropping edges evidently keeps the automaton well-labeled. By the previous argument, our algorithm can generate A_{lf} by repeatedly applying TRACE and MERGE. Then, we can apply GENERALIZE repeatedly to add the deleted edges, noting that there are at most finitely many such edges since the originally program has finitely many loops. These new edges preserve well-labeling and recover A. □

As an immediate corollary, we have the following completeness result.

Corollary C.7. *Let* $\vdash_\beta \{\Phi\}\, P\, \{\Psi\}$ *be derivable in the fragment of* AHL *indicated above. Then, there exists a run of our algorithm in Figure 5.6 given some labeling oracle* label_A *terminating successfully with rule* CORRECT*.*

Proof of Corollary C.7. By Theorem C.5, there exists a well-labeled automaton A proving the specification. By Theorem C.6, there is a run of the algorithm that constructs this automaton. At that point in the execution, rule CORRECT applies and the algorithm succeeds. □

C.6 Implementation Details

This appendix expands on Section 5.6 by providing additional implementation details and examples.

Algorithmic Strategy

Our implementation is a determinization of the algorithm presented in Section 5.4. To ensure that we prove the given specifications by computing tight upper bounds on failure probability, our implementation aggressively tries to apply the MERGE rule—recall that the MERGE rule allows us take the maximum failure probability across two automata, instead of the sum. Specifically, we modify the rule TRACE to return a set of traces $\tau_1, \ldots, \tau_n \in$

$\mathcal{L}(P) \cap \overline{\mathcal{L}(\mathcal{A})}$. Then, we attempt to simultaneously label all traces with the same interpolants at nodes pertaining to the same control location. To ensure that we compute similar interpolants across traces, we attempt to use the same distribution axiom for the same sampling instruction in all traces it appears in. Finally, we apply the rule GENERALIZE to attempt to create cycles into the resulting automaton.

The pseudocode in Figure C.3 shows our determinization of the algorithm from Section 5.4. The loop at line 8 goes through axioms as described below, proposing one axiom in every iteration and checking it. Notice that for every occurrence of a sampling statement, across all traces τ_j, it attempts the same axiom—this is used to force a successful MERGE. Line 15 computes interpolants for every trace τ_j's encoding Ψ_j. This procedure also tries to find the same interpolants for the same control locations—this ensures success of MERGE and GENERALIZE. In all case studies in Section 5.6, the algorithm succeeds by considering all traces that execute 0 or 1 iterations of every loop.

Discovering Axioms

Given a formula of the form $\exists f. \forall X. \varphi$, we check its validity using a propose-and-check loop: (*i*) we propose an interpretation of f and then (*ii*) check if $\forall X. \varphi$ is valid with that interpretation using the SMT solver (more on this below). The first step proposes interpretations of f of increasing size, e.g., for a unary function $f(x)$, it would try $0, 1, x, x + 1$, etc.

Note that this enumerative approach will encounter many axiom parameters that are not well-typed or do not satisfy the conditions required for the parameters. For example, for the Laplace axiom family, we have $f(V^{in}) \in (0, 1]$. Therefore, any instantiation that may be real-valued and $\leqslant 0$ or > 1 is rejected.

```
1: 𝒜 ← ∅
2: i ← 1                                                          ▷ counter
3: while CORRECT does not apply do
4:     The following lines implement TRACE for a set of traces
5:     Get all paths τ_1, ..., τ_n ∈ ℒ(()P) \ ℒ(()𝒜) that go through each loop at most i times.
6:     For every τ_j, let Ψ_j be the encoding in Theorem 5.15, where different occurrences of the
   same sampling statement use the same parameter f(V^in) in their distribution axiom.
7:     done ← false
8:     while not done do
9:         pick an interpretation M for every f(V^in) in {Ψ_j}_j
10:        if M ⊨ ⋀_j Ψ_j then
11:            done ← true
12:            axioms ← M
13:        end if
14:    end while
15:    Compute interpolants for every Ψ_j where f(V^in) are instantiated by axioms and create
   well-labeled automata {A_j}_j
16:    Add {A_j}_j to 𝒜
17:    The following repeatedly applies MERGE
18:    Apply MERGE to every pair of automata in 𝒜 until it does not apply any more
19:    The following loop repeatedly applies GENERALIZE
20:    for every A_i ∈ 𝒜 do
21:        for all q, q′ in A_i s.t. q, q′ denote the same loop head in P do
22:            Apply GENERALIZE to q, q′ with ⊭ ∈ Σ being the loop exit condition
23:        end for
24:    end for
25:    i ← i + 1
26: end while
```

Figure C.3: Implementation of nondeterministic algorithm in Figure 5.6

Checking Validity

The case studies we consider make heavy use of non-linear arithmetic (e.g., $\frac{x \cdot y}{z} + c > 0$) and transcendental functions (namely, log). Non-linear theories are generally undecidable. To work around this fact, we implement an incomplete formula validity checker using an eager version of the *theorem enumeration* technique recently proposed by Srikanth et al. (2017). First, we treat non-linear operations as uninterpreted functions, thus overapproximating their semantics. Second, we strengthen formulas by instantiating *theorems* about those non-linear operations. For instance, the following theorem relates division and multiplication: $\forall x, y.\, y > 0 \Rightarrow \frac{x \cdot y}{y} = x$. We then instantiate x and y with terms over variables in the formula. Of course, there are infinitely many possible instantiations of x and y; we

thus restrict instantiations to terms of size 1, i.e., variables or constants.

Our implementation uses a fixed set of theorems about multiplication, division, and log. These are instantiated for every given formula, typically resulting in ~1000 additional conjuncts. To give an intuition, we list some of those theorems below:

1. $\forall x, y.\, y > 0 \Rightarrow \frac{x \cdot y}{y} = x$
2. $\forall x, y, z.\, z > 0 \Rightarrow \frac{x \cdot y}{z} + \frac{x}{z} = \frac{x \cdot (y+1)}{z}$
3. $\forall x, y.\, x \geqslant 0 \wedge y \geqslant 0 \Rightarrow x \cdot y \geqslant 0$
4. $\forall x, y.\, x \geqslant 0 \wedge y > 0 \Rightarrow \frac{x}{y} \geqslant 0$

In all of the differentially private algorithms, we can prove correctness by treating log completely as uninterpreted, requiring no log specific theorems, just the fact that, e.g., $\log(x) + \log(x) = 2\log(x)$.

Interpolation Technique

Given the richness of the theories we use, we found that existing proof-based interpolation techniques either do not support the theories (e.g., the MathSAT solver) or fail to find generalizable interpolants, e.g., cannot discover quantified interpolants (e.g., Z3). As such, we implemented a *template-guided* interpolation technique (Albarghouthi and McMillan, 2013; Rummer and Subotic, 2013), where we force interpolants to follow syntactic forms that appear in the program. Specifically, for every Boolean predicate φ appearing in the program, the specification, or the axioms, we create a template φ^t, which is φ but with variables replaced by wildcards. For instance, given $x > y$, we generate the template $\bullet_1 > \bullet_2$.

Since the failure probabilities, encoded in variables ω_i increase additively by accumulating e^{ub} expressions from the distribution axioms, we

use the template

$$\omega_i \leqslant \sum_{j=1}^{n} \bullet_j * e_j^{ub},$$

where e_j^{ub} is the failure probability of the axiom used in the jth sampling statement, assuming there are n such statements along the path, and $\bullet_j$ can take terms of V^{det}—following the restriction on labels.

Given a set of templates, our interpolation technique searches for an interpolant as a conjunction of instantiations of those templates, where each $\bullet_i$ can be replaced by a well-typed term over formula variables. Given the infinite set of possible instantiations, our implementation fixes the size of possible instantiations (e.g., to size 1), and proceeds by finding the smallest possible interpolants in terms of number of conjuncts. If it cannot, it expands the search to terms of larger sizes. We ensure that the special variables ω_i only appear in their set of inequality predicates defined above. Therefore, given an interpolant I, we can syntactically divide it into $I = I_V \wedge I_\omega$, where I_V is over program variables V and $I_\omega \coloneqq \omega_i \leqslant \dots$ provides the upper bound on failure probabilities at that point along the trace.

Proof of Report Noisy Max (noisyMax)

We give an abridged form of the proof computed for Report Noisy Max in Figure C.4. The set of queries Q is assumed to be non-empty, and for simplicity, we let b be initialized to 1 instead of $\perp$ and modify the conditional to check if $b = 1$—resulting in an equivalent program. The bottom automaton shows a merge of the two paths through the conditional in the loop. Notice that the propagated error probability is

$$\frac{p \cdot (i - 1)}{|Q|}$$

This is because in each iteration, we apply the Laplace axiom with

$$f(V^{in}) = \frac{p}{|Q|}$$

After k loop iterations, $i = k + 1$, and therefore we have accumulated a failure probability of $\frac{p \cdot (i-1)}{|Q|}$. (If the program were rewritten so as i starts at 0 and the loop condition is $i < |Q|$, we would have the simpler failure expression $\frac{p \cdot i}{|Q|}$.) Finally, we can infer that the total failure probability is p. This is due to the state label (blue) φ.

The label φ is defined as the conjunction of the following formulas, which we simplify for presentation:

$$|Q| + 1 \geqslant i \geqslant 1$$

$$i \geqslant b \geqslant 1$$

$$i \neq 1 \Rightarrow b < i$$

$$\forall j \in [1, i).\ |a[j] - Q[j](d)| \leqslant \frac{2}{\epsilon} \log \frac{|Q|}{p}$$

$$\forall j \in [1, i).\ a[b] \geqslant a[j]$$

The first two conjuncts specify the range of values i takes throughout the loop iterations. The third conjunct specifies that i leaps ahead of b after the first loop iteration, since i is always incremented at the end of the loop, and b can at most be $i - 1$ at that point. (The syntactic form of an implication is derived from the conditional's predicate.) The fourth conjunct specifies that, for every element of j of a, its distance from the corresponding valuation of $Q[j](d)$ is bounded above by $\frac{2}{\epsilon} \log \frac{|Q|}{p}$, which follows from the choice of the axiom. Finally, the last conjunct states that the best element is indeed larger than all previously seen ones.

The last two conjuncts are primarily responsible for implying the post-

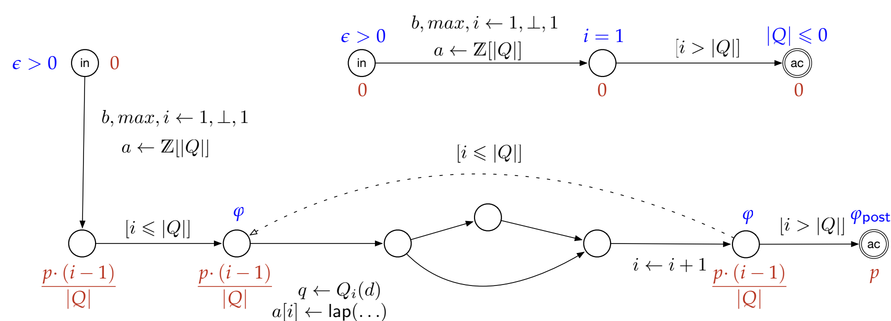


Figure C.4: Main loop of verification algorithm

condition (via the triangle inequality):

$$\forall j \in [1, |Q|].\, Q[b](d) \geqslant Q[j](d) - \frac{4}{\epsilon} \log \frac{|Q|}{p}$$

Notice that the $\frac{2}{\epsilon} \log \frac{|Q|}{p}$ in the fourth conjunct translates to $\frac{4}{\epsilon} \log \frac{|Q|}{p}$ in the postcondition. This is due to the absolute value.

D.1 Definition of KBO

We provide the standard construction of the Knuth-Bendix order.

Definition D.1 (Knuth-Bendix Order (Zankl et al., 2009)). *Let $\succ$ be a total order on C, and $\omega : T_{\Sigma^C}(X) \to \mathbb{N}$ a linear (with respect to contexts and substitutions) function on programs such that $\omega(X) = \omega_0$ (for some constant ω_0) and $\omega(c) \geqslant \omega_0$ for all constants $c \in C$. Then $>$ is the* KBO *parameterized by $\succ$ and ω if, for all $s, t \in T_{\Sigma^C}(X)$:*

$s > t$ if and only if $|s|_x \geqslant |t|_x$ for all $x \in X$ and

- *$\omega(s) > \omega(t)$, or*
- *$\omega(s) = \omega(t)$ and one of the following:*
  - *t is a variable, $s \neq t$, and $s \in T_{\Sigma^U}(\{t\})$*
  - *$s = f(s_1, \ldots, s_n)$ and $t = g(t_1, \ldots, t_n)$ and $f \succ g$*
  - *$s = f(s_1, \ldots, s_n)$ and $t = g(t_1, \ldots, t_n)$, $f = g$, and $s_1, \ldots, s_n >^{lex} t_1, \ldots, t_n$.*

where $U = \{f \in C \mid ar(f) = 1\}$, $>^{lex}$ is the lexicographic lifting of $>$, and $|s|_x$ is the number of x's in s.

D.2 Components and Equations

In Tables D.1 to D.3, we provide the full set of components and equations used in the evaluation in Section 6.7 and describe the benchmarks in Table 6.3.

Benchmark	Description
Integers	
`add`	Integer addition
`max`	Integer maximum from comparisons
`min`	Integer minimum from comparisons
Tuples and integers	
`add-4`	Add 4 integers together
`mult-q`	Multiplication in $\mathbb{Q}$, with rationals as pairs
`div-q`	Division in $\mathbb{Q}$, with rationals as pairs
`add-c`	Addition in $\mathbb{C}$, with complex numbers as pairs
`sub-c`	Subtraction in $\mathbb{C}$, with complex numbers as pairs
`add-q-long`	Addition in $\mathbb{Q}$, with two rationals as four integers
`max-pair`	Element-wise maximum
`intervals`	Join in intervals domain over $\mathbb{N}$
`min-pair`	Element-wise minimum
`sum-to-first`	Sum all pair elements to s, return $(s, 0)$
`sum-to-second`	Sum all pair elements to s, return $(0, s)$
`add-and-mult`	Add first pair elements, multiply second
`distances`	Element-wise distance between integers
Strings and integers	
`str-len`	Return the longest string
`str-trim-len`	Return the longest trimmed string
`str-upper-len`	Return the longest string, cast to uppercase
`str-lower-len`	Return the longest string, cast to lowercase
`str-add`	Treat strings as integers, returning sum as a string
`str-mult`	Treat strings as integers, returning product as a string
`str-max`	Treat strings as integers, returning maximum as a string
`str-split`	Return string with the most words (whitespace delimited)
Lists and integers	
`ls-sum`	Add sums of two lists, returning as a singleton list
`ls-sum2`	Add two plus the sums of two lists, returning as a singleton
`ls-sum-abs`	Add the absolute value of the sum of two lists
`ls-min`	Compute minimum element of two lists, returning as a list
`ls-max`	Compute maximum element of two lists, returning as a list
`ls-stutter`	Compute sum s of two lists, returning as a list with s copies of s

Table D.1: Descriptions of benchmarks from Table 6.3

Component	Type	Description
Integer		
`add`	`int` → `int` → `int`	integer addition
`mult`	`int` → `int` → `int`	integer multiplication
`sub`	`int` → `int` → `int`	integer subtraction
`abs`	`int` → `int`	absolute value
`succ`	`int` → `int`	successor function (increment)
`zero`	`int`	constant 0
`one`	`int`	constant 1
`=`	`int` → `int` → `bool`	integer equality
`!=`	`int` → `int` → `bool`	integer inequality
`>`	`int` → `int` → `bool`	greater-than
`>=`	`int` → `int` → `bool`	greater-than or equal
`<`	`int` → `int` → `bool`	less-than
`<=`	`int` → `int` → `bool`	less-than or equal
Boolean		
`and`	`bool` → `bool` → `bool`	conjunction
`or`	`bool` → `bool` → `bool`	disjunction
`not`	`bool` → `bool`	negation
`ite`	`bool` → $\alpha \to \alpha$	conditional branching
String		
`concat`	`string` → `string` → `string`	string concatenation
`length`	`string` → `int`	length of string
`uppercase`	`string` → `string`	convert string to uppercase
`lowercase`	`string` → `string`	convert string to lowercase
`trim`	`string` → `string`	remove leading and trailing whitespace
`reverse`	`string` → `string`	reverse a string
`toInt`	`string` → `int`	convert string to integer
`fromInt`	`int` → `string`	convert integer to string
`split`	`string` → `string` → [`string`]	split string on delimiter
`join`	[`string`] → `string` → `string`	join list of words with delimiter
Pairs		
`pair`	`int` → `int` → $\texttt{int}^2$	construct tuple of integers
`fst`	$\texttt{int}^2$ → `int`	return first element of a pair
`snd`	$\texttt{int}^2$ → `int`	return second element of a pair
List		
`maxElement`	[`int`] → `int`	returns maximal element of list
`minElement`	[`int`] → `int`	returns minimal element of list
`sum`	[`int`] → `int`	adds list of integers
`cons`	`int` → [`int`] → `int`	adds integer to head of list
`cat`	[`int`] → [`int`] → [`int`]	concatenates two lists together
`singleton`	`int` → [`int`]	singleton list constructor
`length`	[`int`] → `int`	length of a list
`stutter`	`int` → `int` → [`int`]	`stutter`(n, m) returns a list with m n's

Table D.2: Synthesis domain used in used in Section 6.7

Equation	Description
Integer	
`abs(abs(x)) = x`	idempotence of `abs`
`abs(0) = 0`	application of `abs` to a constant
`abs(1) = 1`	application of `abs` to a constant
`add(x,y) = add(y,x)`	commutativity of `add`
`add(x,0) = x`	identity of `add`
`add(x,1) = succ(x)`	definition of `succ`
`add(x,add(y,z)) = add(add(x,y),z)`	associativity of `add`
`mult(x,0) = 0`	annihilation by 0
`mult(x,1) = x`	identity of `mult`
`mult(x,y) = mult(y,x)`	commutativity of `mult`
`mult(x,mul(y,z)) = mult(mult(x,y),z)`	associativity of `mult`
`mult(x,add(y,z)) = add(mult(x,y),mult(x,z))`	distributivity of `add`
`sub(x,0) = x`	right-identity of `sub`
`sub(x,x) = 0`	self-annihilation of `sub`
Comparisons	
`eq(x,y) = eq(y,x)`	symmetry of equality
`neq(x,y) = neq(y,x)`	symmetry of inequality
`leq(x,y) = eq(x,y) ∧ lt(x,y)`	definition of `<=`
`geq(x,y) = eq(x,y) ∧ gt(x,y)`	definition of `>=`
`lt(x,y) = not(geq(x,y))`	alternate definition of `>=`
`gt(x,y) = not(leq(x,y))`	alternate definition of `<=`
`eq(x,y) = not(neq(x,y))`	alternate definition of `!=`
Boolean	
`not(not(x)) = x`	involutiveness of `not`
`and(x,y) = and(y,x)`	commutativity of `and`
`and(x,and(y,z)) = and(and(x,y),z)`	associativity of `and`
`or(x,y) = or(y,x)`	commutativity of `or`
`or(x,or(y,z)) = or(or(x,y),z)`	associativity of `or`
`and(true,x) = x`	identity of `and`
`and(false,x) = false`	annihilation of `and`
`or(true,x) = true`	annihilation of `or`
`or(false,x) = x`	identity of `or`
`or(and(x,y),and(x,z)) = and(x,or(y,z))`	distributivity of `or`
`not(and(x,y)) = or(not(x),not(y))`	DeMorgan's law for `and`
`not(or(x,y)) = and(not(x),not(y))`	DeMorgan's law for `or`
`ite(b,x,x) = x`	equivalence of branching
`ite(true,x,y) = x`	taking the `if-then` branch
`ite(false,x,y) = y`	taking the `else` branch
Pairs	
`fst(pair(x,y)) = x`	definition of `fst`
`snd(pair(x,y)) = y`	definition of `snd`
String	
`uppercase(uppercase(x)) = uppercase(x)`	idempotence of `uppercase`
`lowercase(lowercase(x)) = lowercase(x)`	idempotence of `lowercase`
`trim(trim(x)) = trim(x)`	idempotence of `trim`
`reverse(reverse(x)) = x`	involutiveness of `reverse`
`join(split(x,y),y) = x`	partial inverses
`toInt(fromInt(x)) = x`	partial inverses
`add(length(x),length(y)) = length(cat(x,y))`	`length` distribution
`cat(uppercase(x),uppercase(y)) = uppercase(cat(x,y))`	`uppercase` distribution
`cat(lowercase(x),lowercase(y)) = lowercase(cat(x,y))`	`lowercase` distribution
List	
`add(sum(x),sum(y)) = sum(cat(x,y))`	`sum` distribution
`add(length(x),length(y)) = length(cat(x,y))`	`length` distribution
`cat(singleton(x),y) = cons(x,y)`	definition of `cons`

Table D.3: All equations used in Section 6.7

DISCARD THIS PAGE

COLOPHON

The template for this document was provided by Will B. and modified for minor stylistic preferences.

Improving spark application performance. chapeau.freevariable.com/2014/09/improving-spark-application-performance.html.

Abowd, John M., and Ian M. Schmutte. 2017. Revisiting the economics of privacy: Population statistics and confidentiality protection as public goods. Tech. Rep. 17–37, Center for Economic Studies.

Alarcón, Beatriz, Raúl Gutiérrez, Salvador Lucas, and Rafael Navarro-Marset. 2010. Proving termination properties with mu-term. In *International conference on algebraic methodology and software technology*, 201–208. Springer.

Albarghouthi, Aws. 2017. Probabilistic horn clause verification. In *International symposium on static analysis (sas), new york, new york*, 1–22.

Albarghouthi, Aws, Sumit Gulwani, and Zachary Kincaid. 2013. Recursive program synthesis. In *Cav*.

Albarghouthi, Aws, and Justin Hsu. 2018a. Constraint-based synthesis of coupling proofs. In *International conference on computer aided verification (cav), oxford, england*.

———. 2018b. Synthesizing coupling proofs of differential privacy. *PACMPL* 2(POPL):58:1–58:30.

Albarghouthi, Aws, and Kenneth L. McMillan. 2013. Beautiful interpolants. In *International conference on computer aided verification (cav), saint petersburg, russia*, 313–329. Springer.

Alsubaiee, Sattam, Yasser Altowim, Hotham Altwaijry, Alexander Behm, Vinayak R. Borkar, Yingyi Bu, Michael J. Carey, Inci Cetindil, Madhusudan Cheelangi, Khurram Faraaz, Eugenia Gabrielova, Raman Grover,

Zachary Heilbron, Young-Seok Kim, Chen Li, Guangqiang Li, Ji Mahn Ok, Nicola Onose, Pouria Pirzadeh, Vassilis J. Tsotras, Rares Vernica, Jian Wen, and Till Westmann. 2014. Asterixdb: A scalable, open source BDMS. *PVLDB* (14).

Alur, Rajeev, Rastislav Bodík, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. 2013. Syntax-guided synthesis. In *Fmcad*.

Alur, Rajeev, Pavol Černỳ, and Arjun Radhakrishna. 2015. Synthesis through unification. In *International conference on computer aided verification*, 163–179. Springer.

de Amorim, Arthur Azevedo, Marco Gaboardi, Emilio Jesús Gallego Arias, and Justin Hsu. 2014. Really natural linear indexed type checking. In *Proceedings of the 26th 2014 international symposium on implementation and application of functional languages, IFL '14, boston, ma, usa, october 1-3, 2014*, 5:1–5:12.

de Amorim, Arthur Azevedo, Marco Gaboardi, Justin Hsu, Shin-ya Katsumata, and Ikram Cherigui. 2017. A semantic account of metric preservation. In *Proceedings of the 44th acm sigplan symposium on principles of programming languages*, 545–556. POPL 2017.

Angus, A., and D. Kozen. 2001. Kleene algebra with tests and program schematology. Tech. Rep. 2001-1844, Cornell University.

Apple. Accessed 11-11-2017. Differential privacy. https://images.apple.com/privacy/docs/Differential_Privacy_Overview.pdf.

Baader, Franz, and Tobias Nipkow. 1998. *Term rewriting and all that*. Cambridge University Press.

Bachmair, Leo, Nachum Dershowitz, and David A Plaisted. 1989. Completion without failure. *Resolution of equations in algebraic structures* 2: 1–30.

Baier, Christel, Luca de Alfaro, Vojtech Forejt, and Marta Kwiatkowska. 2018. Model checking probabilistic systems. In *Handbook of model checking*, 963–999. Springer-Verlag.

Barowy, Daniel W., Sumit Gulwani, Ted Hart, and Benjamin G. Zorn. 2015. Flashrelate: extracting relational data from semi-structured spreadsheets using examples. In *Pldi*.

Barthe, Gilles, Juan Manuel Crespo, Sumit Gulwani, César Kunz, and Mark Marron. 2013. From relational verification to SIMD loop synthesis. In *Ppopp*.

Barthe, Gilles, Juan Manuel Crespo, and César Kunz. 2011. Relational verification using product programs. In *Fm*.

Barthe, Gilles, Pedro R. D'Argenio, and Tamara Rezk. 2004. Secure information flow by self-composition. In *Csfw*.

Barthe, Gilles, Thomas Espitau, Luis María Ferrer Fioriti, and Justin Hsu. 2016a. Synthesizing probabilistic invariants via Doob's decomposition. In *International conference on computer aided verification (cav), toronto, ontario*, vol. 9779 of *Lecture Notes in Computer Science*, 43–61. Springer-Verlag. 1605.02765.

Barthe, Gilles, Thomas Espitau, Marco Gaboardi, Benjamin Grégoire, Justin Hsu, and Pierre-Yves Strub. 2018. A program logic for probabilistic programs. In *European symposium on programming (esop), thessaloniki, greece*. To appear.

Barthe, Gilles, Marco Gaboardi, Emilio Jesús Gallego Arias, Justin Hsu, César Kunz, and Pierre-Yves Strub. 2014. Proving differential privacy in hoare logic. In *Csf*.

Barthe, Gilles, Marco Gaboardi, Benjamin Grégoire, Justin Hsu, and Pierre-Yves Strub. 2016b. A program logic for union bounds. In *The 43rd International Colloquium on Automata, Languages and Programming* . Rome, Italy.

Barthe, Gilles, Marco Gaboardi, Benjamin Grégoire, Justin Hsu, and Pierre-Yves Strub. 2016c. A program logic for union bounds. In *International colloquium on automata, languages and programming (icalp), rome, italy*, 107:1–107:15.

Barthe, Gilles, Boris Köpf, Federico Olmedo, and Santiago Zanella Béguelin. 2012. Probabilistic relational reasoning for differential privacy. In *Popl*.

Belle, Vaishak, Andrea Passerini, and Guy Van den Broeck. 2015. Probabilistic inference in hybrid domains by weighted model integration. In *International joint conference on artificial intelligence (ijcai), buenos aires, argentina*, 2770–2776.

Benton, Nick. 2004. Simple relational correctness proofs for static analyses and program transformations. In *Popl*.

Blei, David M., Andrew Y. Ng, and Michael I. Jordan. 2003. Latent dirichlet allocation. *JMLR* 3:993–1022.

Bornholt, James, Emina Torlak, Dan Grossman, and Luis Ceze. 2016. Optimizing synthesis with metasketches. In *Proceedings of the 43rd annual acm sigplan-sigact symposium on principles of programming languages*, 775–788. POPL '16, New York, NY, USA: Association for Computing Machinery.

Boykin, P. Oscar, Sam Ritchie, Ian O'Connell, and Jimmy Lin. 2014. Summingbird: A framework for integrating batch and online mapreduce computations. *PVLDB* 7(13):1441–1451.

Bureau, US Census. Accessed 11-11-2017. On the map. https://onthemap.ces.census.gov/.

Carbin, Michael, Deokhwan Kim, Sasa Misailovic, and Martin C. Rinard. 2012. Proving acceptability properties of relaxed nondeterministic approximate programs. In *Pldi*.

Carbin, Michael, Sasa Misailovic, and Martin C Rinard. 2013. Verifying quantitative reliability for programs that execute on unreliable hardware. In *Acm sigplan notices*, vol. 48, 33–52. ACM.

Chadha, Rohit, Luís Cruz-Filipe, Paulo Mateus, and Amílcar Sernadas. 2007. Reasoning about probabilistic sequential programs. *Theoretical Computer Science* 379(1):142–165.

Chakarov, Aleksandar, and Sriram Sankaranarayanan. 2013. Probabilistic program analysis with martingales. In *International conference on computer aided verification (cav), saint petersburg, russia*, 511–526.

Chan, T.-H. Hubert, Elaine Shi, and Dawn Song. 2011. Private and continual release of statistics. *ACM Transactions on Information and System Security* 14(3):26.

Chatterjee, Krishnendu, Hongfei Fu, and Amir Kafshdar Goharshady. 2016a. Termination analysis of probabilistic programs through Positivstellensatz's. In *International conference on computer aided verification (cav), toronto, ontario*, vol. 9779 of *Lecture Notes in Computer Science*, 3–22. Springer-Verlag.

Chatterjee, Krishnendu, Hongfei Fu, Petr Novotný, and Rouzbeh Hasheminezhad. 2016b. Algorithmic analysis of qualitative and quantitative termination problems for affine probabilistic programs. In *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), saint petersburg, florida*, 327–342.

Chen, Yu-Fang, Chih-Duo Hong, Nishant Sinha, and Bow-Yaw Wang. 2015. Commutativity of reducers. In *Tacas*.

Chistikov, Dmitry, Rayna Dimitrova, and Rupak Majumdar. 2015. Approximate counting in SMT and value estimation for probabilistic programs. In *International conference on tools and algorithms for the construction and analysis of systems (tacas), london, england*, 320–334.

Cimatti, Alessandro, Alberto Griggio, Bastiaan Joost Schaafsma, and Roberto Sebastiani. 2013. The MathSAT5 SMT solver. In *International conference on tools and algorithms for the construction and analysis of systems (tacas), rome, italy*, 93–107. Springer.

Claessen, Koen, Nicholas Smallbone, and John Hughes. 2010. Quickspec: Guessing formal specifications using testing. In *International conference on tests and proofs*, 6–21. Springer.

Claret, Guillaume, Sriram K. Rajamani, Aditya V. Nori, Andrew D. Gordon, and Johannes Borgström. 2013. Bayesian inference using data flow analysis. In *Joint meeting of the european software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering (esec/fse), saint petersburg, russia*, 92–102.

Clarkson, Michael R., and Fred B. Schneider. 2010. Hyperproperties. *JCS* (6).

Cloud, Google. Google cloud. cloud.google.com.

Cohen, William W. Enron emails dataset. cs.cmu.edu/~enron/.

Comon, Hubert, and Florent Jacquemard. 1997. Ground reducibility is exptime-complete. In *Logic in computer science, 1997. lics'97. proceedings., 12th annual ieee symposium on*, 26–34. IEEE.

Cousot, Patrick, and Radhia Cousot. 1977. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th acm sigact-sigplan symposium on principles of programming languages*, 238–252. ACM.

Cousot, Patrick, and Michael Monerau. 2012. Probabilistic abstract interpretation. In *European symposium on programming (esop), tallinn, estonia*, 169–193. Springer-Verlag.

Cusumano-Towner, Marco, Benjamin Bichsel, Timon Gehr, Martin Vechev, and Vikash K Mansinghka. 2018. Incremental inference for probabilistic programs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), philadelphia, pennsylvania*, 571–585.

Datta, Anupam, Matthew Fredrikson, Gihyuk Ko, Piotr Mardziel, and Shayak Sen. 2017. Use privacy in data-driven systems: Theory and experiments with machine learnt programs. In *Proceedings of the 2017 acm sigsac conference on computer and communications security*, 1193–1210. CCS '17, New York, NY, USA: ACM.

De Bruijn, Nicolaas Govert. 1972. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. In *Indagationes mathematicae (proceedings)*, vol. 75, 381–392. Elsevier.

Dean, Jeffrey, and Sanjay Ghemawat. 2004. Mapreduce: Simplified data processing on large clusters. In *Osdi*.

Dechter, Eyal, Jonathan Malmaud, Ryan P Adams, and Joshua B Tenenbaum. 2013. Bootstrap learning via modular concept discovery. In *Ijcai*.

Dehnert, Christian, Sebastian Junges, Joost-Pieter Katoen, and Matthias Volk. 2017. A storm is coming: A modern probabilistic model checker. In *International conference on computer aided verification (cav), heidelberg, germany*, vol. abs/1702.04311 of *Lecture Notes in Computer Science*. Springer-Verlag. 1702.04311.

Dershowitz, Nachum. 1985. Synthesis by completion. *Urbana* 51:61801.

Dwork, Cynthia, Moritz Hardt, Toniann Pitassi, Omer Reingold, and Richard Zemel. 2012. Fairness through awareness. In *ACM SIGACT Innovations in Theoretical Computer Science (itcs), cambridge, massachusetts*, 214–226.

Dwork, Cynthia, Frank McSherry, Kobbi Nissim, and Adam D. Smith. 2006. Calibrating noise to sensitivity in private data analysis. In *IACR Theory of Cryptography Conference (tcc), new york, new york*, vol. 3876 of *Lecture Notes in Computer Science*, 265–284. Springer-Verlag.

Dwork, Cynthia, Moni Naor, Toniann Pitassi, and Guy N. Rothblum. 2010. Differential privacy under continual observation. In *ACM SIGACT Symposium on Theory of Computing (stoc), cambridge, massachusetts*, 715–724.

Dwork, Cynthia, and Aaron Roth. 2014. The algorithmic foundations of differential privacy. *Foundations and Trends® in Theoretical Computer Science* 9(3–4):211–407.

Erlingsson, Ulfar, Vasyl Pihur, and Aleksandra Korolova. 2014. Rappor: Randomized aggregatable privacy-preserving ordinal response. In *Proceedings of the 21st acm conference on computer and communications security*. Scottsdale, Arizona.

Farzan, Azadeh, Zachary Kincaid, and Andreas Podelski. 2013. Inductive data flow graphs. In *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), rome, italy*, 129–142.

Feldman, Michael, Sorelle A. Friedler, John Moeller, Carlos Scheidegger, and Suresh Venkatasubramanian. 2015. Certifying and removing disparate impact. In *Proceedings of the 21th acm sigkdd international conference on knowledge discovery and data mining*, 259–268. KDD '15, New York, NY, USA: ACM.

Feng, Yu, Ruben Martins, Jacob Van Geffen, Isil Dillig, and Swarat Chaudhuri. 2017. Component-based synthesis of table consolidation and transformation tasks from examples. In *Proceedings of the 38th acm sigplan conference on programming language design and implementation*, 422–436. ACM.

Feser, John K., Swarat Chaudhuri, and Isil Dillig. 2015. Synthesizing data structure transformations from input-output examples. In *Pldi*.

Flink. Apache flink. `flink.apache.org`.

Frankle, Jonathan, Peter-Michael Osera, David Walker, and Steve Zdancewic. 2016. Example-directed synthesis: A type-theoretic interpretation. In *Popl*.

Freeman, Tim, and Frank Pfenning. 1991. Refinement types for ML. In *Pldi*, ed. David S. Wise.

Gaboardi, Marco, Andreas Haeberlen, Justin Hsu, Arjun Narayan, and Benjamin C. Pierce. 2013. Linear dependent types for differential privacy. In *The 40th annual ACM SIGPLAN-SIGACT symposium on principles of programming languages, POPL '13, rome, italy - january 23 - 25, 2013*, 357–370.

Gehr, Timon, Sasa Misailovic, Petar Tsankov, Laurent Vanbever, Pascal Wiesmann, and Martin Vechev. 2018. Bayonet: probabilistic inference for

networks. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), philadelphia, pennsylvania*, 586–602.

Gehr, Timon, Sasa Misailovic, and Martin Vechev. 2016. PSI: Exact symbolic inference for probabilistic programs. In *International conference on computer aided verification (cav), toronto, ontario*, 62–83. Springer-Verlag.

Giesl, Jürgen, Peter Schneider-Kamp, and René Thiemann. 2006. Aprove 1.2: Automatic termination proofs in the dependency pair framework. In *International joint conference on automated reasoning*, 281–286. Springer.

Graf, Susanne, and Hassen Saïdi. 1997. Construction of abstract state graphs with PVS. In *International conference on computer aided verification (cav), haifa, israel*, 72–83. Springer-Verlag.

Gulwani, Sumit. 2011. Automating string processing in spreadsheets using input-output examples. In *Popl*.

Gulwani, Sumit, William R. Harris, and Rishabh Singh. 2012. Spreadsheet data manipulation using examples. *CACM* (8).

Gulwani, Sumit, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. 2011. Synthesis of loop-free programs. In *Pldi*.

Gupta, Anupam, Aaron Roth, and Jonathan Ullman. 2012. Iterative constructions and private data release. In *Theory of cryptography conference*, 339–356. Springer.

Gvero, Tihomir, Viktor Kuncak, Ivan Kuraj, and Ruzica Piskac. 2013. Complete completion using types and weights. In *Pldi*.

Halperin, Daniel, Victor Teixeira de Almeida, Lee Lee Choo, Shumo Chu, Paraschos Koutris, Dominik Moritz, Jennifer Ortiz, Vaspol Ruamviboonsuk, Jingjing Wang, Andrew Whitaker, Shengliang Xu, Magdalena Balazinska, Bill Howe, and Dan Suciu. 2014. Demonstration of the myria

big data management service. In *Sigmod*, ed. Curtis E. Dyreson, Feifei Li, and M. Tamer Özsu.

Haney, Samuel, Ashwin Machanavajjhala, John M Abowd, Matthew Graham, Mark Kutzbach, and Lars Vilhuber. 2017. Utility cost of formal privacy for releasing national employer-employee statistics. In *Proceedings of the 2017 acm international conference on management of data*, 1339–1354. ACM.

Hardt, Moritz, Katrina Ligett, and Frank Mcsherry. 2012. A simple and practical algorithm for differentially private data release. In *Advances in neural information processing systems 25*, ed. F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, 2339–2347. Curran Associates, Inc.

Harris, William R., and Sumit Gulwani. 2011. Spreadsheet table transformations from examples. In *Pldi*.

den Hartog, J. 2002. Probabilistic extensions of semantical models. Ph.D. thesis, Vrije Universiteit Amsterdam.

Heizmann, Matthias, Jochen Hoenicke, and Andreas Podelski. 2009. Refinement of trace abstraction. In *International symposium on static analysis (sas), los angeles, california*, 69–85. Springer-Verlag.

———. 2010. Nested interpolants. In *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), madrid, spain*, 471–482.

———. 2013. Software model checking for people who love automata. In *International conference on computer aided verification (cav), saint petersburg, russia*, 36–52. Springer-Verlag.

Henzinger, Thomas A., Ranjit Jhala, Rupak Majumdar, and Kenneth L. McMillan. 2004. Abstractions from proofs. In *ACM SIGPLAN–SIGACT*

Symposium on Principles of Programming Languages (POPL), venice, italy, vol. 39, 232–244.

Hermanns, Holger, Björn Wachter, and Lijun Zhang. 2008. Probabilistic CEGAR. In *International conference on computer aided verification (cav), princeton, new jersey*, 162–175. Springer-Verlag.

Hillenbrand, Thomas, Arnim Buch, Roland Vogt, and Bernd Löchner. 1997. Waldmeister-high-performance equational deduction. *Journal of Automated Reasoning* 18(2):265–270.

Hsu, Justin. 2017. Probabilistic couplings for probabilistic reasoning. Ph.D. thesis, University of Pennsylvania. 1710.09951.

Hu, Qinheping, and Loris D'Antoni. 2018. Syntax-guided synthesis with quantitative syntactic objectives. In *Computer aided verification*, ed. Hana Chockler and Georg Weissenbacher, 386–403. Cham: Springer International Publishing.

Jha, Susmit, Sumit Gulwani, Sanjit A. Seshia, and Ashish Tiwari. 2010. Oracle-guided component-based program synthesis. In *Icse*.

Johnson, Noah, Joseph P Near, and Dawn Song. 2018a. Towards practical differential privacy for SQL queries. *Proceedings of the VLDB Endowment* 11(5):526–539. Appeared at the International Conference on Very Large Data Bases (VLDB), Rio de Janeiro, Brazil.

Johnson, Noah M., Joseph P. Near, and Dawn Xiaodong Song. 2018b. Practical differential privacy for SQL queries using elastic sensitivity.

Kandel, Sean, Andreas Paepcke, Joseph M. Hellerstein, and Jeffrey Heer. 2011. Wrangler: interactive visual specification of data transformation scripts. In *Chi*, ed. Desney S. Tan, Saleema Amershi, Bo Begole, Wendy A. Kellogg, and Manas Tungare, 3363–3372. ACM.

Katoen, Joost-Pieter. 2016. The probabilistic model checking landscape. In *IEEE Symposium on Logic in Computer Science (LICS), new york, new york*, 31–45.

Kattenbelt, Mark, Marta Kwiatkowska, Gethin Norman, and David Parker. 2009. Abstraction refinement for probabilistic software. In *International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI), savannah, georgia*, 182–197. Springer-Verlag.

———. 2010. A game-based abstraction-refinement framework for Markov decision processes. *Formal Methods in System Design* 36(3):246–280.

Kitzelmann, Emanuel, and Ute Schmid. 2006. Inductive synthesis of functional programs: An explanation based generalization approach. *JMLR*.

Klein, Dominik, and Nao Hirokawa. 2011. Maximal completion. In *Lipics-leibniz international proceedings in informatics*, vol. 10. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

Kneuss, Etienne, Ivan Kuraj, Viktor Kuncak, and Philippe Suter. 2013. Synthesis modulo recursive functions. In *Oopsla*.

Knuth, Donald E, and Peter B Bendix. 1983. Simple word problems in universal algebras. In *Automation of reasoning*, 342–376. Springer.

Korp, Martin, Christian Sternagel, Harald Zankl, and Aart Middeldorp. 2009. Tyrolean termination tool 2. In *International conference on rewriting techniques and applications*, 295–304. Springer.

Kozen, Dexter. 1985. A probabilistic PDL. *Journal of Computer and System Sciences* 30(2).

Kuncak, Viktor, Mikaël Mayer, Ruzica Piskac, and Philippe Suter. 2010. Complete functional synthesis. In *Pldi*.

Kurihara, Masahito, and Hisashi Kondo. 1999. Completion for multiple reduction orderings. *Journal of Automated Reasoning* 23(1):25–42.

Kwiatkowska, Marta, Gethin Norman, and David Parker. 2010. Advances and challenges of probabilistic model checking. In *Annual Allerton conference on communication, control, and computing (allerton)*, 1691–1698. IEEE.

———. 2011. PRISM 4.0: Verification of probabilistic real-time systems. In *International conference on computer aided verification (cav), snowbird, utah*, vol. 6806 of *Lecture Notes in Computer Science*, 585–591. Springer-Verlag.

Larson, Jeff, Surya Mattu amd Lauren Kirchner, and Julia Angwin. 2016. How we analyzed the compas recidivism algorithm. Accessed: 2017-11-15.

Le, Vu, and Sumit Gulwani. 2014. Flashextract: a framework for data extraction by examples. In *Pldi*.

Leskovec, Jure, Anand Rajaraman, and Jeffrey D. Ullman. 2014. *Mining of massive datasets, 2nd ed*. Cambridge University Press.

Liang, Percy, Michael I Jordan, and Dan Klein. 2010. Learning programs: A hierarchical bayesian approach. In *Proceedings of the 27th international conference on machine learning (icml-10)*, 639–646.

Lichman, M. 2013. UCI machine learning repository.

Littman, Michael L., Stephen M Majercik, and Toniann Pitassi. 2001. Stochastic boolean satisfiability. *Journal of Automated Reasoning* 27(3):251–296.

Löchner, Bernd. 2006. Things to know when implementing kbo. *Journal of Automated Reasoning* 36(4):289–310.

Löchner, Bernd. 2006. Things to know when implementing lpo. *International Journal on Artificial Intelligence Tools* 15(01):53–79. http://www.worldscientific.com/doi/pdf/10.1142/S0218213006002564.

Lyu, Min, Dong Su, and Ninghui Li. 2016. Understanding the sparse vector technique for differential privacy. *CoRR* abs/1603.01699. 1603.01699.

Mardziel, Piotr, Stephen Magill, Michael Hicks, and Mudhakar Srivatsa. 2011. Dynamic enforcement of knowledge-based security policies. In *IEEE Computer Security Foundations Symposium (CSF), domaine de l'abbaye des vaux de cernay, france*, 114–128.

McCune, William. 1992. Experiments with discrimination-tree indexing and path indexing for term retrieval. *J. Autom. Reason.* 9(2):147–167.

McIver, Annabelle, Carroll Morgan, Benjamin Lucien Kaminski, and Joost-Pieter Katoen. 2018. A new proof rule for almost-sure termination. *Proceedings of the ACM on Programming Languages* 2(POPL):33:1–33:28.

McMillan, Kenneth L. 2003. Interpolation and SAT-based model checking. In *International conference on computer aided verification (cav), boulder, colorado*, 1–13. Springer-Verlag.

———. 2006. Lazy abstraction with interpolants. In *International conference on computer aided verification (cav), seattle, washington*, 123–136. Springer-Verlag.

McSherry, Frank, and Kunal Talwar. 2007. Mechanism design via differential privacy. In *Foundations of computer science, 2007. focs'07. 48th annual ieee symposium on*, 94–103. IEEE.

McSherry, Frank, and Kunal Talwar. 2007. Mechanism design via differential privacy. In *IEEE Symposium on Foundations of Computer Science (focs), providence, rhode island*, 94–103.

McSherry, Frank D. 2009. Privacy integrated queries: an extensible platform for privacy-preserving data analysis. In *Proceedings of the 2009 acm sigmod international conference on management of data*, 19–30. ACM.

Menon, Aditya Krishna, Omer Tamuz, Sumit Gulwani, Butler W. Lampson, and Adam Kalai. 2013. A machine learning framework for programming by example. In *Icml*.

Miltner, Anders, Kathleen Fisher, Benjamin C. Pierce, David Walker, and Steve Zdancewic. 2018. Synthesizing bijective lenses. *PACMPL* 2(POPL): 1:1–1:30.

Miner, Donald, and Adam Shook. 2012. *Mapreduce design patterns: Building effective algorithms and analytics for hadoop and other systems*. 1st ed. O'Reilly.

Monniaux, David. 2000. Abstract interpretation of probabilistic semantics. In *International symposium on static analysis (sas), santa barbara, california*, 322–339. Springer-Verlag.

———. 2001. Backwards abstract interpretation of probabilistic programs. In *European symposium on programming (esop), genova, italy*, 367–382. Springer-Verlag.

———. 2005. Abstract interpretation of programs as markov decision processes. *Science of Computer Programming* 58(1):179–205.

Morgan, Carroll, Annabelle McIver, and Karen Seidel. 1996. Probabilistic predicate transformers. *ACM Transactions on Programming Languages and Systems* 18(3):325–353.

de Moura, Leonardo Mendonça, and Nikolaj Bjørner. 2008. Z3: an efficient SMT solver. In *Tacas*.

Narayan, Arjun, and Andreas Haeberlen. 2012. Djoin: Differentially private join queries over distributed databases. In *Osdi*, 149–162.

Narayanan, Praveen, Jacques Carette, Wren Romano, Chung-chieh Shan, and Robert Zinkov. 2016. Probabilistic inference by program transformation in Hakaru (system description). In *International symposium on functional and logic programming (flops), kochi, japan*, 62–79. Springer-Verlag.

Near, Joseph P., David Darais, Chike Abuah, Tim Stevens, Pranav Gaddamadugu, Lun Wang, Neel Somani, Mu Zhang, Nikhil Sharma, Alex Shan, and Dawn Song. 2019. Duet: An expressive higher-order language and linear type system for statically enforcing differential privacy. *Proc. ACM Program. Lang.* 3(OOPSLA).

Novikov, Petr Sergeevich. 1955. On the algorithmic unsolvability of the word problem in group theory. *Trudy Matematicheskogo Instituta imeni VA Steklova* 44:3–143.

Osera, Peter-Michael, and Steve Zdancewic. 2015. Type-and-example-directed program synthesis. In *Pldi*.

Otto, Friedrich. 1999. *On the connections between rewriting and formal language theory*, 332–355. Berlin, Heidelberg: Springer Berlin Heidelberg.

Perelman, Daniel, Sumit Gulwani, Dan Grossman, and Peter Provost. 2014. Test-driven synthesis. In *Pldi*.

Phothilimthana, Phitchaya Mangpo, Aditya Thakur, Rastislav Bodik, and Dinakar Dhurjati. 2016. Scaling up superoptimization. In *Proceedings of the twenty-first international conference on architectural support for programming languages and operating systems*, 297–310. ACM.

Polikarpova, Nadia, Ivan Kuraj, and Armando Solar-Lezama. 2016. Program synthesis from polymorphic refinement types. In *Proceedings of the*

37th acm sigplan conference on programming language design and implementation, 522–538. PLDI '16.

Polikarpova, Nadia, and Ilya Sergey. 2019. Structuring the synthesis of heap-manipulating programs. *Proc. ACM Program. Lang.* 3(POPL).

Polozov, Oleksander, and Sumit Gulwani. 2015. Flashmeta: A framework for inductive program synthesis. In *Oopsla*.

Proserpio, Davide, Sharon Goldberg, and Frank McSherry. 2014. Calibrating data to sensitivity in private data analysis: a platform for differentially-private analysis of weighted datasets. *Proceedings of the VLDB Endowment* 7(8):637–648.

Prountzos, Dimitrios, Roman Manevich, and Keshav Pingali. 2012. Elixir: a system for synthesizing concurrent graph programs. In *Oopsla*.

———. 2015. Synthesizing parallel graph programs via automated planning. In *Pldi*.

Radoi, Cosmin, Stephen J. Fink, Rodric M. Rabbah, and Manu Sridharan. 2014. Translating imperative code to mapreduce. In *Oopsla*, ed. Andrew P. Black and Todd D. Millstein.

Rand, Robert, and Steve Zdancewic. 2015. VPHL: A verified partial-correctness logic for probabilistic programs. In *Conference on the mathematical foundations of programming semantics (mfps), nijmegen, the netherlands*.

Raychev, Veselin, Madanlal Musuvathi, and Todd Mytkowicz. 2015. Parallelizing user-defined aggregations using symbolic execution. In *Sosp*.

Reddy, Uday. 1989. Rewriting techniques for program synthesis. In *Rewriting techniques and applications*, 388–403. Springer.

Rondon, Patrick Maxim, Ming Kawaguchi, and Ranjit Jhala. 2008. Liquid types. In *Pldi*.

Roy, Indrajit, Srinath TV Setty, Ann Kilzer, Vitaly Shmatikov, and Emmett Witchel. 2010. Airavat: Security and privacy for mapreduce. In *Nsdi*, vol. 10, 297–312.

Rümmer, Philipp, Hossein Hojjat, and Viktor Kuncak. 2013. Classifying and solving Horn clauses for verification. In *Working conference on verified software: Theories, tools and experiments (VSTTE), menlo park, california*, 1–21. Springer.

Rummer, Philipp, and Pavle Subotic. 2013. Exploring interpolants. In *Formal methods in computer-aided design (FMCAD), portland, oregon*, 69–76. IEEE.

Schkufza, Eric, Rahul Sharma, and Alex Aiken. 2013. Stochastic superoptimization. *ACM SIGPLAN Notices* 48(4):305–316.

Singh, Rishabh, and Sumit Gulwani. 2012a. Learning semantic string transformations from examples. *PVLDB* (8).

———. 2012b. Synthesizing number transformations from input-output examples. In *Cav*.

Smith, Calvin, and Aws Albarghouthi. 2016. Mapreduce program synthesis. In *Proceedings of the 37th acm sigplan conference on programming language design and implementation*, 326–340. PLDI '16.

———. 2019a. Program synthesis with equivalence reduction. In *Verification, model checking, and abstract interpretation*, ed. Constantin Enea and Ruzica Piskac, 24–47. Cham: Springer International Publishing.

———. 2019b. Synthesizing differentially private programs. *Proc. ACM Program. Lang.* 3(ICFP).

Smith, Calvin, Gabriel Ferns, and Aws Albarghouthi. 2017. Discovering relational specifications. In *Proceedings of the 2017 11th joint meeting on foundations of software engineering*, 616–626. ACM.

Smith, Calvin, Justin Hsu, and Aws Albarghouthi. 2019. Trace abstraction modulo probability. *Proc. ACM Program. Lang.* 3(POPL):39:1–39:31.

Solar-Lezama, Armando, Liviu Tancau, Rastislav Bodík, Sanjit A. Seshia, and Vijay A. Saraswat. 2006. Combinatorial sketching for finite programs. In *Asplos*.

Spark. Apache spark. `spark.apache.org`.

Srikanth, Akhilesh, Burak Sahin, and William R. Harris. 2017. Complexity verification using guided theorem enumeration. In *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), paris, france*, 639–652.

Summers, Phillip D. 1976. A methodology for lisp program construction from examples. In *Popl*.

Suter, Philippe, Ali Sinan Köksal, and Viktor Kuncak. 2011. Satisfiability modulo recursive programs. In *Sas*.

Teige, Tino, and Martin Fränzle. 2011. Generalized Craig interpolation for stochastic boolean satisfiability problems. In *International conference on tools and algorithms for the construction and analysis of systems (tacas), saarbrücken, germany*, 158–172. Springer-Verlag.

Tran, Quoc Trung, Chee-Yong Chan, and Srinivasan Parthasarathy. 2009. Query by output. In *Sigmod*, ed. Ugur Çetintemel, Stanley B. Zdonik, Donald Kossmann, and Nesime Tatbul, 535–548. ACM.

Twitter. Twitter streaming apis. `dev.twitter.com/streaming`.

Udupa, Abhishek, Arun Raghavan, Jyotirmoy V Deshmukh, Sela Mador-Haim, Milo MK Martin, and Rajeev Alur. 2013. Transit: specifying protocols with concolic snippets. *ACM SIGPLAN Notices* 48(6):287–296.

Wang, Chenglong, Alvin Cheung, and Rastislav Bodik. 2017a. Synthesizing highly expressive sql queries from input-output examples. In *Proceedings of the 38th acm sigplan conference on programming language design and implementation*, 452–466. ACM.

Wang, Di, Jan Hoffmann, and Thomas Reps. 2018. PMAF: An algebraic framework for static analysis of probabilistic programs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), philadelphia, pennsylvania*.

Wang, Xinyu, Isil Dillig, and Rishabh Singh. 2017b. Program synthesis using abstraction refinement. *Proceedings of the ACM on Programming Languages* 2(POPL):63.

Warner, Stanley L. 1965. Randomized response: A survey technique for eliminating evasive answer bias. *Journal of the American Statistical Association* 60(309):63–69.

Wehrman, Ian, Aaron Stump, and Edwin Westbrook. 2006. Slothrop: Knuth-bendix completion with a modern termination checker. In *International conference on rewriting techniques and applications*, 287–296. Springer.

White, Tom. 2015. *Hadoop - the definitive guide: Storage and analysis at internet scale*.

Wikipedia. Wikipedia dumps. dumps.wikimedia.org.

Winkler, Sarah, and Aart Middeldorp. 2010. Termination tools in ordered completion. In *International joint conference on automated reasoning*, 518–532. Springer.

Winograd-Cort, Daniel, Andreas Haeberlen, Aaron Roth, and Benjamin C. Pierce. 2017. A framework for adaptive differential privacy. *Proc. ACM Program. Lang.* 1(ICFP).

Xu, Zhilei, Shoaib Kamil, and Armando Solar-Lezama. 2014. MSL: A synthesis enabled language for distributed implementations. In *Sc*.

Yaghmazadeh, Navid, Yuepeng Wang, Isil Dillig, and Thomas Dillig. 2017. Sqlizer: Query synthesis from natural language. OOPSLA.

Yelp. Yelp dataset challenge. yelp.com/dataset_challenge.

Yu, Yuan, Pradeep Kumar Gunda, and Michael Isard. 2009. Distributed aggregation for data-parallel computing: interfaces and implementations. In *Sosp*.

Yu, Yuan, Michael Isard, Dennis Fetterly, Mihai Budiu, Úlfar Erlingsson, Pradeep Kumar Gunda, and Jon Currey. 2008. Dryadlinq: A system for general-purpose distributed data-parallel computing using a high-level language. In *Osdi*.

Zaharia, Matei, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Nsdi*.

Zaks, Anna, and Amir Pnueli. 2008. Covac: Compiler validation by program analysis of the cross-product. In *Fm*.

Zankl, Harald, Nao Hirokawa, and Aart Middeldorp. 2009. Kbo orientability. *Journal of Automated Reasoning* 43(2):173–201.

Zhang, Sai, and Yuyin Sun. 2013. Automatically synthesizing SQL queries from input-output examples. In *Ase*, ed. Ewen Denney, Tevfik Bultan, and Andreas Zeller, 224–234. IEEE.