Hybrid Resource-Bound Analyses of Programs

Long Pham

CMU-CS-25-122

August 2025

Computer Science Department
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Jan Hoffmann, Chair
Feras Saad
Matt Fredrikson
François Pottier (Inria Paris)

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

This research was sponsored by: the Defense Advanced Research Project Agency under award number 063000; the National Science Foundation under award numbers 1812876, 1845514, 2007784, 2311983 and 1801369; and the Algorand Foundation under award number 1031498. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government or any other entity.

Keywords: program analysis, resource analysis, worst-case costs, amortized costs, potential method, static analysis, type system, data-driven analysis, Bayesian inference

For my family and friends

Abstract

Resource-bound analysis aims to infer symbolic bounds of worst-case resource usage (e.g., running time and memory) of programs. Applications of resource analysis include job scheduling and prevention of side-channel attacks. Different resource-analysis techniques have complementary strengths and weaknesses. (Automatic) static resource analysis, which analyzes the source code of programs, is *sound*: if it successfully infers a cost bound, it is guaranteed to be a valid bound. However, due to the undecidability of resource analysis in general, every static analysis technique is *incomplete*: there exists a program that the analysis technique cannot handle. Meanwhile, data-driven analysis, which statistically analyzes cost measurements obtained by running programs on many inputs, can infer a candidate cost bound for any program. However, it does not guarantee soundness of inference results.

To overcome limitations of individual analysis techniques, this thesis develops *hybrid resource analysis*, which integrates two complementary analysis techniques via a user-adjustable interface. The user first specifies which analysis techniques should analyze which code fragments and quantities. Hybrid analysis then performs its constituent analysis techniques on their respective code fragments and quantities. Finally, their inference results are combined into an overall cost bound. Hybrid resource analysis retains the strengths of constituent analyses while mitigating their respective weaknesses.

The thesis introduces two hybrid-resource-analysis techniques: *Hybrid AARA* and *resource decomposition*. They adopt distinct designs of an interface between constituent analyses, posing a trade-off in the flexibility of hybrid analysis. Hybrid AARA integrates static resource analysis—Automatic Amortized Resource Analysis (AARA)—with data-driven resource analysis via a type-based interface. On the other hand, resource decomposition integrates different pairs of static, data-driven, and interactive resource analyses via a numeric-variable-based interface.

In addition to hybrid resource analysis, I discuss theoretical results of resource analysis: (i) the undecidability of resource analysis; and (ii) the polynomial-time completeness of Conventional AARA. I also describe newly developed Bayesian data-driven resource analysis, which statistically infers cost bounds by Bayesian inference. Finally, I present the optimization of probabilistic program-input generators by a genetic algorithm, showing that its output generator is more effective in triggering high computational cost than randomly generated inputs.

Acknowledgments

First and foremost, I am deeply grateful to my advisor, Jan Hoffmann, for his unwavering support throughout my PhD and patience with my research. He guided me through the process of finding good research problems, which is the most important and difficult aspect of research, as well as solving them. He taught me the value of being incremental in research. Instead of playing with abstract ideas coming out of nowhere, it is better to start with existing works (possibly outside our areas of expertise) and identify how we can apply our expertise to these works. Hybrid resource analysis, which is the primary contribution of this thesis, was born out of Jan's suggestion to investigate how we could apply Bayesian inference to resource analysis. It eventually led us to the development of Hybrid AARA and resource decomposition, which are two approaches to hybrid resource analysis.

I owe my collaborator and committee member, Feras Saad, a great debt of gratitude. He contributed to all of my research projects since he joined CMU in my fourth year of PhD, helping me learn and use probabilistic programming. Had it not been for his guidance, my research on hybrid resource analysis would not have worked out. Feras effectively acted as my co-advisor, teaching me numerous skills ranging from academic writing to managing code bases for implementation and experimentation. I grew up as a researcher by learning from him.

To other thesis committee members, Matt Fredrikson and François Pottier, thank you for serving on the committee. Despite being busy with research, teaching, and managing his startup, Matt was kind enough to join my committee, giving me valuable feedback in the proposal and defense. With a good eye for detail, François read my thesis draft thoroughly, asking me incisive questions and providing helpful feedback on each chapter.

I had the good fortune to work with other PhD students: Di Wang, Yue Niu, and Nathan Glover. Di and I worked together on a research project about programmable probabilistic inference. As a senior PhD student of my research group (when I joined it) and its alumnus, Di has been my role model who is brilliant, kind, and inspiring. Yue, Nathan, and I worked on the resource-decomposition project. Yue's sheer expertise in denotational semantics, category theory, and type theory humbles me, reminding me that there is always something new to learn.

I enjoyed the company of PhD students in my research group: Ankush Das, Di Wang, David Kahn, Nathan Glover, and Ethan Chu. The senior PhD students—Ankush, Di, and David—helped me understand their research when I just joined the group and supported my job search in my final year of PhD. The junior PhD students—Nathan and Ethan—opened my eyes to new research topics through our monthly lunches. It is my honor to be part of this research group.

My office mates hold a special place in my heart: Hugo Sadok, Yonghao Zhuang, Yi Zhou, Justin Whitehouse, Justin Raizes, Siddharth Prasad, Mingjie Sun, and Runming Li. We had lunches together, shared stories, and carried each other through the PhD journey. Hugo enriched my life with his wisdom, humor, and compassion. Yonghao was always there for me when I needed someone to talk to. Yi shared

advice and information with me without hesitation. I thank Justin Whitehouse for brightening my days with his jokes and smiles, and Justin Raizes for keeping me company during lunches and workouts. I am also grateful to Sid for helping me with game theory; to Mingjie for joining me at a magic show in downtown Pittsburgh; and to Runming for helping me with category theory and denotational semantics.

I had the pleasure to make friends with amazing people at CMU: Max Le, Nirav Atre, Jonathan Laurent, Siva Somayyajula, Jalani Williams, Naoki Otani, Hiroaki Hayashi, Lucio Dery, Dorian Chan, Anup Agarwal, Jatin Arora, Francisco Pereira, Myra Dotzel, Jessie Grosen, Harrison Grodin, Aditi Kabra, Yue Yao, Mingkuan Xu, Riku Arakawa, MyAnh Hisaeda, Arnav Sabharwal, and Kimihiro Hasegawa. You all taught me there was more to life than work. I would also like to thank the Funai Foundation for having faith in me and financially supporting my PhD.

Last but not least, I am deeply indebted to my parents, Kien Le Pham and Huong Lien Nguyen, and my sister, Hoa Thanh Morita, for their love and encouragement. You have taught me exactly what I needed to complete my PhD: the strength to rise from failures and dedication to work.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

§1.1 introduces resource-bound analysis of programs and reviews existing resource-analysis techniques studied in the literature. §1.2 then motivates hybrid resource analysis, which integrates two complementary resource-analysis techniques to retain their strengths while mitigating their respective weaknesses. Two approaches to hybrid resource analysis, which are the primary contributions of the thesis are introduced: Hybrid AARA (§1.2.2) and resource decomposition (§1.2.3). Lastly, §1.3 outlines the thesis.

1.1 Resource Analysis

Given a program P, resource-bound analysis aims to infer a symbolic bound $f(x)$ on the worst-case resource usage (e.g., running time, memory, and energy) of the program P as a function of a program input x. The symbolic bound $f(x)$ must be a valid upper bound on the computational cost of executing $P(x)$ for any input x. Hence, the bound $f(x)$ inferred by resource analysis is more precise than the result of asymptotic complexity analysis, which only concerns sufficiently large inputs and disregards constant factors.

Resource analysis of programs has a number of applications. For example, in cloud computing [128, 196, 203], a cloud-service provider seeks to avoid over-provisioning resources, which would waste resources and hence reduce profits, and under-provisioning resources, which could violate service-level agreements. To this end, the cloud-service provider can perform resource analysis to infer cost bounds of the clients' programs. Other applications of resource analysis include worst-case input generation to identify potential algorithmic complexity attacks [42, 155, 160, 185, 219, 226], ensuring constant resource usage[1] to prevent side-channel attacks [27, 46, 176], and detecting performance bugs for programmers [56, 76].

[1] Throughout this thesis, the goal of resource analysis is to infer *upper* bounds of resource usage. However, to prevent resource-based side-channel attacks, it is enough to infer upper bounds. We must either (i) develop a new resource-analysis technique to verify constant resource usage [176]; or (ii) infer a *lower* bound, in addition to an upper bound, and show that the two bounds are equal. In the literature, resource analysis has a broader definition. It is not just about inferring upper bounds—it may infer lower bounds or constant bounds.

Existing resource analyses Three approaches to resource analysis exist in the literature: (automatic) static analysis, data-driven analysis, and interactive analysis. *Static resource analysis* examines the source code of a program and reasons about all possible behaviors of the program, including its worst-case behaviors, to automatically infer a cost bound. Since the pioneering work of Wegbreit [223], numerous static resource-analysis techniques have been developed: type systems [113, 116, 122, 124, 136, 156], recurrence relations [9, 64, 97, 139, 141, 223], term rewriting [22, 23, 123, 174], ranking functions [38, 53, 90, 206], and invariant generation [101, 102, 235].

Data-driven resource analysis first runs a program on many inputs of varying sizes and records execution costs. It then analyzes the dataset of cost measurements to statistically infer a cost bound. To collect cost measurements, most existing works [60, 74, 94, 127, 131, 195, 234] use randomly generated program inputs or representative workloads, which do not necessarily reveal worst-case behaviors of the program. Also, to statistically infer bounds from cost measurements, these works perform optimization (e.g., polynomial regression) without quantifying statistical uncertainty or incorporating the user's domain knowledge into the statistical model.

In *interactive resource analysis*, the user first supplies a candidate cost bound of a program. The cost bound is then either automatically verified by SMT solvers or interactively verified by collaborating with proof assistants. Existing works include those based on separation logic [19, 52, 100, 168, 172], refinement types [62, 104, 144, 218, 222], and dependent types [67, 98, 180].

Complementary strengths and weaknesses Static and data-driven resource analyses have their own strengths and weaknesses that complement each other. Static analysis is *sound*: whenever it successfully returns an inference result, it is guaranteed to be a valid worst-case cost bound of the program. However, static analysis is *incomplete*: for any technique, there exists a program whose cost bound cannot be automatically inferred even if the bound is expressible in the language of symbolic bounds supported by the technique. The incompleteness is due to the undecidability of resource analysis for a Turing-complete programming language.

Remark 1.1.1 (Symbolic cost bounds)**.** *In this thesis, a symbolic cost bound is a computable function f that takes in a program input x (or its size) and returns a non-negative number $f(x)$ as an inferred worst-case cost bound for the input x. Hence, when resource analysis cannot analyze an input program, the resource analysis is not allowed to return a trivial sound bound $f(x) := \infty$. Instead, the resource analysis returns no inference results.* «

In contrast to static analysis, data-driven analysis can infer a candidate cost bound for any program. Another advantage is that data-driven analysis only needs a black-box access to the source code, making the analysis applicable to third-party programs whose source code is not publicly available. However, data-driven analysis does not guarantee soundness of inferred cost bounds, because the analysis does not rigorously reason about worst-case behaviors of the program. Also, a finite dataset of cost measurements used in data-driven analysis may not contain worst-case inputs, making it challenging to infer the true worst-case costs.

Like static resource analysis, interactive resource analysis is sound since user-supplied candidate cost bounds are (automatically or interactively) verified. Furthermore, interactive analysis has greater reasoning power than static analysis: it can analyze more complicated programs and derive more complicated symbolic bounds. However, in exchange for its soundness and

expressive power, interactive analysis sacrifices full automation, which is achieved by static and data-driven analyses.

1.2 Hybrid Resource Analysis

1.2.1 Motivation

In this thesis, to overcome the limitations of individual analysis techniques, I propose and develop *hybrid resource analysis*, which integrates two (or more) resource-analysis techniques with complementary strengths and weaknesses. In hybrid resource analysis, the user first specifies which techniques should analyze which code fragments and quantities in the source code. Next, hybrid analysis performs the two constituent techniques on their designated code fragments and quantities. Finally, the two inference results are combined into an overall cost bound of the entire program. By integrating two complementary analysis techniques, hybrid resource analysis retains their strengths while mitigating their respective weaknesses.

Technical challenge The primary technical challenge of hybrid analysis is the design of the interface between the two constituent analysis techniques. The interface specifies (i) representations of cost bounds inferred by the two analyses and (ii) what information (if any) is exchanged between the two analyses during their inference of cost bounds. Firstly, the cost bounds inferred by the two analyses must have compatible representations such that they can be composed together to yield an overall cost bound. Secondly, some resource-analysis techniques impose numerical constraints on cost bounds to define a set of accepted bounds. Consequently, to successfully compose such a cost bound with another cost bound inferred by a different analysis, the latter bound must satisfy the numeric constraints imposed by the former bound. Therefore, the two analyses must take into account each other's numeric constraints on bounds.

Application To illustrate the benefit of hybrid resource analysis in a real-world use case, consider a cloud-service provider who wishes to estimate resource usage of a client's program to optimize resource allocation on the cloud. One reasonable choice of resource-analysis techniques is static resource analysis as it offers soundness guarantees. The soundness guarantees are beneficial to the service provider because they can rest assured that they will never accidentally under-provision resources. Without the soundness guarantees, it is possible that the client's program consumes more resources than anticipated. In such a case, the provider may need to rerun the program from scratch with more resources, violating service-level agreements on timely execution of the program. However, any single static analysis technique cannot be used to automatically infer cost bounds of all programs due to the incompleteness of static resource analysis. If a static analysis technique of the provider's choice fails to infer a cost bound for a program, the cloud-service provider is left with no clues to guide the resource allocation and scheduling for the program.

Data-driven resource analysis, on the other hand, can always infer a candidate cost bound for any program from its finitely many cost measurements. These measurements are often read-

ily available, especially when the same program is repeatedly executed on many inputs (e.g., the serverless cloud service AWS Lambda). However, data-driven analysis provides no soundness guarantees of inferred cost bounds. Even if the statistical model adds an extra buffer on top of maximum observed costs in the dataset, it may still fail to yield a sufficiently conservative cost bound desired by the cloud-service provider.

Hybrid resource analysis lets the cloud-service provider integrate static and data-driven analyses, thereby striking a desirable balance between soundness (achieved by static analysis) and completeness (achieved by data-driven analysis). For example, the provider can apply static analysis to all code fragments that are amenable to static analysis, and data-driven analysis to the rest of the source code. The cost bounds inferred by static and data-driven analyses are then combined into an overall bound. Oftentimes, even if static analysis fails to analyze the entire program, it is still capable of analyzing a non-trivial amount of code fragments. So it makes sense to apply static analysis wherever possible in the source code, retaining its soundness guarantees as much as possible.

Meanwhile, data-driven analysis yields reasonable (but not necessarily sound) cost bounds for those code fragments that cannot be handled by static analysis. Even though cost bounds inferred by data-driven analysis are not guaranteed to be sound, they are sensible inference results derived using mathematically principled methods (e.g., Bayesian statistics) from observed cost measurements and a statistical model incorporating the user's domain knowledge. Thus, data-driven resource analysis is no less useful than, for example, weather forecasting based on observed data and a weather model, where forecasts never come with guarantees but are nonetheless helpful in our lives.

As I empirically demonstrate in this thesis, hybrid resource analysis returns more accurate cost bounds (i.e., the inferred cost bounds are closer to the ground-truth bound) than purely data-driven analysis, thanks to the integration of static analysis. In summary, hybrid resource analysis can infer cost bounds for program that purely static analysis cannot handle, while obtaining more accurate bounds than purely data-driven analysis.

The thesis statement is therefore:

Thesis Statement *Hybrid resource analysis, which integrates two resource-analysis techniques with complementary strengths and weaknesses, can (i) analyze programs and infer symbolic cost bounds beyond the reach of automatic resource analysis (i.e., static and data-driven analyses); and (ii) add more automation to interactive resource analysis.*

1.2.2 Hybrid AARA

This section introduces the first hybrid resource analysis: Hybrid AARA.

Static and data-driven resource analyses Two approaches to automatic resource analysis are static analysis and data-driven resource analysis. Static analysis analyzes the source code of a program to infer its symbolic cost bound. Data-driven analysis first runs a program on many inputs to collect cost measurements and then statistically infers a symbolic cost bound from the dataset of cost measurements. They both *automatically* infer cost bounds.

Static and data-driven resource analyses have complementary strengths and weaknesses. Static analysis is sound but incomplete in general for a Turing-complete programming language (§6.3.3). If static analysis cannot reason about a given program, the analysis returns no inference results (Remark 1.1.1). In such a case, users must either rewrite their code or resort to interactive resource-analysis techniques [19, 52, 150, 178, 180, 222]. Both of these workarounds require expertise in programming languages and resource analysis, which is a fundamental barrier to adopting static resource analysis for a wider spectrum of applications.

On the other hand, data-driven analysis can infer a candidate cost bound for any program as long as a dataset of cost measurements is available. When viewed as a decision procedure, data-driven analysis is complete: given a ground-truth cost bound, data-driven analysis can correctly conclude that the bound is indeed valid (§6.3.3). In addition to the completeness, another advantage of data-driven analysis is that it does not require access to the source code of a program—only the inputs and outputs of the program need to be observable.

However, data-driven resource analysis comes with its own set of challenges. First, the analysis is sensitive to the given dataset, and data collection can be difficult. For example, inputs generated uniformly at random are unlikely to trigger worst-case behaviors in many programs. Second, commonly used statistical techniques for inferring bounds from cost datasets lack *robustness* (i.e., the inference result has a positive probability of being a sound worst-case cost bound even if the dataset does not contain worst-case inputs) and *accuracy* (i.e., how close the inference result is to a sound worst-case cost bound). Existing data-driven techniques [60, 74, 94, 127, 131, 195, 234] use optimization, and as I demonstrate with experiments (§7.6), bounds derived in this way are prone to being unsound. Prior works also do not quantify any notion of uncertainty in the inferred bounds.

Bayesian resource analysis The first contribution of this work is the design and implementation of *Bayesian resource analyses* for worst-case cost bounds (§7.3). They perform Bayesian inference (§7.1), as opposed to optimization, to statistically infer cost bounds.

Bayesian resource analyses have two advantages over optimization-based ones, which are widely used in the literature of data-driven resource analysis [60, 94, 127, 234]. Firstly, Bayesian resource analyses are generally more customizable than optimization-based ones, as users can express domain knowledge in the form of probabilistic models. Secondly, Bayesian resource analyses return whole posterior distributions of inferred cost bounds, providing greater robustness and richer information about the uncertainty in inference results.

I present two new Bayesian resource analyses: BayesWC (§7.3.3) and BayesPC (§7.3.4). In BayesWC, for each input size present in the runtime-cost data, we conduct Bayesian inference to infer likely values of worst-case costs that are no less than the observed costs. By treating these two costs separately, BayesWC accounts for the possibility that worst-case costs have not been observed in the runtime data. The inferred worst-case costs from BayesWC produce optimization problems that can be solved to obtain cost bounds.

BayesPC, on the other hand, conducts Bayesian inference to directly infer cost bounds, bypassing optimization altogether. Probabilistic models in BayesPC model not only the worst-case costs of individual input sizes but also coefficients of symbolic bounds. Hence, compared to BayesWC, BayesPC allows users to construct richer and more holistic probabilistic models.

In empirical evaluation (§7.6), cost bounds inferred by BayesWC and BayesPC are shown to be more accurate than those inferred by an optimization-based data-driven baseline. However, because they ignore the source code and exclusively rely on data-driven analysis, BayesWC and BayesPC can still fail to infer sound cost bounds in several important benchmarks.

Hybrid AARA The main contribution of this work is the development of a new hybrid resource-analysis—*Hybrid AARA*—that integrates static and data-driven resource analyses via a user-adjustable interface (§7.4). Hybrid AARA mitigates the incompleteness of static resource analysis and improves the robustness and accuracy of data-driven resource analysis. The user-adjustable interface lets the user annotate the source code to specify which code fragments are subject to which analysis techniques. To the best of my knowledge, Hybrid AARA is the first to perform such hybrid resource analysis. A main research challenge is designing a principled interface between static and data-driven resource analyses that enables a modular integration to combine the respective strengths of both approaches.

For the static part of the hybrid resource analysis, I build on Automatic Amortized Resource Analysis (AARA) [113, 114, 116, 122], a type-based state-of-the-art technique that automatically infers polynomial cost bounds of functional programs. Resource-Aware ML (RaML) [117, 118] is an implementation of AARA for analyzing OCaml programs. AARA supports advanced language features such as recursive types [99], side effects [158], and higher-order functions [134]. Two distinguishing features of AARA are the ability to handle non-monotone resources (e.g., memory) and the ability to account for amortization effects. As a type-based technique, AARA is naturally compositional, and automatic cost-bound inference is reduced to off-the-shelf linear-program (LP) solving, even if the derived bounds are higher-degree polynomials.

For the data-driven part of Hybrid AARA, I present a new type system that combines the two Bayesian data-driven resource-analysis methods (BayesPC and BayesWC) with Conventional AARA. Their integration rests on two key technical innovations. For Hybrid AARA with BayesWC, it combines the optimization problems produced by the data-driven Bayesian inference with the linear constraints derived using Conventional AARA type inference to obtain and solve a joint linear program (§7.4.1). For Hybrid AARA with BayesPC, it integrates constraints from Conventional AARA into the probabilistic cost bound models of BayesPC (§7.4.2). Bayesian inference within this model leverages recent innovations from the sampling algorithm literature that allow Hamiltonian Monte Carlo (HMC) sampling to be restricted to a convex polytope defined by AARA's linear constraints [47, 49, 50, 171].

To establish the soundness of Hybrid AARA, I first prove that its inferred bounds are sound with respect to runtime-cost data used in the analysis (Thm. 7.4.1). Additionally, I prove the statistical soundness that the inferred bounds converge to a sound bound if the analysis is repeated with a successively growing set of runtime-cost data that contains worst-case inputs with nonzero probability (Thm. 7.4.2).

Application Hybrid AARA (and its special case of fully data-driven analysis) using Bayesian inference returns a collection of cost bounds that approximate the posterior distribution. Even if the proportion of sound cost bounds in the posterior distribution is less than 100%, Hybrid AARA using Bayesian inference is useful for applications that can tolerate occasional underesti-

mates of the worst-case cost. One such application is job scheduling in cloud computing, where the cloud-service provider would like to have a reasonably accurate (but not necessarily sound at all times) estimate of the resources required to run the job. The job's cost use may respect the tighter bounds in the posterior sample, but if it happens to run out of computational resources, then the cloud-service provider can rerun the job with more resources. Other applications of Hybrid AARA with Bayesian inference include auto-grading of students' programming assignments and annotating software libraries to help users of the library understand its performance characteristics.

Implementation and evaluation My collaborators and I have implemented a prototype of Hybrid AARA (§7.5) on top of RaML [117, 118]. In an empirical evaluation (§7.6), I compare fully data-driven analyses and Hybrid AARA on a curated set of benchmarks that pose challenges to static, data-driven, and hybrid analyses. Hybrid AARA outperforms fully data-driven analyses both when considering soundness and tightness of the bounds.

Contributions In summary, the work on Hybrid AARA makes the following contributions.

1. I present novel Bayesian data-driven resource analyses (BayesWC in §7.3.3; BayesPC in §7.3.4) to infer posterior probability distributions over program cost bounds.
2. I present Hybrid AARA: a novel type-inference system that combines BayesWC and BayesPC with Conventional AARA (§7.4.1 and §7.4.2).
3. I formulate and prove two notions of soundness for Hybrid AARA (Thms. 7.4.1 and 7.4.2)
4. I implement a prototype of Hybrid AARA by extending Resource-Aware ML (RaML) [117].
5. I evaluate Hybrid AARA and fully data-driven resource analyses on a challenging benchmark set, showing examples of improvements in robustness and accuracy (§7.6).

1.2.3 Resource Decomposition

In this section, I first discuss limitations of Hybrid AARA. I then introduce the second hybrid resource analysis: resource decomposition.

Limitations of Hybrid AARA The first hybrid resource analysis, Hybrid AARA (§7.4), has two major limitations. The first limitation is that Hybrid AARA can only express and infer polynomial cost bounds. As a result, Hybrid AARA cannot express, let alone infer, an asymptotically tight bound of the form $cn \log n$ (for some constant $c \in \mathbb{Q}_{\geq 0}$) for MergeSort, because the bound involves a non-polynomial construct (i.e., log). To infer non-polynomial cost bounds within Hybrid AARA, the only way is to use fully data-driven analysis (which is a special case of Hybrid AARA) as it can statistically infer symbolic bounds of arbitrary shapes as long as the corresponding optimization/probabilistic-inference problems can be automatically solved. Alternatively, the user needs to use interactive resource analysis [19, 52, 150, 178, 180, 222], which typically offers more expressive logics and cost bounds at the expense of automation.

The second limitation of Hybrid AARA is that two constituent resource analyses combined by Hybrid AARA must infer quantities of the same resource metric (e.g., running time and memory). Consequently, given a recursive function $P(x)$, Hybrid AARA is unable to infer the

following two quantities using different analysis techniques and then take their product as an overall cost bound: (i) a recursion-depth bound of the function $P(x)$; and (ii) a cost bound of a single recursive call of $P(x)$. This is because the recursion depth and the cost of a recursive call are distinct resource metrics: the former counts the number of recursive calls, while the latter counts the computational cost of a code fragment.

The root cause of these two limitations is the interface between two constituent analyses combined by Hybrid AARA. The interface is the resource-annotated type adopted from Conventional AARA. Resource-annotated types augment standard functional types with polynomial potential functions indicating how much *potential* (i.e., fuel) is available to pay for computational cost. Resource-annotated types are used to encode cost bounds inferred by two constituent analyses of Hybrid AARA, and an overall cost bound is obtained by plugging resource-annotated types statistically inferred by one analysis into a resource-annotated typing tree from the other analysis (Conventional AARA).

The first limitation of Hybrid AARA stems from the fact that resource-annotated types only capture polynomial cost bounds. It is challenging to extend resource-annotated types to logarithm because polynomials and logarithm do not compose well with each other (without sacrificing automatic cost-bound inference). The second limitation of Hybrid AARA arises because all resource-annotated types within an input program P must have the same resource metric. It is impossible to let resource-annotated types in the program P have one resource metric and other types have another resource metric. Such heterogeneous resource metrics would make it impossible to compose resource-annotated types within the program P.

Resource decomposition The first contribution of this work is a new hybrid-resource-analysis technique—*resource decomposition*—that integrates different resource analyses in a more flexible manner than Hybrid AARA. Resource decomposition overcomes the two limitations of Hybrid AARA, albeit with a trade-off[2]. Thanks to the greater flexibility, resource decomposition can coherently integrate more diverse resource analyses than Hybrid AARA: static analyses, data-driven analyses, and manual/interactive analyses. By contrast, due to its interface design based on resource-annotated types, Hybrid AARA is more or less specific to the integration of Conventional AARA and data-driven analyses.

The key idea is to transform the original program into a new program that contains additional constructs that allow different resource analyses to communicate and collaborate with one another. To conduct resource decomposition, a user (or an automatic tool) first identifies custom quantities, called *resource components*, which will be bounded using some resource-analysis method. Examples of resource components include the resource consumption of an auxiliary function, the total cost of all calls to a function, and the maximal recursion depth of a group of function calls.

Once the resource components of a program $P(x)$ are identified, the resource-decomposition framework automatically generate a *resource-guarded* program $P_{\text{rg}}(x, \mathbf{r})$ that facilitates the exchange of information between different analyses. The new numeric parameters $\mathbf{r} \coloneqq (r_1, \ldots, r_m)$ are *resource guards*, one for each resource component, that provide a control-flow skeleton that

[2]Although resource decomposition overcomes the two limitations of Hybrid AARA, it does not strictly improve on Hybrid AARA. §8.8.3 describes resource composition's own drawbacks that Hybrid AARA does not have.

simplifies the resource analysis of P_{rg}.

My collaborators and I prove the following soundness theorem of the framework (§8.4): if $f(x, \mathbf{r})$ is a sound cost bound for the resource-guarded program P_{rg} and if $g_i(x)$ is a sound bound for the resource component r_i ($i = 1, \ldots, m$), then $f(x, g_1(x), \ldots, g_m(x))$ is a sound cost bound for the original program P. The proof of this theorem is based on a novel denotational cost semantics of a resource-sensitive programming language RPCF (§8.1 and §8.2), which equips call-by-value PCF with resource effects, and a binary logical-relation argument.

Instantiations The second contribution of this work is the design and implementation of three instantiations of resource decomposition that demonstrate the efficacy of the technique. They respectively integrate the following pairs of resource analyses:

1. Automatic Amortized Resource Analysis (AARA) [114, 122] and novel Bayesian data-driven resource analysis (§8.5);
2. AARA and an existing interactive resource analysis based on interactive theorem proving (Iris with time credits [52, 168]) (§8.6);
3. Bayesian data-driven resource analysis and another interactive resource analysis based on SMT solving (TiML [222]) (§8.7).

By integrating static and data-driven analyses, the first instantiation of resource decomposition is able to derive tight bounds for challenging programs that existing automatic resource analyses cannot infer. A representative example is the bound $n \log n + m \log n$ for a program that first constructs a balanced binary search tree from a list of n elements and then performs m lookups on this tree. A key innovation is to conduct Bayesian inference on dynamically collected runtime data to derive bounds on a function's recursion depth, which is a challenging task for static analyses. Bayesian inference relies on a domain-specific probabilistic model that infers whether the worst-case recursion depth scales linearly (n) or logarithmically ($\log n$) in the size n of a program input. This probabilistic model infers correct recursion-depth bounds of functions such as MergeSort and QuickSort, which have similar average-case but different worst-case behaviors.

Evaluation To empirically evaluate the effectiveness of the first instantiation, my collaborators and I have curated a set of 13 challenging benchmark programs, for most of which AARA and/or Hybrid AARA fail to derive asymptotically tight bounds, or any bounds at all. The benchmarks include sorting algorithms, repeated operations on balanced and unbalanced binary search trees, and graph algorithms. Using randomly generated inputs for the data-driven analysis, this instantiation infers sound and asymptotically tight cost bounds for most of the benchmarks.

To showcase the efficacy of the second instantiation, I use it to analyze Kruskal's algorithm. By combining a manually verified bound for a union-find data structure [52] with an automatic analysis (AARA), Hybrid AARA successfully infers a sound cost bound involving the inverse Ackermann function, which would be too difficult to derive fully automatically. Likewise, I demonstrate the third instantiation on quicksort where the comparison function has a logarithmic cost in the maximum input number. The integration of SMT-based resource analysis

with Bayesian analysis enables us to infer a symbolic bound involving logarithm, which SMT solvers do not handle well.

Contributions To summarize, the work on resource decomposition makes the following contributions:

1. I propose resource decomposition as a general framework that enables the integration of static, data-driven, automatic, and manual resource-analysis methods (§8.3).
2. I prove that resource decomposition delivers sound resource bounds if all integrated analyses are sound, using a denotational semantics of a resource-sensitive programming language RPCF and a logical-relation argument (Thm. 8.4.1).
3. I develop three concrete instantiations of resource decomposition that each cover distinct combinations of static, data-driven, and manual resource analyses (§8.5–8.7).
4. I empirically demonstrate the efficacy of resource decomposition on a challenging set of benchmark programs that are beyond the reach of existing resource analyses (§8.5–8.7).

1.3 Outline

Structure of the thesis This thesis is structured as follows.

- §2 provides an overview of the primary contribution of this thesis, namely two approaches to hybrid resource analysis: Hybrid AARA and resource decomposition.
- §3 introduces a cost-aware functional programming language, which is equipped with annotations tick q ($q \in \mathbb{Q}$) to indicate resource usage. Conventional AARA [112, 113, 116] analyzes programs written in this programming language.
- §4 introduces Conventional AARA, specifically univariate polynomial AARA [113]. It is a type-based resource-analysis technique that automatically infers univariate polynomial cost bounds of functional programs. Conventional AARA serves as a basis of Hybrid AARA (§7).
- §5 discusses existing works related to this thesis.
- §6 formulates resource analysis as decision problems (§6.2) and proves their undecidability (§6.3). I present both (i) existing undecidability results by Gajser [87, 88], where input programs are partial (i.e., possibly non-terminating); and (ii) new results obtained by adapting Gajser [87, 88] to a setting where input programs are total (i.e., terminating on all inputs). Additionally, I prove polynomial-time completeness of Conventional AARA (§6.4).
- §7 describes Bayesian data-driven resource analysis and Hybrid AARA. Hybrid AARA uses a type-based interface between constituent analysis techniques, and the design of this interface is inspired by Conventional AARA.
- §8 describes resource decomposition and its three concrete instantiations that each integrate different pairs of static, data-driven, and interactive resource analyses.
- §9 presents optimization of program-input generators to improve the inference accuracy of data-driven resource analysis.

- §10 summarizes hybrid resource analysis and discusses future directions.

Publications This thesis is based on the following three publications:

1. Long Pham and Jan Hoffmann. Typable Fragments of Polynomial Automatic Amortized Resource Analysis [186]. Published at CSL 2021.
2. Long Pham, Feras A. Saad, and Jan Hoffmann. Robust Resource Bounds with Static Analysis and Bayesian Inference [188]. Published at PLDI 2024.
3. Long Pham, Yue Niu, Nathan Glover, Feras A. Saad, and Jan Hoffmann. Integrating Resource Analyses via Resource Decomposition [189]. Under submission.

The first paper proves polynomial-time completeness of AARA. The second paper presents Bayesian data-driven analysis and the first hybrid resource analysis, Hybrid AARA. The third paper (under submission) presents the second hybrid resource analysis, resource decomposition, and its three instantiations. Yue Niu, Nathan Glover, and I collaborated on the theoretical foundation of resource decomposition in the third paper.

Chapter 2

Overview

This chapter provides an overview of two approaches to hybrid resource analysis: Hybrid AARA (§2.1) and resource decomposition (§2.2). The two approaches differ in how the resource analysis of a whole program is divided between two constituent resource-analysis techniques. In Hybrid AARA, the resource analysis is divided at the level of code fragments: the user specifies which analysis technique analyzes the cost of which code fragment. Meanwhile, resource decomposition can divide the resource analysis of a program into (i) the analysis of the recursion depth; and (ii) the analysis of the cost of a single recursive call.

2.1 Hybrid AARA

In this section, I review static and data-driven resource analyses, identify their shortcomings, and outline how the first hybrid resource analysis, Hybrid AARA, works. I use the implementation of QuickSort in OCaml (Listing 2.1) as a running example. The goal is to automatically derive a symbolic worst-case cost bound for the function `quicksort`. Let the resource metric of interest be the time cost of executing the comparisons `complex_compare hd pivot` in the function `partition`. I consider different versions of the function `complex_compare`. For example, if the cost of evaluating `complex_compare hd pivot` is bounded by 1, then the worst-case cost of `quicksort xs` is $n(n-1)/2$, where n is the length of the list `xs`.

Static resource analysis A predominant method for automatically deriving symbolic worst-case bounds is *static resource analysis*. Hybrid AARA builds on Automatic Amortized Resource Analysis (AARA) [112, 116, 117, 118] (§4), a compositional type-based static analysis technique.

Resource-Aware ML (RaML [117, 118]) is an implementation of AARA that derives polynomial bounds for a subset of OCaml. The compositionality of AARA ensures that RaML can derive a bound for the function `quicksort` if it can derive a bound for `complex_compare`. Assuming each comparison has cost 1, RaML correctly infers the tight bound $n(n-1)/2$ for `quicksort` in less than 0.1 s. Similarly, assume that the argument of `quicksort` is a list of lists and that `complex_compare` is a lexicographical comparison whose worst-case is k, where k is the length of the first argument list. Then RaML infers the tight bound $mn(n-1)/2$ for `quicksort` in less than 0.2 s, where m is the maximum length of the inner lists. This analysis

```
1  let rec partition pivot xs =
2    match xs with
3    | [] → ([], [])
4    | hd :: tl →
5      let lower, upper = partition pivot tl in
6      if complex_compare hd pivot then
7        (hd :: lower, upper)
8      else (lower, hd :: upper)
```

```
9  let rec quicksort xs =
10   match xs with
11   | [] → []
12   | hd :: tl →
13     let lower, upper = Raml.stat (partition hd tl) in
14     let lower_sorted = quicksort lower in
15     let upper_sorted = quicksort upper in
16     append lower_sorted (hd :: upper_sorted)
```

Lst. 2.1: QuickSort in OCaml. The function `complex_compare` in line 6 is intractable to static resource analysis. The annotation `Raml.stat` in line 13 indicates data-driven resource analysis on `partition`.

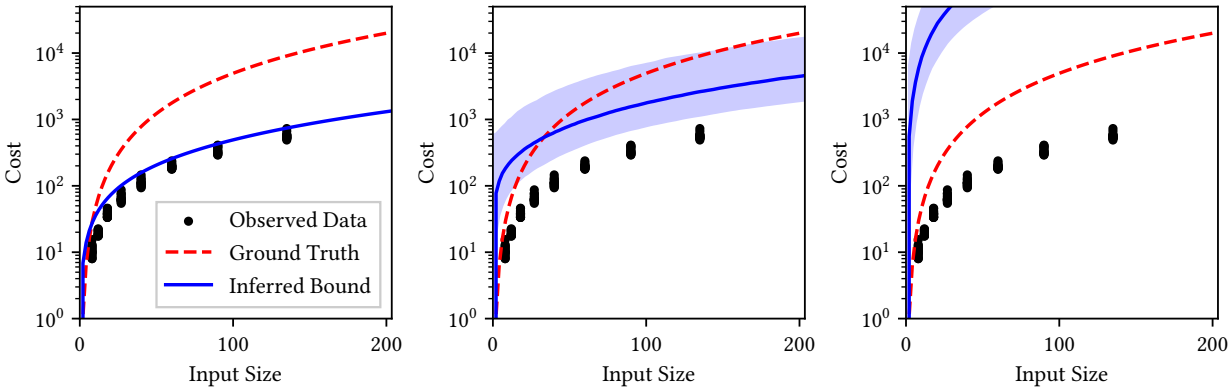
is non-trivial because of the nonstructural recursion: to derive a cost bound, it is not enough to separately analyze the cost of the partition function and the number of recursive steps. We must also analyze how the function `partition` changes input sizes, relaying the size-change information to the next recursive call of `quicksort`.

Because resource analysis is undecidable (§6.3), however, even the most sophisticated static resource-analysis techniques are incomplete: there remain programs that cannot be analyzed automatically. For instance, AARA fails if the control flow depends on mutable data or complex loop conditions. Examples of operations that depend on mutable data include garbage-collector operations and system calls (e.g., accessing files and networks), whose running time depends on the system state. Also, programs such as the function `round` [112, §5.4.3] and the linear-time median-of-medians-based selection algorithm [36] cannot be analyzed by AARA. This is because AARA would need to infer infinitely many typing judgments for the function `round` and reason about mathematical properties of medians for the median-of-medians selection algorithm. Such limitations are not specific to AARA: every static resource-analysis technique has unsupported language features or iteration patterns that make the analysis feel brittle for non-expert users.

Data-driven resource analysis One way to overcome the incompleteness of static analysis is data-driven analysis. Existing data-driven techniques first collect execution costs for different inputs and then solve an optimization problem to fit a symbolic cost bound on the data [60, 74, 94, 127, 195, 234]. Although these optimization-based approaches are fast, they can infer unsound cost bounds and do not quantify the uncertainty over the unknown cost bounds.

To showcase such an approach, suppose that `quicksort` uses the aforementioned comparator `compare_dist` and that the cost varies between 0.5 and 1.0 for different inputs. Fig. 2.1a shows the inferred quadratic cost bound (blue line) of data-driven optimization-based resource analysis adapted from [60, 94, 234], given a randomly generated dataset of measured costs of `quicksort` (black dots). Any worst-case cost bound must lie above all the measured costs and minimize the total L_1 distance between the curve and runtime data. This optimization problem can be framed as a linear program (Opt; §7.3.2). Optimization fails to infer the correct worst-case cost bound (red line), because randomly generated data sets rarely contain worst-case inputs of the function `quicksort`.

To mitigate the shortcomings of such greedy optimizations, I introduce Bayesian data-driven resource analyses (BayesWC and BayesPC; §7.3.3 and §7.3.4). Bayesian resource analy-



(a) Data-Driven Opt (b) Data-Driven BayesWC (c) Hybrid BayesWC

Figure 2.1: Hybrid resource analysis on QuickSort infers more accurate bounds than purely data-driven analyses.

sis enables a user to express their domain knowledge in the form of a probabilistic model that specifies how observed runtime samples are probabilistically generated. It returns an entire probability distribution over the corresponding cost bounds given observed samples. In particular, we condition the probabilistic model on observed cost data and compute the posterior distribution of cost bounds by running sampling-based probabilistic inference algorithms. For `quicksort`, Fig. 2.1b shows the posterior distribution over cost bounds from Bayesian resource analysis. The blue line indicates the median cost bound and the light-blue shade is the 10–90$^{\text{th}}$ percentile range. Most of the cost bounds in the posterior distribution are closer to the true worst-case bound (red line) as compared to optimization-based approach in Fig. 2.1a. In fact, 28/1000 bounds drawn from the posterior distribution are sound. Although this fraction is small, it is already a substantial improvement over optimization.

Hybrid AARA A central contribution of this work is the design, implementation, and evaluation of Hybrid AARA (§7.4), which integrates two novel Bayesian resource-analysis methods (BayesWC and BayesPC) and a simple optimization-based data-driven baseline (Opt) into Conventional AARA. Hybrid AARA is modular: a user freely specifies which code fragments are analyzed by which of data-driven and static analyses. Therefore, Hybrid AARA covers a spectrum ranging from fully data-driven to fully static analyses.

Consider again the example of `quicksort` where the comparator is `compare_dist`, which is intractable for static analysis and has a cost varying between 0.5 and 1.0. If we perform data-driven analysis on the comparator and static analysis on the rest of the code, Hybrid BayesWC amounts to (i) inferring the worst-case constant cost of the comparator by data-driven analysis; and (ii) incorporating the inferred cost into Conventional AARA. All 1000/1000 cost bounds drawn from the posterior distribution are sound (the analysis time in a prototype of Hybrid AARA is 66.2 s).

In the previous example, the interface between the data-driven analysis and the static analysis is particularly simple since the cost of evaluating `quicksort` does not depend on the result of the comparison function. Many examples fall into this category, for which Hybrid AARA

enables the analysis of code that is intractable for purely static methods while clearly outperforming purely data-driven methods. However, an empirical evaluation of Hybrid AARA (§7.6) focuses on a different category of benchmarks for which the integration of data-driven methods is technically challenging and the benefits of a hybrid analysis are less clear.

Challenges for Hybrid AARA Let us revisit the `quicksort` example, using the comparator `compare_dist`. But this time, let us assume that the user marks the call to `partition` for data-driven analysis instead of the call to `complex_compare`. Data-driven analysis can derive a linear cost bound for the function `partition`, but how do we use this information to derive a bound for `quicksort` using AARA? We would need additional information on the size of the result of partition. However, it is not clear what exact information we need. For example, it is insufficient for analyzing `quicksort` to statistically bound the size of each component in the result of `partition`.

A main technical innovation of Hybrid AARA is the design of a principled interface between data-driven and static analyses that can handle such challenging cases. BayesWC and Opt are integrated into AARA by developing a type-inference system that combines the underlying LP problems. Integrating BayesPC into AARA poses a significant challenge because the former relies on sampling algorithms to approximate posterior distributions of resource coefficients, instead of LP problems as in AARA. To overcome this challenge, my collaborators and I leverage an innovative sampling algorithm that allows Hamiltonian Monte Carlo (HMC) sampling [40] to be restricted to a convex polytope [50, 171].

Fig. 2.1c shows the posterior distribution over bounds from Hybrid BayesWC on `quicksort` with the data-driven analysis of `partition`. The 10–90$^{\text{th}}$ percentile range (blue shade) is situated above the true worst-case bound (red line) for all input sizes between 0 and 200. In fact, 471/1000 samples drawn from the posterior distribution in Hybrid BayesWC are theoretically sound bounds for all input sizes, in contrast to the 28/1000 bounds for purely data-driven analysis with BayesWC (Fig. 2.1b) and 0/1000 bounds for data-driven resource analysis with Opt (Fig. 2.1a).

My collaborators and I have empirically evaluated Hybrid AARA on challenging and realistic benchmarks where data-driven analysis is applied to non-trivial code that stresses the interface between data-driven and static analysis (§7.6). The benchmarks demonstrate that (i) Bayesian resource analysis returns more accurate cost bounds than optimization-based analysis; and (ii) Hybrid AARA returns more accurate cost bounds than fully data-driven analysis. Notable among the benchmarks is the linear-time median-of-medians selection algorithm [36]. Conventional AARA cannot statically analyze this program, as it is challenging to reason about how the median of medians influences the partition function. Fully data-driven analysis does not infer a sound cost bound, either, as the worst-case behavior of the partition function rarely occurs in all recursive calls. By integrating static and data-driven analyses, Hybrid AARA successfully infers a sound worst-case cost bound that neither fully data-driven nor fully static analyses can.

```
let rec merge_sort xs =
  let _ = Raml.tick 1.0 in
  match xs with | [] →[]
  | [ x ] → [ x ]
  | _ → let lo, hi = split xs in
    let lo_sorted = merge_sort lo in
    let hi_sorted = merge_sort hi in
    merge lo_sorted hi_sorted
```

(a) MergeSort.

```
1 let rec bubble_sort xs =
2   let _ = Raml.tick 1.0 in
3   let is_xs_sorted, xs_swapped =
4     traverse_and_swap xs in
5   if is_xs_sorted then
6     xs_swapped
7   else
8     bubble_sort xs_swapped
```

(b) BubbleSort.

Lst. 2.2: MergeSort and BubbleSort in OCaml. (a) The call `Raml.tick 1.0` indicates the cost of 1.0 for every function call. (b) The function `traverse_and_swap` (line 4) traverses the input `xs` and returns two outputs: (i) whether `xs` is sorted; and (ii) the result of swapping all out-of-order pairs of consecutive elements in `xs`.

2.2 Resource Decomposition

In this section, I illustrate limitations of (Conventional and Hybrid) AARA and outline how the second hybrid resource analysis, resource decomposition, overcomes the limitations. I use MergeSort (Listing 2.2a) and BubbleSort (Listing 2.2b) as running examples.

The resource-decomposition technique applies to arbitrary resource metrics (e.g., time, energy, and memory) that can be specified by the user. I also allow non-monotone resource metrics such as stack or heap usage, where resources can not only be consumed but also be freed up. For such metrics, we are interested in bounds on the high-water mark of the resource use rather than the net cost. Throughout this section, the resource metric of interest is the number of function calls performed during evaluation, including all recursive calls and helper functions, as an illustrative metric. Under this resource metric, MergeSort and BubbleSort respectively have cost bounds

$$f(x) = 1 + 3.5|x| + 3.5|x|\lceil \log_2(|x|)\rceil \qquad f(x) = 1 + 2|x| + |x|^2, \tag{2.2.1}$$

where $|x|$ is the length of an input list x.

Limitations of Conventional and Hybrid AARA MergeSort and BubbleSort are both challenging for automatic static resource analysis. To infer a resource bound for MergeSort, static analysis must infer that the input list is always split in half. This property is then used to conclude that (i) the recursion depth scales logarithmically; and (ii) the cost across all recursive calls at the same depth scales linearly. For BubbleSort, the analysis must infer that the number of out-of-order pairs of consecutive list elements decreases in each recursive call, which requires a semantic understanding of the code. In Listing 2.2b, BubbleSort repeatedly traverses an input list and swaps out-of-order pairs of elements, until the input list is sorted. Thus, the termination condition of BubbleSort in Listing 2.2b is semantic: the program only terminates when a certain condition holds in the input list.

Automatic Amortized Resource Analysis (AARA) [112, 116] fails to infer desirable cost bounds for MergeSort and BubbleSort. RaML [117, 118], which implements AARA, automatically infers a loose bound

$$f(x) = 1 - 2.5|x| + 3.5|x|^2 \tag{2.2.2}$$

```
let rec merge_sort xs =
  let _ = Raml.mark0 1.0 in
  let _ = Raml.tick 1.0 in
  let result =
    match xs with | [] →[]
    | [ x ] → [ x ]
    | _ → let lo, hi = split xs in
      let lo_sorted = merge_sort lo in
      let hi_sorted = merge_sort hi in
      merge lo_sorted hi_sorted
  in let _ = Raml.mark0 (-1.0)
  in result
```

(a) Resource-decomposed MergeSort.

```
let rec merge_sort xs r =
  let _ = Raml.tick 1.0 in
  let r1 = decrement_r r in
  let result, r_final =
    match xs with | [] →([], r1)
    | [ x ] → ([ x ], r1)
    | _ →
      let (lo, hi) = split xs in
      let lo_sorted, r2 = merge_sort lo r1 in
      let hi_sorted, r3 = merge_sort hi r2 in
      (merge lo_sorted hi_sorted, r3)
  in (result, increment_r r_final)
```

(b) Resource-guarded MergeSort.

Lst. 2.3: MergeSort with a resource component for tracking the recursion depth.

for MergeSort, instead of an asymptotically tight $O(n \log n)$ bound. The reason is that AARA can only express polynomial bounds. AARA fails to infer any bounds for BubbleSort altogether because the size of the input list does not decrease at each recursive step. What decreases in BubbleSort is the number of out-of-order pairs of list elements. However, AARA cannot spot this semantic property of BubbleSort to conclude its termination, much less infer its cost bound.

Furthermore, Hybrid AARA (§4) faces the same challenge as Conventional AARA. Hybrid AARA fails to infer an asymptotically tight $O(n \log n)$ bound for MergeSort and an $O(n^2)$ bound for BubbleSort, unless they are analyzed by fully data-driven resource analysis (which is a special case of Hybrid AARA). Hybrid AARA is designed in such a way that

1. Hybrid AARA inherits the set of expressible symbolic bounds from Conventional AARA;
2. A recursion pattern (i.e., the recursion depth or the number of recursive calls) must be analyzed by Conventional AARA, as opposed to data-driven analysis, if a user chooses to use both static and data-driven analyses.

Since Conventional AARA cannot express an asymptotically tight $O(n \log n)$ bound for MergeSort and cannot reason about the recursion pattern of BubbleSort, neither can Hybrid AARA, unless we resort to fully data-driven analysis.

MergeSort and BubbleSort have been selected to illustrate the limitations of (Conventional and Hybrid) AARA, but these limitations (or similar ones) are common in automatic resource analyses to enable compositionality and scalability. Automatic resource analysis is an undecidable problem (§6.3), so all practical analysis techniques must make trade-offs between efficiency and expressiveness. This results in syntactic limitations for source programs and in restrictions on the mathematical constructs (e.g., logarithm) that can appear in symbolic bounds.

Key insight The main contribution of this work is a hybrid-resource-analysis technique that overcomes the limitations of Conventional and Hybrid AARA. The technique soundly and systematically combines automatic, data-driven, and interactive analyses via a different interface design between constituent analyses.

To illustrate a key insight of the technique, consider a modified version of MergeSort in Listing 2.3b. This version is obtained by augmenting the original program (Listing 2.2a) with a nonnegative numeric variable `r`, called a *resource guard*. This variable is intended to track the recursion depth of MergeSort. At the start of the function body, the variable `r` is decremented

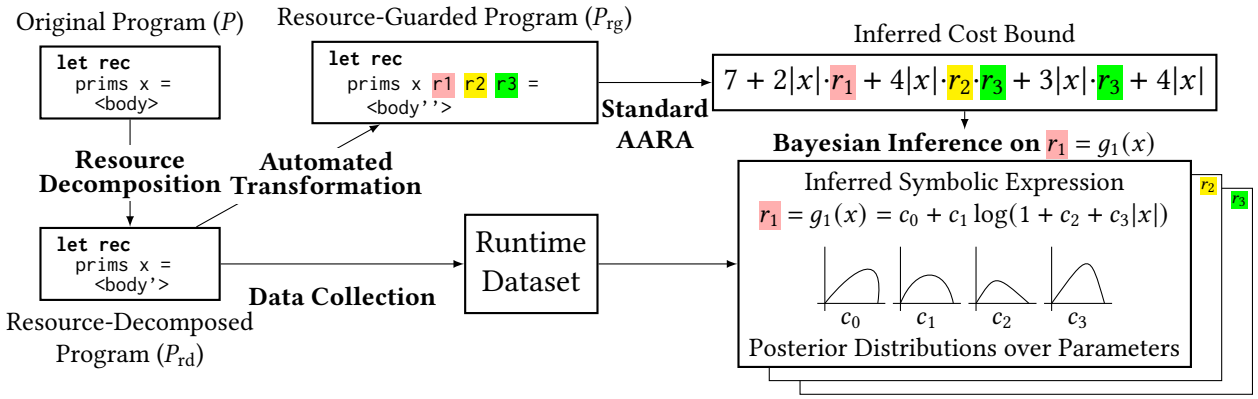


Figure 2.2: Example workflow of resource decomposition, instantiated with AARA and Bayesian data-driven analysis. First, a program P is annotated to specify quantities to be analyzed by Bayesian inference. The resulting resource-decomposed program P_{rd} is transformed (automatically) to a resource-guarded program P_{rg}. It extends P with numeric program variables (`r1`, `r2`, and `r3`), called resource guards, to track the quantities specified in P_{rd}. AARA infers a cost bound of P_{rg}, while Bayesian inference infers symbolic bounds of resource components. An overall bound of the original program P is obtained by inserting resource-component bounds into AARA's inferred bound.

(line 3). If an attempt to decrement `r` is made when its value is zero, then an exception is raised. The decremented resource guard `r` is then passed on to the recursive calls (lines 9 and 10). Finally, its output value is incremented (line 12) before it is returned as the second output of the current function call.

For this modified MergeSort, AARA infers a symbolic resource bound

$$f(x, r) = 1 + 3.5r \cdot |x|, \tag{2.2.3}$$

where r is the resource guard (a nonnegative number) and $|x|$ is the size of the input list `xs`. The modified program is related to the original version as follows: as long as the resource guard `r` is initialized to at least the recursion depth of MergeSort (i.e., $1 + \lceil \log_2(|x|) \rceil$), the modified code successfully terminates (i.e., it does not raise an exception), returns the same output as the original code, and incurs the same cost.

Resource decomposition My collaborators and I have developed *resource decomposition* as a way to systematically exploit the insight illustrated with the modified code of MergeSort. To derive a bound for the original function, we first derive the bound $f(x, r) = 1 + 3.5r \cdot |x|$ for the modified function. Next, we use a different analysis to infer a cost bound (say $g(x)$) for r in terms of the original input x, and substitute the result in f to obtain the overall bound $f(x) = 1 + 3.5g(x) \cdot |x|$, which is proved to be sound for the original program.

Workflow Fig. 2.2 shows the overall workflow of resource decomposition. Given a program $P(x)$, the user or an automatic tool first decides on a set of $m \geq 1$ *resource components*, i.e., the

quantities that the resource guards should track. The program $P(x)$ is then (manually or automatically) instrumented with code annotations $\mathsf{mark}_\ell\, q$, where ℓ is a label that uniquely identifies a resource component and $q \in \mathbb{Q}$[1] denotes how much to increment the resource component's counter, such that their high-water marks (i.e., the highest values reached so far) are equal to the user-specified resource components. Let $P_{\mathrm{rd}}(x)$ denote the resulting *resource-decomposed program*.

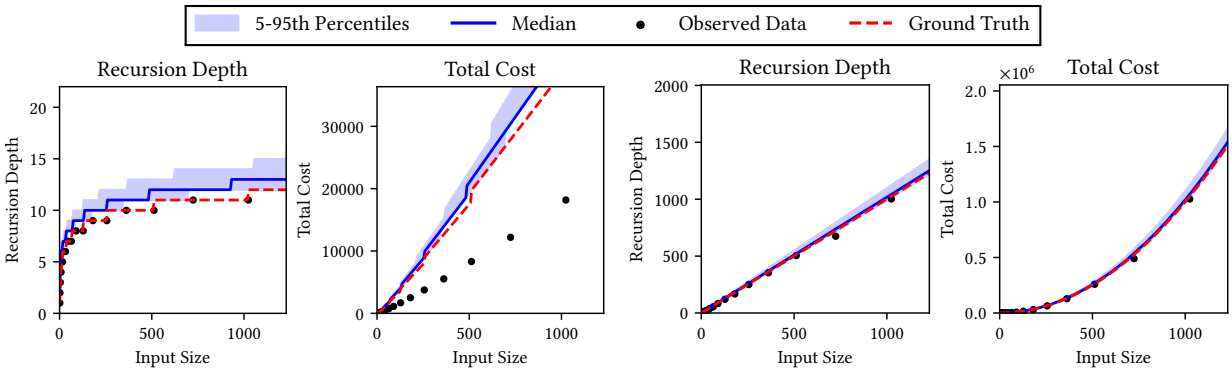
Next, the decomposed program $P_{\mathrm{rd}}(x)$ is automatically translated to a *resource-guarded program* $P_{\mathrm{rg}}(x, \mathbf{r})$ by augmenting the former with resource guards $\mathbf{r} = (r_1, \ldots, r_m)$ as extra input variables, one for each user-specified resource component. In contrast to the annotations $\mathsf{mark}_\ell\, q$ for resource components, the resource guards $\mathbf{r}$ count down, i.e., they are decremented whenever the corresponding resource components are incremented. If the resource-guarded program P_{rg} attempts to decrement a resource guard that is zero, the program raises an exception. We then conduct resource analyses (possibly using different techniques) on (i) the resource-guarded program P_{rg} to derive a symbolic cost bound $f(x, \mathbf{r})$ for the cost of program P; (ii) the resource components of P_{rd} to derive their symbolic bounds $r_i = g_i(x)$ $(i = 1, \ldots, m)$. Finally, we substitute the resource components' symbolic bounds $g_i(x)$ for the resource guards r_i $(i = 1, \ldots, m)$ in the bound $f(x, \mathbf{r})$, obtaining a cost bound $f(x, g_1(x), \ldots, g_m(x))$ for the original program P.

The automatic translation from resource-decomposed programs to the corresponding resource-guarded ones is formalized in §8.3. The soundness of this translation is formulated and proved in §8.4: if $f(x, \mathbf{r})$ is a sound bound for the resource-guarded program $P_{\mathrm{rg}}(x, \mathbf{r})$ and each bound $g_i(x)$ is a sound bound for the i^{th} resource component r_i $(i = 1, \ldots, m)$, then their composition $f(x, g_1(x), \ldots, g_m(x))$ is a sound overall cost bound of the original program $P(x)$.

Instantiation I In §8.5, I instantiate the resource-decomposition framework with AARA and a data-driven resource analysis that focuses specifically on the recursion depths of functions. The rationale of this design is that, on the one hand, AARA struggles with non-trivial recursion patterns such as the logarithmic recursion depth of MergeSort and non-size-decreasing recursion of BubbleSort. On the other hand, recursion depths are particularly well-suited for a data-driven analysis since it is common for the recursion depth to grow linearly or logarithmically in the size of a single parameter of a function, as opposed to, say, a complex multivariate polynomial. This reduces the search space and enables us to design an effective analysis that can infer a logarithmic recursion-depth bound for MergeSort and a linear recursion-depth bound for BubbleSort.

I illustrate the workflow of resource decomposition by describing how this instantiation infers tight bounds for MergeSort and BubbleSort. For MergeSort, we begin by introducing a resource component for the recursion depth. To this end, we insert annotations `Raml.mark0` in the source code of MergeSort (Listing 2.2a), resulting in the resource-decomposed code (Listing 2.3a). Here, the suffix `0` in `Raml.mark0` indicates which resource component we manipulate

[1]Here, the number q in $\mathsf{mark}_\ell\, q$ is a rational number. However, in the formalization of resource decomposition in §8, the number q is required to be an integer. This is because, in a functional programming language already equipped with lists, integers can easily be encoded as lists (i.e., unary encoding). On the other hand, to encode rational numbers, I would need to introduce a new type to the programming language. Nonetheless, conceptually, it is not difficult to extend the type of q from integers to rational numbers.



(a) Posterior resource bounds for MergeSort.

(b) Posterior resource bounds for BubbleSort.

Figure 2.3: Inferred bounds for MergeSort and BubbleSort. (a) In the left and right plots, the ground truth is $1 + \lceil \log_2(|x|) \rceil$ and $1 + 3.5|x| + 3.5|x|\lceil \log_2(|x|) \rceil$, respectively. (b) In the left and right plots, the ground truth is $|x|$ and $1 + 2|x| + |x|^2$, respectively.

(here we only have one). At the start of the function body (line 2), the annotation `Raml.mark0 1.0` increments a recursion-depth counter by one, and at the end of the function body (line 11), the annotation `Raml.mark0 (-1.0)` decrements the counter by one. Thus, if the counter starts with zero, its high-water mark is equal to the recursion depth of the function `merge_sort`.

We then perform a novel Bayesian data-driven resource analysis to infer a symbolic bound g_1 of the resource component (i.e., recursion depth). We first construct a dataset $\mathcal{D}$ of the resource component's runtime measurements by running the resource-decomposed MergeSort (Listing 2.3a) on various inputs. We then statistically infer a recursion-depth bound from the dataset $\mathcal{D}$ by Bayesian inference. A custom probabilistic model, which is another contribution of this work, performs Bayesian model averaging [110] among two models of recursion depths: (i) the linear bound $c_0 + c_1 n$; and (ii) the logarithmic bound $c_0 + c_1 \log(1 + c_2 + c_3 n)$, where $c_0, \ldots, c_3 \in \mathbb{R}_{\geq 0}$ are coefficients to be inferred. The left plot of Fig. 2.3a displays the posterior distribution approximated by 12000 drawn recursion-depth bounds. Here, the red dashed line is the ground-truth bound $1 + \lceil \log_2(|x|) \rceil$, the blue line is the median posterior cost bound, and the light-blue shade shows the 5–95$^{\text{th}}$ percentile range of the posterior samples. All 12,000 posterior samples have the correct asymptotic complexity, that is, logarithmic bounds rather than linear ones.

To infer an overall cost bound of MergeSort, the resource-decomposed code (Listing 2.3a) is automatically translated to the corresponding resource-guarded code (Listing 2.3b). Specifically, the function `merge_sort` in Listing 2.3a is augmented with a resource guard `r`. As described, AARA infers a quadratic bound $f(x, r) = 1 + 3.5r \cdot |x|$. Finally, we substitute the inferred logarithmic recursion-depth bound for the resource guard `r`, deriving an asymptotically tight $O(n \log n)$ bound. The right plot of Fig. 2.3b displays the posterior distribution of overall cost bounds for MergeSort, which are obtained by substituting the posterior recursion-depth bounds into the symbolic bound inferred by AARA.

All 12000/12000 posterior cost bounds have the form $c|x| \log|x|$, instead of the asymptotically looser bound $c|x|^2$ that AARA infers. Furthermore, 7136/12000 (59.5%) posterior samples for the recursion depth are sound with respect to the ground-truth bound $g_1(x) = 1 + \lceil \log_2(|x|) \rceil$

for the resource-guard `r`. This percentage immediately translates to the soundness proportion of 59.5% for the overall cost bound because, by the soundness guarantee of resource decomposition, the symbolic bound $f(x, g_1(x))$ is sound for the overall cost whenever g_1 is sound.

For BubbleSort, I perform a similar analysis by having a resource guard track the recursion depth of `bubble_sort`. Using the same Bayesian model as MergeSort, I obtain 12000/12000 (100%) posterior samples of linear (rather than logarithmic) bounds for the recursion depth. Furthermore, 8762/12000 (73.0%) posterior samples are sound with respect to the ground-truth recursion-depth bound of $g_1(x) = |x|$. Substituting the linear recursion-depth bounds $r_1 = g_1(x)$ for the resource guard `r` in the overall cost bound $f(x, r_1) = 1 + 2r_1 + r_1 \cdot |x|$ inferred by AARA, I obtain quadratic cost bounds $f(x) = 1 + 2|x| + |x| \cdot |x|$. Fig. 2.3b displays the posterior distributions of recursion-depth bounds (left) and overall cost bounds (right) for BubbleSort.

Instantiation II In §8.6, I showcase another instantiation of resource decomposition that integrates AARA and Iris with time credits [168], an interactive resource analysis for OCaml programs that is implemented in the Coq proof assistant [29]. I use this instantiation to analyze Kruskal's algorithm for minimum spanning trees, which uses a union-find data structure. A precise cost bound of this data structure involves the inverse Ackermann function, making the bound too difficult to derive by automatic resource analysis. Existing work by Charguéraud and Pottier [52] has derived a precise bound for a union-find data structure in Coq, and I use AARA to automatically analyze the remaining code to mitigate the burden of interactive resource analysis.

Instantiation III §8.7 presents an instantiation of resource decomposition that integrates Bayesian data-driven analysis with semi-automatic analysis TiML [222], where user-supplied candidate cost bounds are automatically verified by an SMT solver. I demonstrate this instantiation on quicksort where the comparison function has a logarithmic cost in the maximum integer in an input list. As SMT solvers have difficulty reasoning about arithmetic in the presence of logarithm, TiML cannot automatically verify symbolic cost bounds involving logarithm. Hence, I delegate a resource analysis of the comparison function to Bayesian data-driven analysis, which successfully infers logarithmic cost bounds with a high soundness probability.

Resource analysis with resource decomposition The resource-composition framework is very general and not specific to AARA. It can be used to combine more than two analyses to, for instance, integrate an automatic technique with both data-driven and interactive analyses. For instance, we can combine Hybrid AARA (§7) with our new Bayesian analysis for bounding the recursion depth and a recurrence solver.

Chapter 3

Cost-Aware Programming Language

This chapter presents a cost-aware functional programming language used in resource analysis of programs. The language is equipped with a construct tick to indicate resource usage of programs. §3.1 presents the syntax of the language, and §3.2 presents the cost semantics, which augments big-step operational semantics with costs.

3.1 Syntax

Expressions Let $\mathcal{X}$ be a countable set of variable symbols. Expressions e in the programming language are formed by a grammar in Listing 3.1.

The language offers standard constructs of functional programming: constructors and destructors for algebraic data types (i.e., unit, product, sum, and list types) and higher-order functions. The grammar adopts *let-normal form* [113], where constructors (e.g., tuples) and destructors (e.g., pattern matching) operate only on variables[1], as opposed to expressions more generally. Let-normal form simplifies the presentation of operational semantics and a type system. For instance, in a pattern matching case $x\ \{\mathsf{left} \cdot x_1 \hookrightarrow e_1 \mid \mathsf{right} \cdot x_2 \hookrightarrow e_2\}$ for sums, its operational semantics does not need to evaluate $x \in \mathcal{X}$, since it is a variable, rather than an expression that can further evaluate.

In a function definition $\mathsf{fun}\, f\, x = e$, the function body e is allowed to mention the function variable[2] $f \in \mathcal{X}$, thereby defining a recursive function. To define and then use a function, we place a function definition inside a let-binding (e.g., $\mathsf{let}\ y = (\mathsf{fun}\, f\, x = e_1)\ \mathsf{in}\ e_2$).

A construct $\mathsf{share}\ x\ \mathsf{as}\ x_1, x_2\ \mathsf{in}\ e$ indicates variable sharing: two copies of a variable $x \in \mathcal{X}$, denoted by fresh variables $x_1, x_2 \in \mathcal{X}$, are spawned and used inside an expression e. We need this construct to ensure that variables (except functions) are *affine*: they are used at most once in a program. In static resource analysis AARA [112, 113, 116], program variables are equipped with *potential*, which can be viewed as resources or money to pay for computational

[1] Let-normal form is similar to but different from A-normal form [80, 199]: the former requires subexpressions inside constructors and destructors to be variables, whereas the latter allows them to be variables and constants.

[2] The function variable f comes from the set $\mathcal{X}$ of variable symbols. Hence, there is no distinction between function variables and variables in the syntax. Their (slight) difference manifests itself only when a type system is introduced: function variables have arrow types, while variables do not necessarily have arrow types.

$e ::= \langle\,\rangle \mid z$	unit and integer; $z \in \mathbb{Z}$
$\mid x$	variable; $x \in \mathcal{X}$
$\mid \mathsf{left} \cdot x \mid \mathsf{right} \cdot x \mid \mathsf{case}\ x\ \{\mathsf{left} \cdot x_1 \hookrightarrow e_1 \mid \mathsf{right} \cdot x_2 \hookrightarrow e_2\}$	sums
$\mid \langle x_1, x_2 \rangle \mid \mathsf{case}\ x\ \{\langle x_1, x_2 \rangle \hookrightarrow e\}$	products
$\mid [\,] \mid x_1 :: x_2 \mid \mathsf{case}\ x\ \{[\,] \hookrightarrow e_1 \mid (x_1 :: x_2) \hookrightarrow e_2\}$	lists
$\mid \mathsf{fun}\ f\ x = e \mid f\ x$	functions; $f \in \mathcal{X}$
$\mid \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$	let-binding
$\mid \mathsf{share}\ x\ \mathsf{as}\ x_1, x_2\ \mathsf{in}\ e$	variable sharing
$\mid \mathsf{tick}\ q$	resource consumption; $q \in \mathbb{Q}$

Lst. 3.1: Expressions e in a cost-aware functional programming language.

cost. Because potential must not be duplicated, variables with a positive amount of potential must not be used more than once. Hence, to use the same variable $x \in \mathcal{X}$ multiple times, it is first duplicated in the source code, and each of its copies is assigned potential separately. Alternatively, if the programming language incorporates remainder contexts [135, 137], which track the leftover potential of program variables, program variables are no longer required to be affine, allowing us to omit the construct share x as x_1, x_2 in e from the grammar.

Let-normal form coupled with the construct share x as x_1, x_2 in e is called *share-let-normal form*. The syntax restriction of share-let-share-normal form does not affect the expressive power of the language [116, 118]. It is possible to automatically transform arbitrary expressions to share-let-normal form [112, §7.1.1]. Indeed, this automatic transformation is done by a static resource-analysis tool Resource-Aware ML (RaML) [117, 118], which is an implementation of AARA for OCaml programs.

A construct tick q indicates resource usage: it increments a cost counter by $q \in \mathbb{Q}$, which is possibly negative, and returns the unit element $\langle\,\rangle$. If $q \geq 0$, the construct tick q means q units of resources are consumed. Otherwise, if $q < 0$, the construct means $|q|$ units of resources are freed up. To specify a particular resource metric, the user (manually or automatically) inserts tick q throughout their code. As the tick metric lets us consider arbitrary resource metrics, it is a standard practice in the literature of static resource analysis [64, 118, 168, 180, 218].

If all q's in tick q are non-negative, such resource metrics (e.g., running time) are said to be *monotone*. Conversely, if we have $q < 0$, it means resources can be freed up, and such resource metrics (e.g., memory) are *non-monotone*.

Programs A program is given by a closed expression (i.e., it contains no free variables).

In real-world functional programming languages (e.g., OCaml), a program is represented as a pair of

1. A finite set of (possibly mutually recursive) function definitions $\mathsf{fun}\ f\ x = e$ (for all $f \in F$), where F is a finite set of function variables; and

$$
\begin{array}{lll}
v ::= \langle\,\rangle \mid z & & \text{unit and integer; } z \in \mathbb{Z} \\
\quad \mid \mathsf{left} \cdot v \mid \mathsf{right} \cdot v \mid \langle v_1, v_2 \rangle & & \text{sums and product} \\
\quad \mid [\,] \mid v_1 :: v_2 & & \text{lists} \\
\quad \mid \mathsf{closure}(V; f, x.e) & & \text{function closure; } f, x \in \mathcal{X}
\end{array}
$$

Lst. 3.2: Values v in a cost-aware functional programming language.

2. A main expression e_{main} to execute or analyze.

This representation of a program can be transformed to a closed expression by concatenating function definitions by let-bindings and also duplicating function definitions if necessary. For instance, consider a program P where (i) two functions f_1 and f_2 call each other and (ii) a main expression e_{main} mentions both f_1 and f_2. The program P can be transformed to a single closed expression:

$$
\begin{aligned}
P \coloneqq\ & \mathsf{let}\ f_1 = (\mathsf{fun}\, f_1\, x = (\mathsf{let}\ f_2 = (\mathsf{fun}\, f_2\, x = e_2)\ \mathsf{in}\ e_1))\ \mathsf{in} \\
& \quad \mathsf{let}\ f_2 = (\mathsf{fun}\, f_2\, x = e_2)\ \mathsf{in} \\
& \quad e_{\text{main}}.
\end{aligned} \tag{3.1.1}
$$

Terminology The word "function" refers to either a mathematical function or a (computable) function written in some programming language. In this thesis, I mostly simply say functions because it is usually clear from the context whether I refer to mathematical functions or functions defined inside programs. For instance, when I say that resource analysis returns a worst-case cost bound as a *function* $f(x)$ parametric in a program input x, the function f is treated as a mathematical function that is not coded in any particular programming language.

3.2 Cost Semantics

Values Values (i.e., runtime expressions that do not further evaluate) are formed by the grammar in Listing 3.2. A value $\mathsf{closure}(V; f, x.e)$ is a function closure, where V is an environment (i.e., a mapping from variable symbols to their values), $f \in \mathcal{X}$ is a function variable, x is an input variable, and e is an expression defining the function f. When a function definition $\mathsf{fun}\, f\, x = e$ is created and contains a free variable $y \in \mathcal{X}$ that is defined outside this function definition, the value of y is stored in the environment V of a function closure.

Resource monoids Computational cost of programs is described by pairs $(h, r) \in \mathbb{Q}^2_{\geq 0}$. The first component $h \in \mathbb{Q}_{\geq 0}$ is the *high-water-mark cost* (i.e., the peak cost that is reached during the execution). That is, h is the minimum amount of resources necessary for evaluating the expression e successfully (i..e, without running out of resources). The second component $r \in \mathbb{Q}_{\geq 0}$ is the amount of remaining resources after feeding h many resources to the execution.

Thus, the *net cost* is given by $h - r$. Under monotone resource metrics (i.e., all occurrences of tick q satisfy $q \geq 0$), we have $r = 0$, and hence the peak cost is equal to the net cost.

Pairs (h, r) of high-water-mark costs $h \in \mathbb{Q}_{\geq 0}$ and remaining resources $r \in \mathbb{Q}_{\geq 0}$ form a monoid [112, 116], which is called the *resource monoid*.

Definition 3.2.1 (Resource monoid). *Pairs $(h, r) \in \mathbb{Q}^2_{\geq 0}$ form the resource monoid*

$$\mathsf{RM} \coloneqq (\mathbb{Q}^2_{\geq 0}, (0, 0), \oplus), \tag{3.2.1}$$

where the binary operator $\oplus$ is defined as

$$(h_1, r_1) \oplus (h_2, r_2) \coloneqq \begin{cases} (h_1 + h_2 - r_1, r_2) & \text{if } r_1 \leq h_2 \\ (h_1, r_2 + r_1 - h_2) & \text{otherwise,} \end{cases} \tag{3.2.2}$$

and $(0, 0)$ is the identity of the operator $\oplus$.

The binary operator $\oplus$ in Defn. 3.2.1 is used to combine the costs of two sequentially composed expressions. To illustrate the intuition behind Eq (3.2.2), consider two expressions e_1 and e_2 to be executed one after another, each with the costs of $(h_1, r_1) \in \mathbb{Q}^2_{\geq 0}$ and $(h_2, r_2) \in \mathbb{Q}^2_{\geq 0}$, respectively. Suppose $r_1 \leq h_2$, that is, the remaining resources after running the expression e_1 are insufficient to run the expression e_2. In this case, to successfully run the expression e_1 followed by the expression e_2, we should ensure that the expression e_2 receives at least h_2 resources. Hence, we need to supply $h_1 + (h_2 - r_1)$ resources to the sequential composition of the two expressions. Also, after running the sequential composition, the remaining resources are r_2. Thus, when $r_1 \leq h_2$, the combined cost of the sequential composition is $(h_1 + (h_2 - r_1), r_2)$, as in the first line of Eq (3.2.2). The second line of Eq (3.2.2) can be justified by similar reasoning.

The operator $\oplus$ can be defined more succinctly as

$$(h_1, r_1) \oplus (h_2, r_2) \coloneqq (h_1 + \max(h_2 - r_1, 0), r_2 + \max(r_1 - h_2, 0)). \tag{3.2.3}$$

The operator is associative, but is not commutative, as demonstrated by

$$(2, 1) \oplus (1, 0) = (2, 0) \qquad (1, 0) \oplus (2, 1) = (3, 2). \tag{3.2.4}$$

Judgment A big-step cost semantics of the programming language is given by a judgment

$$V \vdash e \Downarrow v \mid (h, r), \tag{3.2.5}$$

where V is an environment (i.e., a mapping from variable symbols to their values), e is an expression, and v is an output value of evaluating the expression e under the environment V. Additionally, $(h, r) \in \mathbb{Q}^2_{\geq 0}$ is an element of the resource monoid for evaluating the expression e under the environment V. The evaluation judgment (3.2.5) is inductively defined in Listing 3.3, where the call-by-value evaluation strategy is adopted.

The only inference rules in Listing 3.3 that manipulate costs are E:Let and E:Tick. The rule E:Let concerns a let-binding let $x = e_1$ in e_2, where we first evaluate expression e_1, bind its output value to variable x, and then evaluate expression e_2. The operator $\oplus$ of the resource

monoid (Defn. 3.2.1) is used to combine costs (h_i, r_i) of expression e_i $(i = 1, 2)$. The rule E:Tick states that, given an expression tick q, if $q \geq 0$, then the high-water-mark cost is $h := q$, and $r := 0$ resources remain after evaluating the expression. Conversely, if $q < 0$, the high-water-mark cost is $h := 0$, and $|q|$ resources are freed up, yielding $r := |q|$.

It is possible to define the high-water-mark cost (but not the net cost) of non-terminating programs. For instance, if a non-terminating program (e.g., a web server) uses a bounded amount of memory throughout its execution, its high-water-mark cost is finite and hence is well-defined. To formally define high-water-mark costs in the presence of non-termination, we could modify the evaluation judgment Eq (3.2.5) such that it is allowed to stop the evaluation at an arbitrary point, recording the high-water-mark cost so far [112, 135]. An overall high-water-mark cost would be defined as the maximum high-water-mark cost reached at any point during the execution.

In this thesis, however, we are not concerned with non-terminating programs, since data-driven resource analysis, which is a key component of this thesis, requires programs to terminate. If programs do not terminate, we cannot measure their high-water-mark costs to be used in statistical analysis, even if their high-water-mark costs are known to be finite theoretically.

E:Var
$$\frac{v = V(x)}{V \vdash x \Downarrow v \mid (0,0)}$$

E:Unit
$$\frac{}{V \vdash \langle\,\rangle \Downarrow \langle\,\rangle \mid (0,0)}$$

E:Int
$$\frac{z \in \mathbb{Z}}{V \vdash z \Downarrow z \mid (0,0)}$$

E:Sum:L
$$\frac{v = V(x)}{V \vdash \mathsf{left} \cdot x \Downarrow \mathsf{left} \cdot v \mid (0,0)}$$

E:Sum:R
$$\frac{v = V(x)}{V \vdash \mathsf{right} \cdot x \Downarrow \mathsf{right} \cdot v \mid (0,0)}$$

E:Case:Sum:L
$$\frac{V(x) = \mathsf{left} \cdot v_1 \qquad V, x_1 \mapsto v_1 \vdash e_1 \Downarrow v \mid (h,r)}{V \vdash \mathsf{case}\ x\ \{\mathsf{left} \cdot x_1 \hookrightarrow e_1 \mid \mathsf{right} \cdot x_2 \hookrightarrow e_2\} \Downarrow v \mid (h,r)}$$

E:Case:Sum:R
$$\frac{V(x) = \mathsf{right} \cdot v_2 \qquad V, x_2 \mapsto v_2 \vdash e_2 \Downarrow v \mid (h,r)}{V \vdash \mathsf{case}\ x\ \{\mathsf{left} \cdot x_1 \hookrightarrow e_1 \mid \mathsf{right} \cdot x_2 \hookrightarrow e_2\} \Downarrow v \mid (h,r)}$$

E:Prod
$$\frac{v_1 = V(x_1) \qquad v_2 = V(x_2)}{V \vdash \langle x_1, x_2\rangle \Downarrow \langle v_1, v_2\rangle \mid (0,0)}$$

E:Case:Prod
$$\frac{V(x) = \langle v_1, v_2\rangle \qquad V, x_1 \mapsto v_1, x_2 \mapsto v_2 \vdash e \Downarrow v \mid (h,r)}{V \vdash \mathsf{case}\ x\ \{\langle x_1, x_2\rangle \hookrightarrow e\} \Downarrow v \mid (h,r)}$$

E:List:Empty
$$\frac{}{V \vdash [\,] \Downarrow [\,] \mid (0,0)}$$

E:List:Cons
$$\frac{v_1 = V(x_1) \qquad v_2 = V(x_2)}{V \vdash (x_1 :: x_2) \Downarrow (v_1 :: v_2) \mid (0,0)}$$

E:Case:List:Empty
$$\frac{V(x) = [\,] \qquad V \vdash e_1 \Downarrow v \mid (h,r)}{V \vdash \mathsf{case}\ x\ \{[\,] \hookrightarrow e_1 \mid (x_1 :: x_2) \hookrightarrow e_2\} \Downarrow v \mid (h,r)}$$

E:Case:List:Cons
$$\frac{V(x) = v_1 :: v_2 \qquad V, x_1 \mapsto v_1, x_2 \mapsto v_2 \vdash e_2 \Downarrow v \mid (h,r)}{V \vdash \mathsf{case}\ x\ \{[\,] \hookrightarrow e_1 \mid (x_1 :: x_2) \hookrightarrow e_2\} \Downarrow v \mid (h,r)}$$

E:Fun
$$\frac{}{V \vdash \mathsf{fun}\, f\, x = e \Downarrow \mathsf{closure}(V; f, x.e) \mid (0,0)}$$

E:App
$$\frac{V \vdash f \Downarrow \mathsf{closure}(U; g, y.e) \mid (0,0) \qquad U, g \mapsto \mathsf{closure}(U; g, y.e), y \mapsto V(x) \vdash e \Downarrow v \mid (h,r)}{V \vdash f\, x \Downarrow v \mid (h,r)}$$

E:Let
$$\frac{V \vdash e_1 \Downarrow v_1 \mid (h_1, r_1) \qquad V, x \mapsto v_1 \vdash e_2 \Downarrow v_2 \mid (h_2, r_2)}{V \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \Downarrow v_2 \mid (h_1, r_1) \oplus (h_2, r_2)}$$

E:Share
$$\frac{V(x) = v \qquad V, x_1 \mapsto v, x_2 \mapsto v \vdash e \Downarrow v' \mid (h,r)}{V \vdash \mathsf{share}\ x\ \mathsf{as}\ x_1, x_2\ \mathsf{in}\ e \Downarrow v' \mid (h,r)}$$

E:Tick
$$\frac{}{V \vdash \mathsf{tick}\ q \Downarrow \langle\,\rangle \mid (\max(0,q), \max(0,-q))}$$

Lst. 3.3: Cost semantics.

Chapter 4

Automatic Amortized Resource Analysis

This chapter introduces state-of-the-art static resource analysis Automatic Amortized Resource Analysis (AARA) [55, 112, 116, 118, 122, 126, 156]. It is a type-based resource-analysis technique that automatically infers polynomial cost bounds of functional programs, where cost bounds are embedded inside types. AARA serves as the foundation of the first hybrid resource analysis, Hybrid AARA (§7). Being a type-based technique, AARA is *compositional*: for every expression e, its cost bound is given by composing the cost bounds of e's constituent subexpressions. This compositionality of AARA lends itself to the design of Hybrid AARA, which integrates inference results of two analysis techniques performed on different code fragments.

First, §4.1 gives an overview of AARA. It is followed by the formulation of resource-annotated types (§4.2) and type system (§4.3). Finally, §4.4 describes how AARA automatically infers polynomial cost bounds.

I focus on univariate (polynomial) AARA [113], which can express univariate polynomial cost bounds. For instance, given two values v_1 and v_2, univariate AARA can express a cost bound $|v_1|^2 + |v_2|^2$, where $|v_i|$ is the size of value v_i ($i = 1, 2$) and each term (e.g., $|v_1|^2$ and $|v_2|^2$) is a univariate polynomial. A more expressive variant of AARA is multivariate AARA [116], and it can capture multivariate polynomial cost bounds such as $|v_1| \cdot |v_2|$, where a term can involve multiple variables. Since univariate AARA has a simpler notation than multivariate AARA, this chapter only introduces univariate AARA. Nonetheless, in the implementation and evaluation of hybrid resource analysis, my collaborators and I build on RaML [117, 118], which implements full-fledged multivariate AARA.

4.1 Overview

This section describes the potential method of amortized resource analysis, which underlies AARA, and illustrates how AARA encodes cost bounds using potential functions assigned to program inputs and outputs.

Potential method AARA automates the potential method from amortized analysis of algorithms and data structures by Sleator and Tarjan [208, 209]. Every variable $x \in \mathcal{X}$ in the source code of a program is assigned a *polynomial potential function* parametric in the size of x's value.

```
1 let rec partition (p : int) (x : int list) =
2   match x with
3   | [] → ([], [])
4   | hd :: tl →
5       let lower, upper = partition p tl in
6       let _ = Raml.tick 1.0 in
7       if hd <= p then (hd :: lower, upper)
8       else (lower, hd :: upper)
```

(a) `partition`

```
let rec quicksort (x : int list) =
  match x with
  | [] → []
  | hd :: tl →
      let lower, upper = partition hd tl in
      let lower_sorted = quicksort lower in
      let upper_sorted = quicksort upper in
      append lower_sorted (hd :: upper_sorted)
```

(b) `quicksort`

Lst. 4.1: QuickSort in RaML [117, 118], which is an implementation of AARA for analyzing OCaml programs. The resource metric of interest is the number of integer comparisons. (a) The expression `Raml.tick 1.0` in line 6 increments a cost counter by one.

Potential can be viewed as fuel or money to pay for computational cost. The goal of the potential method is to assign potential functions such that (i) the potential is always non-negative throughout a program execution and (ii) for every step of computation, the pre-state potential is larger than or equal to the post-state potential plus the cost of computation. Under these two conditions, the initial total potential of the programs (i.e., the combined potential function of all input variables) is a worst-case bound on the total computational cost.

Resource-annotated types To encode polynomial potential functions, AARA augments standard functional-programming types with polynomial coefficients of potential functions, resulting in *resource-annotated types*. To illustrate the types, consider the `partition` function that partitions an integer list around an integer pivot. Listing 4.1a displays an implementation of the function `partition` in Resource-Aware ML (RaML) [117, 118], an implementation of AARA for analyzing resource usage of OCaml programs. Our goal is to derive a worst-case bound on the number of comparisons during an evaluation of `partition`, namely n, where n is the input list length.

In AARA, we type an expression $\texttt{partition}\,(p, x)$, where p is a pivot and x is an input list, as follows:

$$\{p : \texttt{int}, x : L^1(\texttt{int})\}; 0 \vdash \texttt{partition}\,(p, x) : \langle L^0(\texttt{int}) \times L^0(\texttt{int}), 0\rangle. \tag{4.1.1}$$

A resource-annotated type $L^q(\texttt{int})$ indicates that an integer list stores $q \in \mathbb{Q}_{\geq 0}$ units of potential per element. Hence, in the typing context of the typing judgment (4.1.1), the resource-annotated type $L^1(\texttt{int})$ of variable x encodes a linear potential function

$$\Phi(v : L^1(\texttt{int})) = 1 \cdot |v|, \tag{4.1.2}$$

where a list v is a value of variable x at runtime and $|v|$ denotes the length of the list v. Also, the annotation 0 to the left of the turnstile (i.e., $\vdash$) indicates that 0 additional constant potential is stored in the typing context. In total, the input potential in the typing judgment (4.1.1) is $1 \cdot |x| + 0$. Likewise, the output resource-annotated type $\langle L^0(\texttt{int}) \times L^0(\texttt{int}), 0\rangle$ means: (i) the two output lists of $\texttt{partition}\,(p, x)$ store 0 potential; and (ii) 0 additional constant potential remains in the output. In summary, the typing judgment (4.1.1) states that, if we start with the

linear input potential $1 \cdot |x|$, the expression $\texttt{partition}\ (p, x)$ runs successfully without running out of potential, with zero potential remaining in the output of the expression.

A worst-case bound for the *high-water-mark cost* is given by the input potential function in the judgment (4.1.1), namely $1 \cdot |x|$. This bound also serves as a bound for the *net cost*.

Remark 4.1.1 (Cost bounds parametric in input and output sizes)**.** *Instead of an input potential function, we could use the difference between the input and output potential functions as a possibly tighter bound for the net cost. However, the resulting bound is now parametric in not only the input size but also the output size.* «

AARA is naturally compositional because resource-annotated typing judgments assign potential functions to both the input and output of program expressions. Consequently, to derive a cost bound of two sequentially composed expressions e_1 and e_2, it suffices to derive their respective typing judgments such that the expression e_1's output potential is larger than or equal to the input potential required by the expression e_2. To achieve compositionality, other static resource-analysis techniques (e.g., recurrence relations [64, 223]) explicitly track not only costs but also output sizes of expressions. By contrast, in AARA, potential functions stored in the input and output of an expression e tell us both e's cost bound and how the output size relates to the input size. Hence, AARA does not need to separately track costs and output sizes.

To illustrate how resource-annotated types can be composed, assume we have two nested calls to `partition` as in the following function f:

$$\mathsf{fun}\ f\ x = \mathsf{let}\ \langle x_1, x_2\rangle = \texttt{partition}\ (42, x)\ \mathsf{in}\ \texttt{partition}\ (1, x_1) \tag{4.1.3}$$

In the second function call $\texttt{partition}\ (1, x_1)$, we can use the previous typing judgment (4.1.1) for the function `partition`. However, for the first function call $\texttt{partition}\ (42, x)$, we use the typing judgment

$$\{p : \texttt{int}, x : L^2(\texttt{int})\}; 0 \vdash \texttt{partition}\ (p, x) : \langle L^1(\texttt{int}) \times L^1(\texttt{int}), 0\rangle. \tag{4.1.4}$$

It assigns a resource-annotated type $L^1(\texttt{int})$ to the two output lists such that they have enough potential to pay for the subsequent computation. Let v, v_1, v_2 be values of variables x, x_1, x_2, respectively. The intuition is that the input potential $\Phi(v : L^2(\texttt{int})) = 2 \cdot |v|$ of the typing judgment (4.1.1) is used to cover both the cost (i.e., $1 \cdot |v|$) and the potential of the result (i.e., $1 \cdot |v_1| + 1 \cdot |v_2|$) of the first function call. It relies on the fact $|v_1| + |v_2| = |v|$, which AARA's type system implicitly figures out. The potential $\Phi(v_1 : L^1(\texttt{int})) = 1 \cdot |v_1|$ stored in the first output list covers the cost of the second function call $\texttt{partition}\ (1, x_1)$.

Type inference In general, a resource-annotated typing judgment of $\texttt{partition}\ (p, x)$ can be expressed with linear constraints over polynomial coefficients of potential functions:

$$\begin{aligned}&\{p : \texttt{int}, x : L^{q_1}(\texttt{int})\}; q_0 \vdash \texttt{partition}\ (p, x) : \langle L^{r_1}(\texttt{int}) \times L^{r_2}(\texttt{int}), r_0\rangle \\ &\text{subject to } q_1 \geq 1 + q', q' \geq r_1, q' \geq r_2, q_0 \geq r_0.\end{aligned} \tag{4.1.5}$$

The type system of AARA emits similar linear constraints during type inference. The constraints, which are all linear, are then automatically solved with an off-the-shelf linear-program (LP) solver (e.g., CLP [82]). If the linear constraints are solvable (i.e., they have a solution), the solution yields a polynomial potential function of a program input, which in turn serves as a polynomial cost bound.

Polynomial AARA Although the resource-annotated type of the `partition` function only stores *linear* potential functions, AARA can encode *polynomial* potential functions and therefore polynomial cost bounds while retaining compositionality and type inference with linear constraint solving [113, 116]. In (polynomial) AARA, resource annotations inside resource-annotated types record polynomial coefficients of potential functions.

For illustration, consider the function `quicksort` (Listing 4.1b). In terms of the number of comparisons, the worst-case cost of `quicksort` is $n(n-1)/2$, where n is the input list length. This cost bound is expressed by the following typing judgment in AARA:

$$\{x : L^{(0,1)}(\texttt{int})\}; 0 \vdash \texttt{quicksort}\, x : \langle L^{(0,0)}(\texttt{int}), 0\rangle. \tag{4.1.6}$$

Here, the resource-annotated type $L^{(0,1)}(\texttt{int})$ assigns a quadratic potential function

$$\Phi(v : L^{(0,1)}(\texttt{int})) = 0 \cdot \binom{|v|}{1} + 1 \cdot \binom{|v|}{2}, \tag{4.1.7}$$

where $|v|$ is the input list length. More generally, the vector $\vec{q}$ in a resource-annotated type $L^{\vec{q}}(\texttt{int})$ stores coefficients q_i associated with binomial coefficients $\binom{n}{i}$ $(i = 1, \ldots |\vec{q}|)$ in the ascending order of polynomial degrees.

4.2 Resource-Annotated Types

This section describes resource-annotated types of univariate AARA, which encode univariate polynomial potential functions parametric in input sizes.

Types Resource-annotated types A in univariate AARA are formed by the grammar

$A ::= \langle a, q\rangle$	constant potential
$a ::= \texttt{unit} \mid \texttt{int} \mid A_1 + A_2 \mid a_1 \times a_2$	polynomial types; $q_1, q_2 \in \mathbb{Q}_{\geq 0}, d \in \mathbb{N}$
$\mid L^{\vec{q}}(a)$	list type; $\vec{q} \in \mathbb{Q}^d_{\geq 0}$
$\mid A_1 \to A_2$	arrow type.

In a type $\langle a, q\rangle$, $q \in \mathbb{Q} \geq 0$ denotes the amount of constant potential stored in the type, and a is a resource-annotated type that excludes constant potential. A tuple $\vec{q} \in \mathbb{Q}^d_{\geq 0}$ records coefficients of degree-d ($d \in \mathbb{N}$) polynomial potential functions, except their constant (i.e., degree-zero) potential, in the ascending order of polynomial degrees. In a resource-annotated sum type $\langle a_1, q_1\rangle + \langle a_2, q_2\rangle$, the constant potential $q_1, q_2 \in \mathbb{Q}_{\geq 0}$ on the two sides are allowed to be different. Consequently, we can express a fine-grained cost bound that is parametric in both the numbers of left-tagged and right-tagged values (e.g., Boolean values).

Potential functions The amount of potential stored in a value v according to a resource-annotated type a is denoted by $\Phi(v : a)$ and is defined as follows:

$$\Phi(\langle\,\rangle : \mathtt{unit}) = \Phi(z : \mathtt{int}) \coloneqq 0 \qquad z \in \mathbb{Z} \tag{4.2.1}$$

$$\Phi(\mathsf{left} \cdot v : A_1 + A_2) \coloneqq q_1 + \Phi(v : a_1) \qquad \text{where } A_1 = \langle a_1, q_1\rangle \tag{4.2.2}$$

$$\Phi(\mathsf{right} \cdot v : A_1 + A_2) \coloneqq q_2 + \Phi(v : a_2) \qquad \text{where } A_2 = \langle a_2, q_2\rangle \tag{4.2.3}$$

$$\Phi(\langle v_1, v_2\rangle : a_1 \times a_2) \coloneqq \Phi(v_1 : a_1) + \Phi(v_2 : a_2) \tag{4.2.4}$$

$$\Phi([\,] : L^{\vec{q}}(a)) \coloneqq 0 \tag{4.2.5}$$

$$\Phi(v_1 :: v_2 : L^{\vec{q}}(a)) \coloneqq q_1 + \Phi(v_1 : a) + \Phi(v_2 : L^{\triangleleft(\vec{q})}(a)) \qquad \text{where } \vec{q} = (q_1, \ldots, d_d) \tag{4.2.6}$$

$$\Phi(v : A_1 \to A_2) \coloneqq 0. \tag{4.2.7}$$

In Eq (4.2.6), the shift operator $\triangleleft$ on tuples $\vec{q} \in \mathbb{Q}^d_{\geq 0}$ is used to derive an appropriate resource annotation for the tail v_2 of a list $v_1 :: v_2$. The operator is defined as

$$\triangleleft(q_1, \ldots, q_d) \coloneqq (q_1 + q_2, \ldots, q_{d-1} + q_d, q_d). \tag{4.2.8}$$

It is possible to show that a length-n list $[v_1, \ldots, v_n]$ of type $L^{\vec{q}}(a)$ has potential

$$\Phi([v_1, \ldots, v_n] : L^{\vec{q}}(a)) = \sum_{i=1}^{d} q_i \binom{n}{i} + \sum_{i=1}^{n} \Phi(v_i : a) \qquad \text{where } \vec{q} = (q_1, \ldots, q_d). \tag{4.2.9}$$

It is a degree-d polynomial parametric in the list length n if $\Phi(v_i : a) = 0$ holds for $i = 1, \ldots, n$.

Polynomial potential functions expressible in AARA are linear combinations of binomial coefficients $\binom{n}{i}$ $(i = 0, \ldots, d)$, rather than powers n^i $(i = 1, \ldots, d)$, where the coefficients in the linear combinations are non-negative:

$$\sum_{i=0}^{d} q_i \binom{n}{i} \qquad (q_0, \ldots, q_d \in \mathbb{Q}_{\geq 0}). \tag{4.2.10}$$

The coefficients q_i $(i = 0, \ldots, d)$ must be non-negative such that the outputs of potential functions are non-negative[1]. Binomial coefficients are more suitable as a basis of polynomial potential functions' space because non-negative linear combinations (i.e., linear combinations with non-negative coefficients) of binomial coefficients are a strict superset of those of powers. For instance, non-negative linear combinations of powers cannot express a polynomial bound $\binom{n}{2} = n(n-1)/2$ of the function `quicksort` (Eq (4.1.6)), as we would need a negative coefficient in a linear combination of powers. Meanwhile, every power n^d $(d \in \mathbb{N})$ can be expressed as a non-negative linear combination of binomial coefficients (Lem. 6.4.3).

Eq. (4.2.7) states that function closures store zero potential. The type system of AARA requires the environment V inside a function closure $\mathsf{closure}(V; f, x.e)$ to carry zero potential. This allows the same function closure to be called arbitrarily many times. However, a downside is that we cannot freely define functions. We can only create function closures that only use potential inside input variables, but not potential stored inside environments.

[1] A polynomial potential function $\Phi(x) = \sum_{i=0}^{d} q_i \binom{n}{i}$ can have non-negative outputs for all inputs x even if some coefficients q_i are negative. However, if the coefficients are allowed to be negative, it is non-trivial to automatically check whether a given polynomial potential function is non-negative for all inputs. Hence, for simplicity, AARA requires all coefficients in linear combinations to be non-negative.

4.3 Type System

This section presents a type system of univariate AARA [113] and its soundness theorem. I introduce the following meta-variables to be used in the type system:

$$\begin{aligned} \mathcal{T} &::= \{A_{1,i} \to A_{2,i} \mid i \in I\} && \text{set of resource-annotated arrow types; } I \subseteq \mathbb{N} \\ \mathbf{a} &::= a \mid \mathcal{T} \\ \mathbf{A} &::= \langle \mathbf{a}, q \rangle && q \in \mathbb{Q}_{\geq 0}. \end{aligned}$$

A set $\mathcal{T}$ of resource-annotated types can only appear at the outermost level. It cannot be placed inside a, such as $L^{\vec{q}}(\mathcal{T})$.

Judgment A resource-annotated typing judgment of univariate AARA is

$$\Gamma; p \vdash e : \langle \mathbf{a}, q \rangle, \tag{4.3.1}$$

where Γ is a resource-annotated typing context, $p \in \mathbb{Q}_{\geq 0}$ is constant potential of the typing context, $\mathbf{a}$ is a resource-annotated type (or a set of resource-annotated arrow types) of the output, and $q \in \mathbb{Q}_{\geq 0}$ is constant potential stored in the output. The typing context Γ maps a variable $x \in \mathcal{X}$ to a resource-annotated type a if x is not a function (i.e., the type a is not of the form $A_1 \to A_2$). Conversely, if $f \in \mathcal{X}$ is a function variable, then the typing context Γ maps the function f to a (possibly infinite) set $\mathcal{T}_f$ of resource-annotated types.

The typing context Γ assigns sets of resource-annotated types, as opposed to single types, to functions in order to achieve *resource-polymorphic recursion* [112]. It means that, to justify a resource-annotated type of a recursive function f, we are allowed to use a different resource-annotated type of the function f's recursive call. Furthermore, by assigning multiple resource-annotated arrow types to a function f, different call sites of the function f are allowed to use different types.

The typing judgment (4.3.1) means, given a well-typed environment $V : \Gamma$ that carries potential $p + \Phi(V : \Gamma)$, if expression e evaluates to value v, then v is well typed with respect to a resource-annotated type (or a set thereof) $\mathbf{a}$ and carries $q + \Phi(v : \mathbf{a})$ units of potential. Here, I define

$$\Phi(v : \mathbf{a}) \coloneqq \begin{cases} \Phi(a) & \text{if } \mathbf{a} = a \\ 0 & \text{otherwise,} \end{cases} \tag{4.3.2}$$

where $\Phi(a)$ in the first line is defined in Eqs. (4.2.1)–(4.2.7). Additionally, the potential $\Phi(V : \Gamma)$ of an environment V is defined as

$$\Phi(V : \Gamma) \coloneqq \sum_{x \in \mathsf{dom}(V)} \Phi(V(x) : \Gamma(x)). \tag{4.3.3}$$

Syntax-directed rules Listing 4.2 displays syntax-directed typing rules of univariate AARA. The rule T:List:Empty states that we can type the empty list [ ] with any resource-annotated list type $L^{\vec{q}}(a)$. This is sensible because the empty list carries zero potential regardless of resource annotations. The rule T:List:Cons states that, to prepend an element x_1 to a list x_2

such that the resulting list has a resource annotation $\vec{q}$, we need a judgment $x_2 : L^{\triangleleft \vec{q}}$ and constant potential $q_1 \in \mathbb{Q}_{\geq 0}$ in the typing context. Here, the shift operator $\triangleleft$ (Eq (4.2.8)) is used to compute the resource annotation of a list's tail. Dually, in the rule T:Case:List, if we perform pattern matching on a list $x : L^{\vec{p}}$, its potential is split into the potential stored inside the tail $x_2 : L^{\triangleleft \vec{p}}(a)$ and the constant potential p_1. By Eq (4.2.6), pattern matching of lists preserves the total potential.

The rule T:Fun concerns function definitions. It derives a type $\langle \mathcal{T}, 0 \rangle$, where $\mathcal{T}$ is a (possibly infinite) set of resource-annotated arrow types. The premise of the rule T:Fun states that, to justify each arrow type in the set $\mathcal{T}$, we can assign any arrow type in the set to the recursive call, and this type is allowed to be different from the original arrow type being justified. The typing context $|\Gamma|$ in the premise of T:Fun is obtained by removing all resource annotations in the original resource-annotated typing context Γ, except those resource annotations inside arrow types. Formally, we define

$$|\Gamma| \coloneqq \{x : |a| \mid x : a \in \Gamma\} \cup \{x : \mathcal{T} \mid (x : \mathcal{T}) \in \Gamma\}, \tag{4.3.4}$$

where $|a|$ is defined as

$$|\mathtt{unit}| = \mathtt{unit} \qquad |\mathtt{int}| = \mathtt{int} \tag{4.3.5}$$

$$|\langle a_1, q_1 \rangle + \langle a_2, q_2 \rangle| = \langle |a_1|, 0 \rangle + \langle |a_2|, 0 \rangle \qquad |a_1 \times a_2| = |a_1| \times |a_2| \tag{4.3.6}$$

$$|L^{\vec{q}}(a)| = L^0(|a|) \qquad |A_1 \to A_2| = A_1 \to A_2. \tag{4.3.7}$$

The rule T:Var:Fun is used to select a type from a set $\mathcal{T}$ of resource-annotated arrow types for functions. This rule is used before we apply the rule T:App, which expects a typing judgment $f : A_1 \to A_2$ as opposed to $f : \mathcal{T}$. Additionally, the rule T:Var:Fun is necessary when we want to embed functions inside data types. For instance, suppose we wish to create a list $x : L^{\vec{q}}(A_1 \to A_2)$ of functions. To prepend a function variable $f : \mathcal{T}$ to the list x, we must extract the type $A_1 \to A_2$ from the set $\mathcal{T}$ using the rule T:Var:Fun, provided that $A_1 \to A_2 \in \mathcal{T}$ holds.

The rule T:Share splits the potential in variable $x : \mathbf{a}$ between variables $x_1 : \mathbf{a}_1$ and $x_2 : \mathbf{a}_2$. The sharing of potential is denoted by $\mathbf{a} \curlyvee (\mathbf{a}_1, \mathbf{a}_2)$, and is defined in Listing 4.3. The rules SH:Fun and SH:Set allow resource-annotated arrow types to be duplicated freely. This is because well-typed functions do not contain free variables equipped with potential, as enforced by the rule T:Fun.

The rules T:Tick:Pos and T:Tick:Neg type the construct tick q ($q \in \mathbb{Q}$). If $q \geq 0$, the rule T:Tick:Pos requires q constant potential in the typing context, and the leftover potential in the output is zero. Conversely, if $q < 0$, the rule T:Tick:Neg requires zero potential in the typing context, and $-q$ units of potential become available in the output.

Structural rules Listing 4.4 displays structural typing rules of univariate AARA. The rules T:Sub and T:Sup mention the subtyping relation $a_1 <: a_2$ between resource-annotated types. It means that the type a_1 is a subtype of the type a_2, that is, any variable $x : a_1$ can be treated as $x : a_2$. The subtyping relation is defined in Listing 4.5. The rule T:Relax is used to inject additional potential into the typing context and the output type.

Soundness The soundness of univariate AARA relates the typing judgment (4.3.1) and the cost-semantics evaluation judgment (3.2.5). The soundness means that, if an expression e is well-typed in univariate AARA, then when we run the expression e under a well-typed environment V, the output v is also well typed, provided that the expression e terminates. Furthermore, if the environment V carries potential as specified by its resource-annotated type, the output v also carries potential as specified by its resource-annotated type.

Thm. 4.3.1 formally states the soundness of univariate AARA. The theorem mentions well-typed values $v : \mathbf{a}$ and well-typed environments $V : \Gamma$. They are defined in Listing 4.6. Whenever values and environments are well-typed, the notations $\Phi(v : \mathbf{a})$ (Eq (4.3.2)) and $\Phi(V : \Gamma)$ (Eq (4.3.3)) are well-defined. Furthermore, if $\mathbf{a} = \mathcal{T}$ is a set of resource-annotated arrow types, a function closure $v : \mathcal{T}$ must exhibit the cost behavior according to the resource-annotated types in the set $\mathcal{T}$ when the function v is applied (V:Set). This property of well-typed function closures is necessary to make the inductive proof of Thm. 4.3.1 go through.

Theorem 4.3.1 (Soundness of univariate AARA [113])**.** *Consider (i) a well-typed expression e such that $\Gamma; p \vdash e : \langle \mathbf{a}, q \rangle$ and (ii) a well-typed environment V such that $V : \Gamma$. If $V \vdash e \Downarrow v \mid (h, r)$, then we have $v : \mathbf{a}$. Furthermore, we have*

$$\Phi(V : \Gamma) + p \geq h \qquad \Phi(V : \Gamma) + p - \Phi(v : \mathbf{a}) - q \geq h - r. \tag{4.3.8}$$

That is, $\Phi(V : \Gamma) + p$ is a bound on the high-water-mark cost h, and $\Phi(V : \Gamma) + p - \Phi(v : \mathbf{a}) - q$ is a bound on the net cost $h - r$.

Limited expressiveness of univariate AARA Although multivariate polynomials (e.g., $(n_1 + n_2)^2$) can always be bounded by univariate polynomials (e.g., $2n_1^2 + 2n_2^2$), univariate AARA is strictly less expressive than multivariate AARA: there exists a program that can be typed in multivariate AARA but not in univariate AARA. For instance, consider the append function that takes two lists x and y and concatenates them. Suppose we want the output of the function append to store super-linear potential (e.g., quadratic potential). Since the output of the function append has length $|x| + |y|$, to equip the output with quadratic potential, the two input lists should store a total of $(|x| + |y|)^2$ units of potential. This is a multivariate polynomial potential function, which is bounded above by a univariate polynomial $2|x|^2 + 2|y|^2$. However, the function append cannot be typed this way in univariate AARA [112, 113, 116].

4.4 Type Inference

Cost bounds of programs are given by potential functions encoded by the inputs' resource-annotated types. Hence, cost-bound inference amounts to inference of resource-annotated types of AARA. Just like functional programming's type inference, AARA's type inference works by solving constraints, specifically numeric linear constraints over polynomial coefficients of potential functions.

Workflow The user first supplies a functional program P (i.e., a closed expression) to be analyzed, which typically has an arrow type. Additionally, the user specifies a maximum polynomial degree $d \in \mathbb{N}$ of cost bounds considered by AARA. The type-inference engine first

transforms the program P to share-let-normal form [118] and infers *unannotated* types (i.e., without resource annotations) by the Hindley-Milner type inference [109, 170].

Next, the type-inference engine walks through the program P, constructing its typing tree according to AARA's typing rules (Listings 4.2 and 4.4). Resource-annotated types (e.g., $L^{\vec{q}}(\texttt{int})$) appearing in this typing tree are created by augmenting unannotated types (e.g., $L(\texttt{int})$) with resource annotations (e.g., $\vec{q}$) consisting of non-negative numeric variables. These numeric variables represent polynomial coefficients of potential functions assigned to variables.

AARA's typing rules induce linear constraints over (symbolic) resource annotations. For instance, the rules T:List:Cons and T:Case:List induce linear constraints that encode the shift operator $\lhd$. The rule T:Share induces a linear constraint to ensure that the potential is correctly split between two copies of a variable (Listing 4.3). The rule T:Tick:Pos for the construct tick q ($q \geq 0$) induces a linear constraint that the typing context must store q units of potential such that it can pay for the computational cost of running the expression tick q.

Let C be a set of linear constraints extracted from the typing trees of the program $P : a_1 \to a_2$. RaML [117, 118], an implementation of multivariate AARA, creates a *lexicographic linear program* L where (i) the constraints are C; and (ii) the objective is to minimize the resource annotations of the input resource-annotated type a_1 in a lexicographic order of their polynomial degrees. That is, we first optimize the sum of all resource annotations of the highest degree subject to the constraints C, fixing the resource annotations to their optimal values. We repeat the same step for the remaining resource annotations in the descending order of their polynomial degrees.

Alternatively, the objective of the linear program L can be to minimize a weighted sum of all coefficients, with higher weights assigned to higher-degree coefficients. This objective is adopted in an earlier version of RaML [112].

Finally, the linear program L is fed to a linear-program solver (e.g., CLP [82]). If the linear program L has a solution, it translates to a resource-annotated input type a_1 with concrete resource annotations. It serves as a worst-case polynomial cost bound of the program P's high-water-mark cost.

Resource-polymorphic recursion In the typing judgment (4.3.1), the typing context Γ assigns to each function variable f a possibly infinite set $\mathcal{T}_f$ of resource-annotated arrow types. We let the set $\mathcal{T}_f$ contain multiple types to enable *resource-polymorphic* recursion: a type of a function f can be justified using a different type for f's recursive call. Furthermore, some functions require infinite sets $\mathcal{T}$ of resource-annotated arrow types. For example, a recursive function $\texttt{round} : L(\texttt{int}) \to L(\texttt{int})$ in Hoffmann [112, §5.4.3] requires an infinite set of types

$$\mathcal{T} := \{\langle L^q(\texttt{int}), 0\rangle \to \langle L^q(\texttt{int}), 0\rangle \mid q = 2^k, k \in \mathbb{N}\}, \tag{4.4.1}$$

where the type derivation of $\texttt{round} : \langle L^q(\texttt{int}), 0\rangle \to \langle L^q(\texttt{int}), 0\rangle$ requires the type $\texttt{round} : \langle L^{2q}(\texttt{int}), 0\rangle \to \langle L^{2q}(\texttt{int}), 0\rangle$ ($q \in \mathbb{N}$) for the recursive call, resulting in an infinite chain of arrow types.

Infinite sets of resource-annotated types pose a challenge to automatic type inference: we cannot algorithmically infer infinitely many types one by one. To overcome this challenge, we can either (i) restrict the sets of resource-annotated arrow types to be finite or (ii) develop a

finite encoding of infinite sets of resource-annotated arrow types. AARA [112, 113, 116] and its implementation RaML [117, 118] adopt the first idea, where *resource-monomorphic recursion* eventually kicks in to terminate the type-inference algorithm. On the other hand, Kahn [135, §8] pursues the second idea, using linear maps (i.e., transformations represented by matrices) to encode infinite sets of resource-annotated types (e.g. Eq (4.4.1)). However, the idea of linear maps is not a strict improvement over the original AARA. Since linear maps cannot represent non-linear transformations (e.g., min) of resource annotations, there exist functions whose types can be algorithmically inferred by the original AARA and RaML but not by linear maps.

In the algorithmic type inference in univariate AARA [113], a key idea is to express a resource-annotated arrow type of a function f as the (component-wise) sum of two resource-annotated arrow types of the same user-specified polynomial degree $d \in \mathbb{N}$: (i) $f : \mathbf{a}_1 \to \mathbf{a}_2$ under the tick metric, where the cost is indicated by the construct tick in the source code of the function f; and (ii) $f : \mathbf{a}_{1,\mathrm{cf}} \to \mathbf{a}_{2,\mathrm{cf}}$ under the *cost-free metric*, where the computational cost of running the function f is zero. The first tick-metric type is shared by all call sites of the function f in the source code, whereas the second cost-free type is specific to each call site. Thus, by varying cost-free types of f from call site to call site, we can also create multiple distinct tick-metric types of f, one for each call site. Furthermore, a cost-free type $f : \mathbf{a}_{1,\mathrm{cf}} \to \mathbf{a}_{2,\mathrm{cf}}$ is derived resource-monomorphically: a recursive call to f is assigned the same cost-free type $\mathbf{a}_{1,\mathrm{cf}} \to \mathbf{a}_{2,\mathrm{cf}}$ as the original call.

$$\text{T:Var}\ \frac{}{x : \mathbf{a}; 0 \vdash x : \langle \mathbf{a}, 0\rangle} \qquad \text{T:Var:Fun}\ \frac{(A_1 \to A_2) \in \mathcal{T}}{f : \mathcal{T}; 0 \vdash f : \langle A_1 \to A_2, 0\rangle} \qquad \text{T:Unit}\ \frac{}{\cdot; 0 \vdash \langle\,\rangle : \langle \mathtt{unit}, 0\rangle} \qquad \text{T:Int}\ \frac{z \in \mathbb{Z}}{\cdot; 0 \vdash z : \langle \mathtt{int}, 0\rangle}$$

$$\text{T:Sum:L}\ \frac{x : a; p \vdash x : A_1}{x : a; p \vdash \mathsf{left} \cdot x : \langle A_1 + A_2, 0\rangle} \qquad \text{T:Sum:R}\ \frac{x : a; p \vdash x : A_2}{x : a; p \vdash \mathsf{right} \cdot x : \langle A_1 + A_2, 0\rangle}$$

$$\text{T:Case:Sum}\ \frac{\Gamma, x_1 : a_1; p_1 + p_0 \vdash e_1 : \mathbf{A} \qquad \Gamma, x_2 : a_2; p_2 + p_0 \vdash e_2 : \mathbf{A}}{x : \langle a_1, p_1\rangle + \langle a_2, p_2\rangle, \Gamma; p_0 \vdash \mathsf{case}\ x\ \{\mathsf{left} \cdot x_1 \hookrightarrow e_1 \mid \mathsf{right} \cdot x_2 \hookrightarrow e_2\} : \mathbf{A}}$$

$$\text{T:Prod}\ \frac{}{x_1 : a_1, x_2 : a_2; 0 \vdash \langle x_1, x_2\rangle : \langle a_1 \times a_2, 0\rangle} \qquad \text{T:Case:Prod}\ \frac{\Gamma, x_1 : a_1, x_2 : a_2; p \vdash e : \mathbf{A}}{x : a_1 \times a_2, \Gamma; p \vdash \mathsf{case}\ x\ \{\langle x_1, x_2\rangle \hookrightarrow e\} : \mathbf{A}} \qquad \text{T:List:Empty}\ \frac{}{\cdot; 0 \vdash [\,] : \langle L^{\vec{q}}(a), 0\rangle}$$

$$\text{T:List:Cons}\ \frac{\vec{q} = (q_1, \ldots, q_d)}{x_1 : a, x_2 : L^{\triangleleft(\vec{q})}(a); q_1 \vdash x_1 :: x_2 : \langle L^{\vec{q}}(a), 0\rangle}$$

$$\text{T:Case:List}\ \frac{\Gamma; p_0 \vdash e_1 : \mathbf{A} \qquad \vec{p} = (p_1, \ldots, p_d) \qquad \Gamma, x_1 : a, x_2 : L^{\triangleleft(\vec{p})}(a); p_1 + p_0 \vdash e_2 : \mathbf{A}}{x : L^{\vec{p}}(a), \Gamma; p_0 \vdash \mathsf{case}\ x\ \{[\,] \hookrightarrow e_1 \mid (x_1 :: x_2) \hookrightarrow e_2\} : \mathbf{A}}$$

$$\text{T:Fun}\ \frac{\forall(\langle a_1, q_1\rangle \to A_2) \in \mathcal{T}.\ (|\Gamma|, f : \mathcal{T}, x : a_1; q_1 \vdash e : A_2)}{\Gamma; 0 \vdash \mathsf{fun}\ f\ x = e : \langle \mathcal{T}, 0\rangle} \qquad \text{T:App}\ \frac{\Gamma; 0 \vdash f : \langle\langle a_1, p_1\rangle \to A_2, 0\rangle}{\Gamma, x : a_1; p_1 \vdash f\ x : A_2}$$

$$\text{T:Let}\ \frac{\Gamma_1; p_0 \vdash e_1 : \langle \mathbf{a}_1, p_1\rangle \qquad \Gamma_2, x : \mathbf{a}_1; p_1 \vdash e_2 : \mathbf{A}_2}{\Gamma_1, \Gamma_2; p_0 \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : \mathbf{A}_2} \qquad \text{T:Share}\ \frac{\mathbf{a} \curlyvee (\mathbf{a}_1, \mathbf{a}_2) \qquad \Gamma, x_1 : \mathbf{a}_1, x_2 : \mathbf{a}_2; q \vdash e : \mathbf{A}}{\Gamma, x : \mathbf{a}; q \vdash \mathsf{share}\ x\ \mathsf{as}\ x_1, x_2\ \mathsf{in}\ e : \mathbf{A}}$$

$$\text{T:Tick:Pos}\ \frac{q \geq 0}{\cdot; q \vdash \mathsf{tick}\ q : \langle \mathtt{unit}, 0\rangle} \qquad \text{T:Tick:Neg}\ \frac{q < 0}{\cdot; 0 \vdash \mathsf{tick}\ q : \langle \mathtt{unit}, -q\rangle}$$

Lst. 4.2: Syntax-directed typing rules of univariate AARA. The sharing relation $\mathbf{a} \curlyvee (\mathbf{a}_1, \mathbf{a}_2)$ used in T:Share is defined in Listing 4.3.

$$\text{SH:Unit}\ \frac{}{\texttt{unit} \curlyvee (\texttt{unit}, \texttt{unit})} \qquad \text{SH:Int}\ \frac{}{\texttt{int} \curlyvee (\texttt{int}, \texttt{int})} \qquad \text{SH:Prod}\ \frac{a_1 \curlyvee (a_{1,1}, a_{1,2}) \qquad a_2 \curlyvee (a_{2,1}, a_{2,2})}{(a_1 \times a_2) \curlyvee (a_{1,1} \times a_{2,1}, a_{1,2} \times a_{2,2})}$$

$$\text{SH:Sum}\ \frac{A_i = (a_i, q_i) \qquad a_i \curlyvee (a_{i,1}, a_{i,2}) \qquad q_i = q_{i,1} + q_{i,2} \qquad A_{i,j} = (a_{i,j}, q_{i,j}) \qquad (i = 1, 2, j = 1, 2)}{(A_1 + A_2) \curlyvee (A_{1,1} + A_{2,1}, A_{1,2} + A_{2,2})}$$

$$\text{SH:List}\ \frac{\vec{q} = \vec{q}_1 + \vec{q}_2 \qquad a \curlyvee (a_1, a_2)}{L^{\vec{q}}(a) \curlyvee (L^{\vec{q}_1}(a_1), L^{\vec{q}_2}(a_2))} \qquad \text{SH:Fun}\ \frac{}{(A_1 \to A_2) \curlyvee (A_1 \to A_2, A_1 \to A_2)} \qquad \text{SH:Set}\ \frac{}{\mathcal{T} \curlyvee (\mathcal{T}, \mathcal{T})}$$

Lst. 4.3: Sharing of resource annotations $\mathbf{a} \curlyvee (\mathbf{a}_1, \mathbf{a}_2)$.

$$\text{T:Sub}\ \frac{\Gamma; p \vdash e : \langle a, q \rangle \qquad a <: a'}{\Gamma; p \vdash e : \langle a', q \rangle} \qquad \text{T:Sup}\ \frac{\Gamma, x : a; p \vdash e : \mathbf{A} \qquad a' <: a}{\Gamma, x : a'; p \vdash e : \mathbf{A}} \qquad \text{T:Weak}\ \frac{\Gamma_1; q \vdash e : \mathbf{A}}{\Gamma_1, \Gamma_2; q \vdash e : \mathbf{A}}$$

$$\text{T:Relax}\ \frac{\Gamma; p_2 \vdash e : \langle \mathbf{a}, q_2 \rangle \qquad p_1 \geq p_2 \qquad p_1 - q_1 \geq p_2 - q_2}{\Gamma; p_1 \vdash e : \langle \mathbf{a}, q_1 \rangle}$$

Lst. 4.4: Structural rules of univariate AARA. The subtyping relation $a_1 <: a_2$ in T:Sub and T:Sup is defined in Listing 4.5.

$$\text{S:Unit}\ \frac{}{\texttt{unit} <: \texttt{unit}} \qquad \text{S:Int}\ \frac{}{\texttt{int} <: \texttt{int}} \qquad \text{S:Sum}\ \frac{a_i <: a_{i+2} \qquad q_i \geq q_{i+2} \qquad (i = 1, 2)}{\langle a_1, q_1 \rangle + \langle a_2, q_2 \rangle <: \langle a_3, q_3 \rangle + \langle a_4, q_4 \rangle} \qquad \text{S:Prod}\ \frac{a_1 <: a_3 \qquad a_2 <: a_4}{a_1 \times a_2 < a_3 \times a_4}$$

$$\text{S:List}\ \frac{\vec{q}_1 \geq \vec{q}_2 \qquad a_1 <: a_2}{L^{\vec{q}_1}(a_1) <: L^{\vec{q}_2}(a_2)} \qquad \text{S:Fun}\ \frac{A_3 <: A_1 \qquad A_3 <: A_4}{A_1 \to A_2 <: A_3 \to A_4}$$

Lst. 4.5: Subtyping relation $a_1 <: a_2$.

$$\text{V:Unit}\ \frac{}{\langle\,\rangle : \texttt{unit}} \qquad \text{V:Int}\ \frac{z \in \mathbb{Z}}{z : \texttt{int}} \qquad \text{V:Sum:L}\ \frac{v : a_1}{\mathsf{left} \cdot v : \langle a_1, q_1\rangle + A_2} \qquad \text{V:Sum:R}\ \frac{v : a_2}{\mathsf{right} \cdot v : A_1 + \langle a_2, q_2\rangle} \qquad \text{V:Prod}\ \frac{v_1 : a_1 \qquad v_2 : a_2}{\langle v_1, v_2\rangle : a_1 \times a_2}$$

$$\text{V:List:Empty}\ \frac{}{[\,] : L^{\vec{q}}(a)} \qquad \text{V:List:Cons}\ \frac{v_1 : a \qquad v_2 : L^{\vec{q}}(a)}{v_1 :: v_2 : L^{\vec{q}}(a)} \qquad \text{V:Fun}\ \frac{\mathsf{closure}(V; f, x.e) : \mathcal{T} \qquad (A_1 \to A_2) \in \mathcal{T}}{\mathsf{closure}(V; f, x.e) : A_1 \to A_2}$$

$$\text{V:Set}\ \frac{V : \Gamma \qquad \Gamma; 0 \vdash \mathsf{fun}\, f\, x = e : \langle \mathcal{T}, 0\rangle}{\mathsf{closure}(V; f, x.e) : \mathcal{T}} \qquad \text{V:Cont}\ \frac{\mathsf{dom}(V) = \mathsf{dom}(\Gamma) \qquad \forall x \in \mathsf{dom}(V).V(x) : \Gamma(x)}{V : \Gamma}$$

Lst. 4.6: Well-typed values and environments in univariate AARA.

Chapter 5

Related Work

This chapter reviews existing works related to the thesis. §5.1 discusses existing theoretical result about undecidability of resource analysis, language-based characterization of complexity classes, and language-based cost models. §5.2 discusses existing resource-analysis techniques, including static, data-driven, and interactive analyses.

5.1 Computability, Complexity, and Cost Models

This section discusses existing theoretical results about resource analysis.

Undecidability of resource analysis Various resource-analysis problems of Turing machines have been shown to be undecidable in the literature. Hájek [103] studies (un)decidability (and classification in the arithmetic hierarchy) of resource analysis of multi-tape Turing machines. Hájek [103] first proves that it is undecidable (more precisely, Π_1^0-complete) whether a multi-tape Turing machine halts within $n + 1$ steps on inputs of length n. Additionally, Hájek [103] shows that it is undecidable (more precisely, Σ_2^0-complete) whether a multi-tape Turing machines runs in polynomial time with any polynomial degree $d \in \mathbb{N}$ or a fixed polynomial degree $d \geq 1$.

Hoffmann [112] shows that it is undecidable whether a given Turing machine terminates in constant (and also polynomial) time on all input strings.

Gajser [88] organizes and extends existing undecidability results. Gajser [88] first introduces two classes of resource-analysis decision problems: (i) $\text{HALT}_{T(n)}$, where $T(n)$ is a fixed target symbolic bound (e.g., $2n + 3$); and (ii) $\text{HALT}_{\mathcal{F}}$, where $\mathcal{F}$ is a (possibly infinite) set of target bounds (e.g., all polynomials). Gajser [88] then derives necessary (and sometimes also sufficient) conditions on the symbolic bounds $T(n)$ and $\mathcal{F}$ for undecidability of resource analysis. Notably, Gajser [88] analyzes not only multi-tape Turing machines but also one-tape Turing machines, where the latter is more challenging to analyze than the former. One-tape Turing machines pose a unique technical challenge because multi-tape Turing machines can simulate a one-tape Turing machine while counting its number of steps at the same time, whereas one-tape Turing machines are not capable of such multi-tasking.

Although input Turing machines under resource analysis are allowed to be partial (i.e., possibly non-terminating) in the aforementioned works, it is sometimes more sensible to require input Turing machines to be total (i.e., terminating on all inputs). In this thesis, the restriction to total Turing machines is motivated by data-driven analysis, which requires Turing machines to terminate in order to collect cost measurements. Gajser [87] remarks that its undecidability proofs, which are for partial Turing machines, can be adapted to the setting of total Turing machines. However, it does not describe how the undecidability proofs should be modified.

Implicit computational complexity Implicit computational complexity (ICC) is a research field whose goal is to design programming languages that exactly characterize certain complexity classes (e.g., PTIME and PSPACE) by imposing syntactic restrictions on programs [65, 66]. These syntactic restrictions are *implicit* (hence the name implicit computational complexity) because they do not explicitly mention the resource usage of programs. To characterize a complexity class C, a programming language L must be designed such that (i) every program expressible in the language L belongs to the complexity class C; and (ii) every computable function in the complexity class C is simulated by some program in the language L.

Bellantoni and Cook [28] propose a programming language (dubbed **BC** [65]) that characterizes the set of polynomial-time computable functions. Built on the language of primitive recursive functions [142], the language **BC** restricts primitive recursion, which can lead to exponentially growing functions, by introducing two classes of function parameters: *normal* and *safe* parameters. Primitive recursion is allowed to be performed on normal arguments, but not safe arguments. In a recursive function, the output of a recursive call is a safe argument. Hence, we are not allowed to perform another (inner) recursion on the recursive call's output.

Hofmann [119] extends the language **BC** to a higher-order primitive recursion. In the language of primitive recursive functions [142], on which **BC** is based, all functions are first-order: they map natural numbers to natural numbers. In higher-order primitive recursion (as in System **T** of Gödel), the primitive-recursion operator is allowed to return higher-order functions. In the language **SLR**, normal and safe parameters are tracked by a type system equipped with a type modality $\Box$ from the modal logic S4. If we stop here, the resulting language contains exponential-time functions [65, 154]. To further trim the set of expressible programs, Hofmann [119] additionally imposes a linear constraint on the usage of higher-order functions.

Although the languages **BC** and **SLR** exactly capture polynomial-time *functions*, they cannot express some real-world polynomial-time *algorithms* (e.g., insertion sort [120]). This is because these algorithms apply (inner) primitive recursion to the output of the outer primitive recursion's recursive call, but such nested recursion is disallowed in **BC** and **SLR**.

To fix this issue, Hofmann [120] introduces a language commonly known as Linear Functional Programming Language (**LFPL**) in the literature [20, 121, 122]. To prune super-polynomial-time computation, **LFPL** ensures that all computation is non-size-increasing while permitting nested recursion. To prevent output sizes from growing excessively, **LFPL** uses a diamond type $\diamond$ to represent a resource, which is spawned by primitive recursion and can be used to pay for a type constructor. The idea of a type capturing resources has led to the development of a type system for inferring polynomial cost bounds, namely Automatic Amortized Resource Analysis (AARA) [113, 116, 122]. **LFPL** has been extended to EXPTIME by replacing primitive recursion

with general recursion [126].

Atkey [20] develops dependent type systems that capture polynomial-time computation. One of the two dependent type systems in Atkey [20] incorporates ideas from **LFPL**.

Another major line of research on ICC stems from linear logic [91]. In logics, computation is represented by proof reduction, which is a process of iteratively simplifying proofs. Linear-logic-based approaches to ICC differ from type-based approaches (e.g., **SLR** and **LFPL**) in how they avoid unfettered complexity explosion: the former controls duplication of function arguments, while the latter controls the depth of recursion [65, §2]. Soft linear logic by Lafont [149], light linear logic by Girard [92], light affine logic by Asperti and Roversi [18], and bounded linear logic by Girard et al. [93] all capture polynomial-time computation. Some linear logics have been translated to programming languages (e.g., Baillot and Mogbil [26], Gaboardi and Rocca [85] for PTIME and Gaboardi et al. [86] for PSPACE).

Language-based cost models Traditionally, computational costs (e.g., time and space) are defined on *machines* (e.g., Turing machines and the PRAM model [84]). However, for resource analysis of programs, it is more convenient to work with computational costs directly defined in *languages* (e.g., λ-calculi and programming languages). A language-based cost model is said to be *reasonable* if this model and Turing machine's cost model can simulate each other with a polynomial overhead in time and a constant overhead[1] in space [210].

Although a natural cost model for the vanilla λ-calculus is the number of β-reductions, it is non-trivial to prove that it is a reasonable cost model because substitutions in the λ-calculus lead to exponential size explosion of expressions [5]. A workaround is to encode λ-terms more compactly by allowing subexpressions to be shared. Blelloch and Greiner [34, 35] present a language-based cost model of the weak call-by-value λ-calculus, proving that it can be mapped to the cost model of the PRAM. Subsequent works extend language-based cost models to other evaluation strategies, such as strong call-by-value evaluation by Accattoli and Dal Lago [6], Accattoli et al. [7], Biernacka et al. [32]. Forster et al. [83] present a mechanized proof that the weak call-by-value λ calculus and Turing machines can simulate each other with a polynomial overhead in time. More recently, Accattoli et al. [8] present a language-based cost model that accounts for logarithmic space usage.

5.2 Resource Analysis

This section discusses existing resource-analysis techniques. They are classified into three categories: static analysis (§5.2.1), data-driven analysis (§5.2.2), and interactive analysis (§5.2.3).

5.2.1 Static Resource Analysis

This section reviews the literature of static resource analysis, which automatically infers symbolic cost bounds by analyzing the source code of input programs.

[1]While Slot and van Emde Boas [210] require the space overhead to be constant, Accattoli et al. [8] allow the space overhead to be linear.

AARA Automatic Amortized Resource Analysis (AARA) is a type-based resource-analysis technique that automatically infers polynomial cost bounds. AARA automates the potential method of amortized analysis [208, 209, 213], and AARA's types encode the amount of potential stored inside program variables. Type inference in AARA is fully automatic because all numeric constraints are linear and hence can be solved a linear-program (LP) solver (e.g., CLP [82]).

The development of AARA was inspired by the resource-capturing type $\diamond$ used to control size growth in **LFPL** [120, 121]. Hofmann and Jost [122] present the first version of AARA that infers linear bounds on heap space usage of first-order functional programs. Subsequently, Hoffmann and Hofmann [113] develop univariate polynomial AARA, and Hoffmann et al. [116] develop multivariate polynomial AARA. Resource-Aware ML (RaML) [117, 118] is an implementation of multivariate polynomial AARA for analyzing OCaml programs. AARA has been adapted to infer exponential bounds [135, 136] and logarithmic bounds [126, 156, 157]. However, it has remained a challenge to develop an AARA-like type system that can automatically infer symbolic bounds where polynomials and logarithm coexist (e.g., $O(n \log n)$ for MergeSort). Hoffmann and Jost [114] give a survey on AARA.

AARA is the only type-based resource analysis capable of automatic bound inference. Other type-based approaches that are capable of automatic *verification*, but not automatic *inference*, are discussed in §5.2.3.

Numerous advanced programming features have been incorporated into AARA. Examples include refinement types [143, 144], higher-order functions [134], side effects [158], lazy evaluation [205, 217], and recursive types [99]. Beyond functional programming, AARA has been extended to various programming paradigms: imperative programs [43], parallel programs [115], probabilistic programs [157, 177, 220], and session-typed concurrent programs [69, 70].

Hybrid AARA [188] (§7) builds on AARA. The type system of AARA is extended such that user-specified code fragments are allowed to be treated as black boxes (i.e., leaves in typing trees). These code fragments are analyzed by non-AARA resource analysis (e.g., data-driven resource analysis) to infer their resource-annotated types, which are then plugged into the typing tree inferred by AARA. Thus, resource-annotated types serve as an interface between two constituent analysis techniques combined by Hybrid AARA. A technical challenge in Hybrid AARA is that resource-annotated types inferred by data-driven analysis (e.g., Bayesian data-driven analysis) must respect the linear constraints produced by AARA's type system.

Resource decomposition (§8) addresses a drawback of polynomial AARA and Hybrid AARA: they can only automatically infer polynomial cost bounds. By integrating static and data-driven analyses, resource decomposition achieves automatic inference of cost bounds where polynomials and logarithm coexist (e.g., $O(n \log n)$ for MergeSort), albeit at the expense of soundness. Automatic inference of such bounds has been an open challenge in purely static resource analysis. ATLAS [126, 156, 157] is a variant of AARA that automatically infers logarithmic cost bounds (i.e., $O(\log n)$) for splay-tree operations. However, ATLAS cannot infer more sophisticated cost bounds such as $O(n^2 \log n)$, because ATLAS uses potential functions tailored to splay trees. More generally, it is an open challenge to design a family of potential functions that (i) admit both polynomials and logarithm; (ii) are closed under some operations (e.g., resource-annotation sharing and constructors and destructors of data types); and (iii) are amenable to automated reasoning. To overcome this challenge, resource decomposition decomposes resource analysis into two analysis techniques, where one of them (e.g., data-driven analysis) infers log-

arithmic parts of an overall cost bound.

Recurrence relations Recurrence relations are equations defining the symbolic cost bound of a function f in terms of the symbolic cost bounds of f's recursive calls. Wegbreit [223], the first to study static resource analysis, adopts a recurrence-relation-based method. Grobauer [97] describes how to automatically extract recurrence relations from Dependent ML programs [231]. Program-analysis tools that use recurrence relations include CiaoPP [108, 159, 201, 202] and COSTA [9, 10, 11]. The correctness of recurrence-relation extraction has been formulated and proved using logical relations in Cutler et al. [64], Danner et al. [68], Kavvos et al. [139]. Solving recurrence relations has been investigated in Breck et al. [37], Farzan and Kincaid [77], Kincaid et al. [140, 141].

Term rewriting Term rewrite systems are a low-level programming model where terms are repeatedly transformed according to rewrite rules. Term rewrite systems serve as an abstract and unifying model of functional programming free of language-specific details. Resource analysis of term rewrite systems has been investigated [22, 23, 123, 174]. A resource-analysis tool TcT [24] first translates programs (e.g., OCaml functional programs and object-oriented bytecode) to term rewrite systems and then conducts resource analysis.

Ranking functions Given a loop or a recursive function, a ranking function is a numeric function that is bounded below and strictly decreases after each iteration or each recursive call. Although ranking functions are typically derived to prove termination, they can also be used to derive cost bounds. Existing works include KoAT [38, 39, 90], Loopus [206], and CoFloCo [81]. Chatterjee et al. [53] investigate inference of non-polynomial bounds from ranking functions. COSTA [9, 10, 11] uses ranking functions to solve recurrence relations.

Invariant generation Invariants are logical formulas that always hold at certain program points every time they are visited. Analysis methods developed by Gulavani and Gulwani [101], Gulwani et al. [102], Zuleger et al. [235] obtain invariants by abstract interpretation and extract symbolic cost bounds from the invariants.

SPEED [102] leverages *counters* to statically infer symbolic cost bounds of imperative programs. Counters track the number of times we follow back-edges in a call graph (i.e., the number of loop iterations and recursive calls) at runtime. SPEED employs a linear-invariant generation tool to infer linear bounds on the counters. An overall cost bound is obtained by summing and multiplying the counters' linear bounds appropriately.

Although counters in SPEED and resource guards in resource decomposition (§8) are similar, they have two differences. Firstly, counters in SPEED only track back-edges in a call graph, while resource guards in resource decomposition can track any custom quantities (e.g., recursion depths of functions) as long as they can be expressed as high-water marks of annotations mark and unmark (§8.1.1). Secondly, SPEED requires counters' symbolic bounds to be linear due to its reliance on linear-invariant generation tools, while resource guards in resource decomposition can be substituted with any symbolic bounds, including non-polynomial bounds.

5.2.2 Data-Driven Resource Analysis

This section reviews the literature of data-driven resource analysis. It first runs an input program on many inputs to construct a dataset of cost measurements and then statistically infers a symbolic cost bound from the dataset.

Experimental algorithmics Pioneered by Mcgeoch [167], experimental algorithmics [73, 133, 165, 166, 173, 200] is a research field that concerns the inference of algorithmic complexity, particularly asymptotic complexity, from experiment data. The goal of experimental algorithmics is to supplement pen-and-paper analysis of asymptotic complexity of algorithms with their empirical analysis.

Data-driven resource analysis Program profilers that run an input program on many inputs to infer a symbolic cost bound include the trend profiler [94], algorithmic profiling [234], and input-sensitive profiling [60]. Huang et al. [127] present data-driven resource analysis that first automatically extracts a large number of features from an input program and then performs sparse polynomial regression to infer a polynomial cost bound in terms of a few features. Rogora et al. [195] develop a language of symbolic cost bounds that come with deterministic and probabilistic conditionals.

Chambon et al. [51] apply optimization-based data-driven resource analysis to Python programs to construct a coding benchmark suite BigO(Bench). The benchmark suite contains (i) coding problem descriptions collected from an online competitive programming platform; (ii) human solutions; and (iii) their complexity (in terms of running time and space usage). BigO(Bench) is used to evaluate the ability of large language models (LLMs) to reason about program complexity and synthesize code with desirable complexity.

Instead of treating input program as black boxes, data-driven resource analysis can be augmented with information from the source code. For example, Demontiê et al. [74] present a more fine-grained data-driven analysis technique than input-sensitive profiling [60]. Given an imperative program P, instead of performing data-driven analysis on the *entire* program P, their technique performs data-driven analysis on *each loop* in the program P. An overall cost bound of the program P is the sum of individual loops' cost bounds.

Dynaplex [131] first infers a recurrence relation of a recursive program from its execution traces by optimization and then solves the recurrence relation by the master theorem.

Rustenholz et al. [197, 198] use machine learning (specifically sparse linear regression and evolutionary-search-based symbolic regression) to guess candidate solutions to recurrence relations and then formally verify them by SMT solvers.

These data-driven techniques all use optimization (e.g., polynomial regression) to statistically infer symbolic bounds. Furthermore, these techniques do not let the user adjust how conservative inferred symbolic bounds are based on the user's domain knowledge.

5.2.3 Interactive Resource Analysis

This section reviews the literature of interactive resource analysis. The user first provides a candidate cost bound. The bound can be either automatically verified by SMT solvers (e.g.,

Z3 [72] is used in TiML [222]) or interactively verified by collaborating with a proof assistant (e.g., Agda [182] is used in **calf** [180], and Coq [29] is used in Iris with time credits [52, 168]).

Automatic verification Among type-based resource analyses, AARA is the only one capable of automatic bound inference. Other type-based analyses are capable of automatic verification, but lack automatic inference. Such analyses all use refinement types or indexed types annotated with information about resources. Examples include typed assembly language [62], TiML [222], Liquid Haskell [104], liquid resource types [143, 144], and sized types [21, 218].

Interactive verification The literature offers a large body of work on manual (or semi-automatic) inference and verification of cost bounds. For example, potential-based reasoning has been integrated with (concurrent) separation logic [19, 52, 168, 172] and higher-order logic [178]. Other techniques are based on dependent types [67, 98, 150, 180]. Relational cost analysis [54, 55] is a refinement-type system for reasoning about symbolic bounds on a difference in costs between two programs.

In a cost-aware logical framework (**calf**) [180], a program is instrumented with an extra input variable, called a *clock*, to track the recursion depth. Although clocks in **calf** and resource guards in resource decomposition (§8) behave similarly, they have two major differences. Firstly, clocks are motivated by the need to ensure termination of programs in total type theory, while resource guards are motivated by the desire to integrate complementary analysis methods. Secondly, clocks track recursion depths, while resource guards track a wider variety of user-specified quantities.

5.3 Input Generation

This section reviews the literature on worst-case input generation and property-based testing.

5.3.1 Worst-Case Input Generation

Given a program, worst-case inputs refer to program inputs that trigger the worst-case costs of the program, given a fixed (but possibly large) input size. Many techniques have been developed to identify worst-case inputs, with or without soundness guarantees. These techniques are classified into two families:

1. Techniques that identify worst-case execution paths by symbolic execution and obtain the corresponding worst-case program inputs by SMT solving; and
2. Techniques that perform fuzzing to explore the space of program inputs and return inputs with the highest costs.

Symbolic execution Burnim et al. [42] propose WISE, the first technique for worst-case input generation. Given a target program P, WISE first exhaustively explores the space of small program inputs by symbolic execution, identifying execution paths that incur the highest cost. These execution paths are then generalized into a pattern G (called a branch-policy generator)

that determines which branch of an if-else statement in the source code should be taken in order to achieve the worst-case cost. The branch dictated by the generator G must be taken whenever the same if-else statement is encountered during program execution and both branches are feasible. Next, for a larger input size, the program P is symbolically executed while following an execution path π according to the generator G. Finally, the execution path π is solved using an SMT solver.

SPF-WCA [160] extends WISE by extending generators from branch policies to path policies. Unlike branch policies in WISE [42], path policies in SPF-WCA [160] take into account the history of past branch choices.

Wang and Hoffmann [219] develop a type-based approach to worst-case input generation. Unlike WISE [42] and SPF-WCA [160], whose outputs are not guaranteed to be worst-case inputs, type-guided worst-case input generation [219] guarantees that the returned program inputs are worst-case inputs. In this approach, given a program P, Conventional AARA is performed to derive a resource-annotated typing tree of the program P. For a fixed input size, the program P is symbolically executed while traversing the resource-annotated typing tree. The goal of symbolic execution is to find an execution path π where no potential is discarded according to the typing tree. Once such an execution path π is discovered, a corresponding program input is derived by solving π's path constraints.

Fuzzing Resource-guided fuzzing runs a genetic algorithm in the space of program inputs, where the objective is to maximize the computational cost of a target program. Being heuristic-based, fuzzing does not guarantee its outputs to be worst-case program inputs. SlowFuzz [185] is the first fuzzer that aims to generate worst-case program inputs. Subsequent works on resource-guided fuzzing include PerfFuzz [155], Saffron [152], and MemLock [226]. Badger [181] is a hybrid approach to worst-case input generation, combining symbolic execution and fuzzing.

Wu and Wang [230] observe that worst-case input generation of a target program P is equivalent to maximum-a-posteriori (MAP) estimation of a probabilistic model/program π whose density is equal to the cost of the program P. They develop DSE-SMC, which combines Sequential Monte Carlo (SMC) with an evolutionary algorithm (e.g., a genetic algorithm).

Pattern fuzzing Wei et al. [224] extend resource-guided fuzzing from fixed-size program inputs to patterns of variable-length inputs (e.g., sorted lists and lists of identical elements), resulting in so-called pattern fuzzing. They first present a domain-specific language (DSL) of program-input generators, which are programs producing values of varying sizes. The output values of a generator are used to collect cost measurements of a target program and infer its asymptotic complexity. Wei et al. [224] then develop a pattern fuzzer called Singularity. Their empirical evaluation shows that pattern fuzzing is more scalable than fuzzing of fixed-size data structures as target input sizes grow larger.

Although the generators developed in §9 also adopt the idea of pattern fuzzing, they differ from the generators in Singularity [224] in three aspects. Firstly, the generators in §9 are probabilistic: every time they are executed, they can generate different values of a user-specified size. In fact, the probabilistic generators have a small yet positive probability of deviating from the standard behavior: the generators instead sample integers from a broad interval. That is,

once a target size is fixed, any value of that size has a positive probability of being produced by probabilistic generators. On the other hand, Singularity's generators are deterministic: they always produce the same values every time they are executed. Secondly, the generators in §9 can handle all algebraic data types, whereas the generators in Singularity only support integers, lists, and graphs by default. As a consequence of this difference in the supported types, the two kinds of generators differ in how they build larger values from smaller ones. The generators in §9 build larger values using data constructors, while Singularity's generators do so using operators (e.g., arithmetic operations, length of lists, append, and list concatenation). Lastly, the generators in §9 can generate values of any target size as long as it is feasible. By contrast, Singularity's generators may only be able to generate data structures of certain sizes (e.g., lists of odd lengths [224, §2.2]).

5.3.2 Property-Based Testing

Property-based testing (PBT) is a program-testing paradigm where a user writes a program specification of the form $Q_1(x) \implies Q_2(y)$, where a precondition $Q_1(x)$ (x is an input) and a postcondition $Q_2(y)$ (y is an output) are written in the same programming language as the program under test [95, 151]. Given a program $P(x)$ under test, a PBT tool generates many program inputs, checking whether the specification $Q_1(x) \implies Q_2(y)$ holds for each input-output pair $(x, P(x))$.

PBT was pioneered by the tool QuickCheck developed by Claessen and Hughes [57] for Haskell. QuickCheck [57] offers built-in generators for some types (e.g., integer lists), but it also allows a user to define custom generators.

The generators developed in §9 are different from QuickCheck in their expressiveness. §9 defines a (fairly restrictive) language of generators for an arbitrary target algebraic data type. An optimization algorithm (specifically a genetic algorithm) automatically searches this space of generators for a worst-case-input generator. Meanwhile, QuickCheck does not offer a generator out of the box for an arbitrary data type, especially a user-defined one. Claessen and Hughes [57] state that, although they could use polytypic typing to automatically synthesize a generator for an arbitrary algebraic data type, they choose not to do so. This is because it is difficult to automatically synthesize a high-quality generator (i.e., a generator that satisfies a precondition $Q_1(x)$ with a high probability). In return, QuickCheck lets the user define custom generators in Haskell, which is a Turing-complete language. Thus, QuickCheck is more expressive than the language of generators in §9.

The second difference between the generators in §9 and QuickCheck is how the user specifies a target size. In §9, the size of a value is given by the number of data constructors (i.e., $\text{fold}_t(\cdot)$ for a type name t). Meanwhile, QuickCheck does not formally define the size of a value—the user defines an appropriate size measure and incorporates it into a custom generator.

Chapter 6

Expressiveness of Resource Analysis

This chapter discusses negative and positive theoretical results on the expressiveness of resource analysis. It is natural to ask whether resource analysis can analyze all programs in a Turing-complete programming language. To study (un)decidability of resource analysis, I first frame it as a decision problem in three different ways. Resource analysis, when viewed as a decision problem, has been proved to be undecidable in the literature [88, 103, 112]. Hence, for any resource-analysis technique, if it is sound, then it must be *incomplete*. Furthermore, I prove a stronger result: resource analysis remains undecidable even if we assume that input programs terminate on all inputs (i.e., we can run programs on any input and measure the resource usage when the programs terminate).

§6.1 sets the stage by defining Turing machines and halting problems. §6.2 then frames resource analysis as decision problems. Finally, §6.3 presents existing results on the undecidability of resource analysis [88, 103, 112] and proves stronger undecidability results when resource analysis are only applied to total Turing machines.

Next, §6.4 shows that a typable fragment of static resource analysis AARA is *polytime complete*: for any polytime function f, there exists a program P_f with the same input-output behavior and cost as the function f such that the program P_f is typable in AARA. The proof assumes that the polynomial time bound of the function f is known. A key proof idea is to encode a working tape of a Turing machine as a list of length $f(n)$, where n is the input length, in such a way that the list can be typed in AARA. The idea of encoding a cost bound as a list also appears in the development of the second hybrid resource analysis, resource decomposition (§8).

The word "completeness" in *incompleteness* (§6.3) and *polytime completeness* (§6.4) refers to properties of two different objects: the former is a property of resource-analysis techniques, while the latter is a property of a class of programs.

6.1 Setting the Stage

This section sets the stage by first defining one-tape and multi-tape Turing machines and then introducing halting problems. I introduce both one-tape and multi-tape Turing machines because halting problems are formulated using one-tape Turing machines, while decision problems of resource analysis are formulated using multi-tape Turing machines (§6.2). In §6.3, I

reduce a halting problem to resource-analysis decision problems, thereby proving their undecidability.

One-tape Turing machines Turing machines are used as a computational model to formulate and prove undecidability of resource analysis. I first introduce one-tape Turing machines, which are used to formulate halting problems.

Definition 6.1.1 (One-tape Turing machine [207]). *A one-tape Turing machine M is a 7-tuple*

$$M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}}), \tag{6.1.1}$$

where

- *Q is a finite set of states.*
- *Σ is a finite alphabet of input symbols, and $\Gamma \supseteq \Sigma \cup \{\bot\}$ is a finite alphabet of tape symbols The tape alphabet Γ contains all input symbols Σ and the blank symbol $\bot \notin \Sigma$.*
- *$\delta : Q \times \Gamma \to Q \times \Gamma \times \{\text{left}, \text{right}\}$ is the transition function.*
- *$q_0 \in Q$ is the initial state; $q_{\text{accept}} \in Q$ is the accept state; and $q_{\text{reject}} \in Q$ is the reject state. These three states are all distinct.*

A one-tape Turing machine M comes with a semi-infinite read-write tape. In the initial configuration, an input string $w \in \Sigma^*$ is placed on the tape (to the left of the semi-infinite tape). Also, the initial state of the machine is q_0, and the read/write head is positioned on the first cell of the tape (i.e., on the first symbol of the input string w). The rest of the tape is filled with the blank symbol $\bot \in \Gamma$.

The one-tape Turing machine M runs as follows. In each step, let $a \in \Sigma$ be the character in the cell currently under the tape head and $q \in Q$ be the current state $q \in Q$ of the machine. The machine then overwrites the current cell (if necessary), updates the machine's state, and moves the head to the left or right according to the output $\delta(a, q)$ of the transition function. This step is repeated until the machine enters either $q_{\text{accept}} \in Q$ or $q_{\text{reject}} \in Q$. If the accept state q_{accept} is reached, the input word w is accepted. If the reject state q_{reject} is reached, the input word w is rejected. It is possible that the Turing machine runs forever, in which case the input w is neither accepted nor rejected.

The running time of a Turing machine M for an input w is given by the number of steps the machine M makes on the input w before termination (if it does).

Multi-tape Turing machines Next, I introduce multi-tape Turing machines, which operate on a read-only input tape and (one or more) read-write working tapes. Multi-tape Turing machines are used to formulate decision problems of resource analysis (§6.2).

Definition 6.1.2 (Multi-tape Turing machine [207]). *A multi-tape Turing machine M is a 7-tuple*

$$M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}}), \tag{6.1.2}$$

just like in one-tape Turing machines (Defn. 6.1.1), except that the multi-tape Turing machine M operates on $k \geq 2$ tapes. One tape is a read-only input semi-infinite tape that stores an input $w \in \Sigma^*$, and the remaining $k - 1$ tapes are read-write working semi-infinite tapes. The transition function is now $\delta : Q \times \Gamma^k \to Q \times \Gamma^{k-1} \times \{\text{left}, \text{right}\}^{k-1}$, where k is the number of tapes (including

the read-only input tape). Multi-tape Turing machines operate on all tapes (i.e., writing to them and moving their tape heads) in a single step.

Multi-tape Turing machines do not generalize one-tape Turing machines. The former has a read-only input tape and (one or more) read-write working tapes, while the latter has a single tape that stores the input and also acts as a working tape. It is possible to show that one-tape and multi-tape Turing machines are equally expressive because they can simulate each other, although their simulation does not preserve running time.

Halting problems A standard halting problem is to decide whether a given one-tape Turing machine M halts on a given input w:

$$\mathrm{HALT}^1 \coloneqq \{\langle M, w\rangle \mid \text{one-tape Turing machine } M \text{ halts on input } w\}. \tag{6.1.3}$$

HALT^1 is undecidable [207, §5.1]. That is, there exist no Turing machines that accept every $\langle M, w\rangle \in \mathrm{HALT}^1$ and reject every $\langle M, w\rangle \notin \mathrm{HALT}^1$.

To prove that a decision problem is undecidable, a common strategy is to reduce a known undecidable problem (e.g., HALT^1) to the problem at hand. To prove undecidability of resource analysis (§6.3), I reduce the following variant of the halting problem, which is also undecidable, to resource analysis's decision problems.

Theorem 6.1.1 (Variant of the halting problem)**.** *Consider a variant of the halting problem:*

$$\mathrm{HALT}^1_\epsilon \coloneqq \{\langle M\rangle \mid \textit{one-tape Turing machine } M \textit{ halts on input } \epsilon\}, \tag{6.1.4}$$

where $\langle M\rangle$ denotes an encoding of a Turing machine M and ϵ is the empty string. The language HALT^1_ϵ is undecidable.

Thm. 6.1.1 is proved by reducing the standard halting problem HALT^1 (6.1.3) to HALT^1_ϵ.

6.2 Decision Problems of Resource Analysis

To study (un)decidability of resource analysis, I frame it as a decision problem. Resource analysis (e.g., AARA [112, 113, 116]) is a *computational problem* where answers are symbolic cost bounds encoded in some format (e.g., coefficients of polynomial cost bounds). Meanwhile, decidability is only defined for *decision problems* where answers are yes or no.

Without loss of generality, the resource metric of interest is the running time of Turing machines. The undecidability of resource analysis under the time metric can be extended to the tick metric, which is more general than the time metric.

6.2.1 Partial Computation

Resource analysis can be framed as a decision problem in three ways. In all formulations, a resource-analysis decision problem asks whether an input Turing machine M has a symbolic bound $f(|w|)$ on the running time for all input strings w. Here, Turing machines represent *partial computation*: they are allowed to run forever. The difference between the three formulations lies in how target symbolic bounds are chosen: the first formulation fixes a single target bound, while the second formulation considers a set of target bounds, and the third formulation treats a target bound as an input.

First formulation We first fix a symbolic cost bound $f(|w|)$ (e.g., $2|w|+3$), where $|w|$ denotes the length of an input string w to a Turing machine. The decision problem asks, given a Turing machine M, whether it will halt within $f(|w|)$ steps for all input strings w.

Definition 6.2.1 (First resource-analysis decision problem)**.** *Given a symbolic bound $f : \mathbb{N} \to \mathbb{Q}_{\geq 0}$, a resource-analysis decision problem* $\text{HALT}_f^{\text{par}}$ *is defined as*

$$\text{HALT}_f^{\text{par}} \coloneqq \{\langle M \rangle \mid \forall w \in \Sigma^*.\textit{multi-tape TM } M \textit{ halts within } f(|w|) \textit{ steps on input } w\}, \quad (6.2.1)$$

where Σ is an input alphabet of Turing machines (TM). Note that this decision problem concerns multi-tape Turing machines as opposed to one-tape ones.

Existing resource-analysis techniques, which infer symbolic bounds, can be retrofitted to answer the decision problem $\text{HALT}_f^{\text{par}}$. For example, for AARA, if a symbolic bound $f(|w|)$ is a polynomial bound of degree $d \in \mathbb{N}$, we run AARA on an input program with the polynomial degree d. If AARA returns a polynomial bound $g(|w|)$, we compare the coefficients of polynomials f and g. If g is smaller than or equal to f in all coefficients, then it means that $f(|w|)$ is a sound cost bound of an input program/Turing machine, thanks to the soundness guarantee of AARA (Thm. 4.3.1). Otherwise, if AARA fails to infer any polynomial bound g or the polynomial g inferred by AARA is larger than the polynomial f in some coefficient, then we report no to the decision problem (although we cannot draw any definitive conclusions about the input program/Turing machine).

Second formulation In the second formulation of resource analysis as a decision problem, we fix a (possibly infinite) set F of symbolic cost bounds of type $\mathbb{N} \to \mathbb{Q}_{\geq 0}$. The decision problem then asks whether an input Turing machine M has some symbolic cost bound $f(|w|) \in F$ on the running time. Thus, the second formulation subsumes the first formulation $\text{HALT}_f^{\text{par}}$ (Defn. 6.2.1) by setting $F \coloneqq \{f\}$. Examples of the set F include (i) all constant (i.e., degree-zero polynomial) bounds; (ii) all polynomials of a fixed degree $d \in \mathbb{N}$; and (iii) all polynomials (of any degree).

Definition 6.2.2 (Second resource-analysis decision problem)**.** *Given a (possibly infinite) set F of symbolic bounds $f : \mathbb{N} \to \mathbb{Q}_{\geq 0}$, a resource-analysis decision problem* $\text{HALT}_F^{\text{par}}$ *is*

$$\begin{aligned}\text{HALT}_F^{\text{par}} \coloneqq \{\langle M \rangle \mid &\exists f \in F.\forall w \in \Sigma^*.\textit{multi-tape TM } M \textit{ halts within} \\ &f(|w|) \textit{ steps on input } w\},\end{aligned} \quad (6.2.2)$$

where Σ is an input alphabet of Turing machines (TM).

Again, we can retrofit existing resource-analysis techniques to answer the decision problem $\text{HALT}_F^{\text{par}}$. For example, in AARA, suppose the user specifies a polynomial degree $d \in \mathbb{N}$. If AARA successfully infers a polynomial bound, then we return yes to the decision problem. Otherwise, if AARA fails to infer a polynomial bound (i.e., linear constraints collected from AARA's type system are infeasible), we return no to the decision problem.

Third formulation The third formulation is identical to the first one $\text{HALT}_f^{\text{par}}$ (Defn. 6.2.1), except that a target symbolic bound $f : \mathbb{N} \to \mathbb{Q}_{\geq 0}$ is given as an extra input of the decision

problem. Here, we assume that a target symbolic bound f can be encoded in some way. For example, polynomial bounds can be encoded by their coefficients, and computable functions can be encoded by their corresponding total (i.e., terminating) Turing machines.

Definition 6.2.3 (Third resource-analysis decision problem)**.** *Consider a (possibly infinite) set F of symbolic cost bounds $f : \mathbb{N} \to \mathbb{Q}_{\geq 0}$. A resource-analysis decision problem* $\mathrm{HALT}^{\mathrm{par}}_{\mathrm{input},F}$ *is defined as*

$$\begin{aligned}\mathrm{HALT}^{\mathrm{par}}_{\mathrm{input},F} := \{\langle M, f\rangle \mid & f \in F, \forall w \in \Sigma^*. \textit{multi-tape TM } M \textit{ halts within} \\ & f(|w|) \textit{ steps on input } w\},\end{aligned} \tag{6.2.3}$$

where Σ is an input alphabet of Turing machines (TM).

All resource-analysis decision problems in Defns. 6.2.1–6.2.3 are undecidable under some conditions on target cost bounds f and F (§6.3.1).

6.2.2 Total Computation

In the decision problems 6.2.1–6.2.3, input Turing machines represent partial computation (i.e., possibly non-terminating computation). These decision problems on partial computation have been considered in existing works [88, 103, 112].

However, it is sometimes sensible to require that input Turing machines should be *total* (i.e., they terminate). This requirement is motivated by data-driven resource analysis. Data-driven analysis runs an input program $P(x)$ on many inputs $x_1, \ldots, x_N$ and constructs a dataset $\mathcal{D}$ of P's cost measurements. To record cost measurements, the program P must terminate on all inputs (or at least those inputs used for constructing the dataset $\mathcal{D}$). Therefore, data-driven resource analysis is only applicable to terminating programs. In the presence of data-driven analysis, an interesting theoretical question is: is resource analysis decidable if we assume that data-driven analysis is applicable (i.e., input programs are guaranteed to terminate)? Put differently, does the ability to run programs on any inputs and measure their cost measurements upon termination buy us any additional power in resource analysis?

To this end, I introduce total computation's counterparts of the resource-analysis decision problems 6.2.1–6.2.3.

Definition 6.2.4 (First resource-analysis decision problem for toal computation)**.** *Given a symbolic bound $f : \mathbb{N} \to \mathbb{Q}_{\geq 0}$, a resource-analysis decision problem* $\mathrm{HALT}^{\mathrm{tot}}_{f}$ *is defined as*

$$\begin{aligned}\mathrm{HALT}^{\mathrm{tot}}_{f} := \{\langle M\rangle \mid & \forall w \in \Sigma^*. \textit{total multi-tape TM } M \textit{ halts within} \\ & f(|w|) \textit{ steps on input } w\},\end{aligned} \tag{6.2.4}$$

where Σ is an input alphabet of Turing machines (TM).

Definition 6.2.5 (Second resource-analysis decision problem for total computation)**.** *Given a (possibly infinite) set F of symbolic bounds $f : \mathbb{N} \to \mathbb{Q}_{\geq 0}$, a resource-analysis decision problem* $\mathrm{HALT}^{\mathrm{tot}}_{F}$ *is*

$$\begin{aligned}\mathrm{HALT}^{\mathrm{tot}}_{F} := \{\langle M\rangle \mid & \exists f \in F. \forall w \in \Sigma^*. \textit{total multi-tape TM } M \textit{ halts within} \\ & f(|w|) \textit{ steps on input } w\},\end{aligned} \tag{6.2.5}$$

where Σ is an input alphabet of Turing machines (TM).

Definition 6.2.6 (Third resource-analysis decision problem for total computation)**.** *Consider a (possibly infinite) set F of symbolic cost bounds $f : \mathbb{N} \to \mathbb{Q}_{\geq 0}$. A resource-analysis decision problem* $\text{HALT}^{\text{tot}}_{\text{input},F}$ *is defined as*

$$\begin{aligned}\text{HALT}^{\text{tot}}_{\text{input},F} \coloneqq \{\langle M, f\rangle \mid & f \in F, \forall w \in \Sigma^*. \textit{total multi-tape TM } M \textit{ halts within} \\ & f(|w|) \textit{ steps on input } w\},\end{aligned} \tag{6.2.6}$$

where Σ is an input alphabet of Turing machines (TM).

The restriction of input Turing machines to total computation render undecidable problems problems decidable. For instance, the halting problem HALT^1 is trivially decidable (i.e., the answer is always yes) if we assume that input Turing machines are guaranteed to terminate. Similarly, it is decidable whether a given Turing machine M returns a desirable output on a given input w: it suffices to run the Turing machine M on the input w until it terminates.

Nonetheless, resource analysis is undecidable (under some conditions on target cost bounds f and F) even with the assumption of total computation (§6.3.2). This is because, to check if a program P has a certain cost bound, we must test the program P on all inputs x. Due to infinitely many possible inputs, it is computationally infeasible to test all inputs even if the program P terminates on all of them.

6.2.3 Resource Analysis with Fixed Program Inputs

Throughout this thesis, resource analysis concerns symbolic cost bounds $f(x)$ that are parametric in program inputs x. In resource analysis as a *computational* problem, the goal is to infer symbolic bounds, and in resource analysis as a *decision* problem, the goal is to check whether a given program/Turing machine has a certain symbolic bound. Here, symbolic cost bounds are not properties of programs on *fixed* program inputs—the bounds $f(x)$ must be sound across *all* possible inputs x.

Alternatively, one could consider a different variant of resource analysis where program inputs are fixed. Fixing program inputs simplifies resource analysis, rendering some resource-analysis decision problems decidable. For example, consider a variant of the decision problem $\text{HALT}^{\text{par}}_f$ (Defn. 6.2.1) where a program input is provided as an input of the decision problem:

$$\text{BOUNDED}^{\text{par}}_f \coloneqq \{\langle M, w\rangle \mid \text{multi-tape TM } M \text{ halts within } f(|w|) \text{ steps on input } w\}. \tag{6.2.7}$$

Assuming that the target symbolic bound $f : \mathbb{N} \to \mathbb{Q}_{\geq 0}$ is computable, the decision problem $\text{BOUNDED}^{\text{par}}_f$ is decidable. To solve the decision problem, we first evaluate $f(|w|)$ for a given input string w and then run an input Turing machine M for $f(|w|)$ many steps. Likewise, a similar variant of the decision problem $\text{HALT}^{\text{par}}_{\text{input},F}$ (Defn. 6.2.3) is decidable.

Some resource-analysis decision problems remain undecidable even if we fix program inputs. For example, consider a variant of the second decision problem (Defn. 6.2.2):

$$\begin{aligned}\text{BOUNDED}^{\text{par}}_F \coloneqq \{\langle M, w\rangle \mid & \exists f \in F. \text{multi-tape TM } M \text{ halts within} \\ & f(|w|) \text{ steps on input } w\}.\end{aligned} \tag{6.2.8}$$

If the set F of symbolic bounds contains arbitrarily large constant functions, then the decision problem $\text{BOUNDED}_F^{\text{par}}$ is equivalent to the halting problem of multi-tape Turing machines (i.e., (6.1.3) adapted to multi-tape Turing machines). Hence, $\text{BOUNDED}_F^{\text{par}}$ is also undecidable.

6.3 Undecidability of Resource Analysis

This section proves undecidability of resource analysis. §6.3.1 first presents existing undecidability results in the literature where input Turing machines are partial (i.e., possibly nonterminating). §6.3.2 then provides stronger undecidability results where input Turing machines are total (i.e., terminating). Finally, §6.3.3 discusses soundness and completeness of static, data-driven, and interactive resource analyses.

6.3.1 Partial Computation

Existing works have proved undecidability of resource-analysis decision problems for partial computation (§6.2.1). Hájek [103] shows that (i) $\text{HALT}_{n+1}^{\text{par}}$ is undecidable; and (ii) $\text{HALT}_F^{\text{par}}$ is undecidable if F is the set of polynomials (of any degree) or the set of polynomials of a fixed degree $d \geq 1$. Hoffmann [112, §1.2] shows that $\text{HALT}_{O(1)}^{\text{par}}$ (i.e., constant bounds) and $\text{HALT}_{\text{P}}^{\text{par}}$ (i.e., polynomial bounds) are undecidable by reducing the halting problem $\text{HALT}_{\epsilon}^{1}$. Gajser [87, 88] proves more general undecidability theorems on the resource-analysis decision problems $\text{HALT}_f^{\text{par}}$ (Defn. 6.2.1) and $\text{HALT}_F^{\text{par}}$ (Defn. 6.2.2) with some conditions on the target symbolic bounds f and F.

Theorem 6.3.1 (Undecidability of $\text{HALT}_f^{\text{par}}$ [87, 88])**.** *The decision problem* $\text{HALT}_f^{\text{par}}$ *is undecidable if and only if* $\forall n \in \mathbb{N}.f(n) \geq n + 1$.

The (necessary and sufficient) condition $\forall n \in \mathbb{N}.f(n) \geq n + 1$ stems from the fact that multi-tape Turing machines need at least $|w| + 1$ steps to read an entire input string w and then halt by entering accept or reject states. If this condition does not hold, let n_0 be a number such that $f(n_0) \leq n_0$. If a given Turing machine indeed has a bound $f(n)$, then any input string longer than n_0 cannot trigger a different behavior of the Turing machine from strings of length n_0. Hence, to solve the decision problem $\text{HALT}_f^{\text{par}}$, it suffices to inspect the Turing machine's behavior for all input strings w where $|w| \leq n_0$.

Theorem 6.3.2 (Undecidability of $\text{HALT}_F^{\text{par}}$ [87, 88])**.** *Let* $F \subseteq \{f : \mathbb{N} \to \mathbb{Q}_{\geq 0}\}$ *be an (infinite) set of functions that contains arbitrarily large constant functions. Then the decision problem* $\text{HALT}_F^{\text{par}}$ *is undecidable.*

The decision problems $\text{HALT}_f^{\text{par}}$ and $\text{HALT}_F^{\text{par}}$ considered in Thms. 6.3.1 and 6.3.2 are both about the running time of multi-tape Turing machines. Gajser [87, 88] also studies the one-tape Turing machine's counterparts of the decision problems $\text{HALT}_f^{\text{par}}$ and $\text{HALT}_F^{\text{par}}$, which are denoted by $\text{HALT}_f^{1,\text{par}}$ and $\text{HALT}_F^{1,\text{par}}$, respectively. Gajser [87, 88] then proves that $\text{HALT}_f^{1,\text{par}}$ and $\text{HALT}_F^{1,\text{par}}$ are undecidable, where the author requires that a target symbolic bound f in $\text{HALT}_f^{1,\text{par}}$ should be in $\Omega(n \log n)$, in addition to the assumptions in Thms. 6.3.1 and 6.3.2.

Finally, it follows from Thm. 6.3.1 that the decision problem $\text{HALT}^{\text{par}}_{\text{input},F}$ (Defn. 6.2.3) is undecidable under a similar condition on the set F.

Theorem 6.3.3 (Undecidability of $\text{HALT}^{\text{par}}_{\text{input},F}$)**.** *Consider a (possibly infinite) set F of symbolic cost bounds containing $f \in F$ such that $\forall n.f(n) \geq n+1$. Then the decision problem $\text{HALT}^{\text{par}}_{\text{input},F}$ is undecidable.*

Proof. Assume a set F of symbolic cost bounds containing $f \in F$ such that $\forall n.f(n) \geq n+1$. The proof goes by reducing $\text{HALT}^{\text{par}}_{f}$ to $\text{HALT}^{\text{par}}_{\text{input},F}$. Given a Turing machine M, the pair $\langle M, f\rangle$ satisfies

$$\langle M\rangle \in \text{HALT}^{\text{par}}_{f} \iff \langle M, f\rangle \in \text{HALT}^{\text{par}}_{\text{input},F}. \tag{6.3.1}$$

Since $\text{HALT}^{\text{par}}_{f}$ is undecidable (Thm. 6.3.1), so is $\text{HALT}^{\text{par}}_{\text{input},F}$. □

6.3.2 Total Computation

This section proves undecidability of resource-analysis decision problems for total computation (§6.2.2). It is a stronger result than the undecidability of resource analysis for partial computation (§6.3.1). Gajser [87, 88] focuses only on undecidability of decision problems $\text{HALT}^{\text{par}}_{f}$ and $\text{HALT}^{\text{par}}_{F}$ for partial computation. Nonetheless, Gajser [87] states in passing that its results and proofs can be adapted to total computation. This section substantiates the claim by formally stating and proving undecidability of resource analysis for total computation. All proofs in this section are adapted from the proofs in Gajser [87, 88], which are originally for partial computation.

I start with Lem. 6.3.1 stating that a Turing machine has a constant time bound if it has a bound f where $\exists n.f(n) \leq n$.

Lemma 6.3.1 (Constant time bound [87, 88])**.** *Consider a symbolic bound $f : \mathbb{N} \to \mathbb{Q}_{\geq 0}$ such that $\exists n.f(n) \leq n$. If a (one-tape or multi-tape) Turing machine M has a time bound f, then the machine M has a constant time bound.*

Proof. Let $n_0 \in \mathbb{N}$ be such that $f(n_0) \leq n_0$. For any input string w such that $|w| \geq n_0$, the machine M can only read at most n_0 first characters in the input w, without knowing whether the $(n_0+1)^{\text{th}}$ character is the blank symbol or not. Therefore, the machine M halts in n_0 steps on all inputs w such that $|w| \geq n_0$, yielding a constant time bound of n_0. □

Lem. 6.3.2 plays a key role in reducing the halting problem $\text{HALT}^{1}_{\epsilon}$ to resource-analysis decision problems for total computation.

Lemma 6.3.2. *Let $f : \mathbb{N} \to \mathbb{Q}_{\geq 0}$ be a symbolic bound such that $\forall n.f(n) \geq n+1$. Also, let g be a computable symbolic bound. Then there exists an algorithm that takes as input a one-tape Turing machine M and returns a total multi-tape Turing machine $\tilde{M}$ such that*

$$M \text{ diverges on input } \epsilon \implies \forall w \in \Sigma^*.\tilde{M} \text{ halts in } |w|+1 \text{ steps on input } w \tag{6.3.2}$$

$$\implies \forall w \in \Sigma^*.\tilde{M} \text{ halts within } f(|w|) \text{ steps on input } w \tag{6.3.3}$$

$$M \text{ halts on input } \epsilon \implies \tilde{M} \text{ runs in at least } g(|w|) \text{ steps for all sufficiently long } w. \tag{6.3.4}$$

In Eq (6.3.4), sufficiently long w means $|w| \geq N$ for some constant N (which is dependent on the machine M).

Proof. Given a one-tape Turing machine M, a multi-tape Turing machine $\tilde{M}$ works as follows. Let w be an input string for the machine $\tilde{M}$. The machine $\tilde{M}$ reads the string w on the input tape by moving its head to the right until it reaches the first blank symbol[1]. At the same time, the machine $\tilde{M}$ simulates the one-tape Turing machine M on the empty string ϵ using the working tape(s) of $\tilde{M}$. If the machine M terminates within $|w|$ steps (where the number of steps is tracked by the input-tape head of $\tilde{M}$), then the machine $\tilde{M}$ computes $g(|w|)$ and then runs $g(|w|)$ more steps. Conversely, if the machine M does not terminate within $|w|$ steps, then the machine $\tilde{M}$ halts by entering the accept state.

Suppose that the machine M diverges on the empty string ϵ. Then the machine $\tilde{M}$ runs in $|w| + 1$ steps on all inputs w because the machine M fails to terminate within $|w|$ steps on the empty string ϵ, regardless of how long the input w is. Therefore, Eqs. (6.3.2) and (6.3.3) hold.

Conversely, suppose that the machine M halts on the empty string ϵ in N steps. Then for any input w such that $|w| \geq N$, the machine $\tilde{M}$ runs in at least $n + 1 + g(|w|)$ steps. To see why, the machine $\tilde{M}$ first runs $|w|$ steps to read the input w while simulating the machine M on the empty string ϵ for $|w|$ steps. Because M terminates within $|w| \geq N$ steps, the machine $\tilde{M}$ computes $g(|w|)$ (which is computable by assumption) and then runs $g(|w|)$ steps before halting. Because the computation of $g(|w|)$ will require additional steps, the total running time of the machine $\tilde{M}$ is more than $n + 1 + g(|w|)$. Thus, Eq (6.3.4) holds. □

I now present undecidability theorems for the three resource-analysis decision problems Defns. 6.2.4–6.2.6 for total computation.

Theorem 6.3.4 (Undecidability of $\text{HALT}_f^{\text{tot}}$)**.** *Let $f : \mathbb{N} \to \mathbb{Q}_{\geq 0}$ be a symbolic bound such that $\forall n. f(n) \geq n + 1$. Also, let g be a computable symbolic bound such that $\forall n. g(n) > f(n)$. Then the decision problem $\text{HALT}_f^{\text{tot}}$ for total computation is undecidable.*

Proof. The proof goes by reducing the halting problem HALT_ϵ^1, which is known to be undecidable (Thm. 6.1.1), to $\text{HALT}_f^{\text{tot}}$. Given a one-tape Turing machine M, we construct a total multi-tape Turing machine $\tilde{M}$ according to Lem. 6.3.2. We have

$$M \text{ diverges on input } \epsilon \implies \forall w \in \Sigma^*. \tilde{M} \text{ halts within } f(|w|) \text{ steps on input } w \tag{6.3.5}$$

$$M \text{ halts on input } \epsilon \implies \tilde{M} \text{ runs in at least } g(|w|) \text{ steps for all sufficiently long } w, \tag{6.3.6}$$

where $\forall n. g(n) > f(n)$ by assumption. Because it is undecidable whether the one-tape Turing machine M halts on the empty string ϵ, so is the decision problem $\text{HALT}_f^{\text{tot}}$. □

[1] As the machine $\tilde{M}$ is a multi-tape Turing machine, it is capable of simulating the one-tape Turing machine M and tracking its number of steps at the same time. Otherwise, if the machine $\tilde{M}$ were a one-tape Turing machine, it would be unable to perform these two tasks in parallel. As a result of the difference in multi-tasking capabilities between one-tape and multi-tape Turing machines, the undecidability results of resource analysis for multi-tape and one-tape Turing machines have slightly different formulations [87, 88].

Theorem 6.3.5 (Undecidability of $\mathrm{HALT}_F^{\mathrm{tot}}$). *Let $F \subseteq \{f : \mathbb{N} \to \mathbb{Q}_{\geq 0}\}$ be an infinite set of functions that contains arbitrarily large constant functions. Additionally, assume that there exists a computable function g such that*

$$\forall f \in F. \exists n_0. \forall n \geq n_0. g(n) > f(n). \tag{6.3.7}$$

That is, a computable function g is asymptotically larger than every symbolic bound $f \in F$. Then the decision problem $\mathrm{HALT}_F^{\mathrm{tot}}$ is undecidable.

Proof. Let F_0 be $F_0 \coloneqq \{f \in F \mid \forall n. f(n) \geq n + 1\}$. I conduct case analysis on whether the set F_0 is empty or not. In both cases, I reduce the halting problem HALT_ϵ^1, which is undecidable (Thm. 6.1.1), to the decision problem $\mathrm{HALT}_F^{\mathrm{tot}}$.

Suppose $F_0 = \emptyset$. To reduce the halting problem HALT_ϵ^1 to $\mathrm{HALT}_F^{\mathrm{tot}}$, given a one-tape Turing machine M, we construct a total multi-tape Turing machine $\tilde{M}$ as follows. Given an input string w, the machine $\tilde{M}$ simulates the machine M on the empty string ϵ for at most $|w|$ steps, where the number of steps is tracked by the input-tape head of $\tilde{M}$. If the machine M halts within $|w|$ steps, then the machine $\tilde{M}$ also halts at the same time. Conversely, if the machine M does not halt within $|w|$ steps, the machine $\tilde{M}$ makes one more step and then halts. Consequently, we have

$$\begin{aligned} M \text{ halts on input } \epsilon &\iff \tilde{M} \text{ has a constant time bound} && (6.3.8)\\ &\iff \exists f \in F. \forall w \in \Sigma^*. \tilde{M} \text{ halts within } f(|w|) \text{ steps on input } w. && (6.3.9) \end{aligned}$$

In Eq (6.3.9), the direction $\Rightarrow$ follows from the assumption that F contains arbitrarily large constant functions, and the other direction $\Leftarrow$ follows from Lem. 6.3.1 and the assumption $F_0 = \emptyset$ (i.e., $\forall f \in F. \exists n. f(n) \leq n$). Since the halting problem HALT_ϵ^1 is undecidable, so is the decision problem $\mathrm{HALT}_F^{\mathrm{tot}}$.

Next, suppose $F_0 \neq \emptyset$. Again, I reduce the halting problem HALT_ϵ^1 to $\mathrm{HALT}_F^{\mathrm{tot}}$. Given a one-tape Turing machine M, we construct a total multi-tape Turing $\tilde{M}$ according to Lem. 6.3.2, where a symbolic bound f is chosen arbitrarily from the set F_0. It gives us

$$M \text{ diverges on input } \epsilon \implies \exists f \in F_0 \subseteq F. \forall w \in \Sigma^*. \tilde{M} \text{ halts within } f(|w|) \text{ steps on input } w \tag{6.3.10}$$

$$\begin{aligned} M \text{ halts on input } \epsilon &\implies \tilde{M} \text{ runs in at least } g(|w|) \text{ steps for all sufficiently long } w && (6.3.11)\\ &\implies \nexists f \in F. \forall w \in \Sigma^*. \tilde{M} \text{ halts within } f(|w|) \text{ steps on input } w. && (6.3.12) \end{aligned}$$

In Eq (6.3.12), the implication follows from the assumption that the computable bound g is asymptotically larger than any $f \in F$. Since HALT_ϵ^1 is undecidable, so is the decision problem $\mathrm{HALT}_F^{\mathrm{tot}}$. Thus, whether F_0 is empty or not, the decision problem $\mathrm{HALT}_F^{\mathrm{tot}}$ is undecidable. □

To apply Thm. 6.3.5 to $\mathrm{HALT}_{\mathrm{P}}^{\mathrm{tot}}$ (i.e., polynomial bounds), we can set a computable function g to be $g(n) \coloneqq e^n$, which is asymptotically larger than all polynomial functions.

It is important that such a computable and asymptotically large function g exists. Otherwise, the decision problem $\mathrm{HALT}_F^{\mathrm{tot}}$ can be decidable. For instance, if F is the set of all computable

functions $f : \mathbb{N} \to \mathbb{Q}_{\geq 0}$, then $\mathrm{HALT}^{\mathrm{tot}}_{F}$ is trivially decidable (i.e., the answer is always yes). This is because, if a multi-tape Turing machine M terminates on all inputs, a symbolic cost bound of the machine M is simply M itself (with a slight modification to return a numeric output instead of yes or no).

Theorem 6.3.6 (Undecidability of $\mathrm{HALT}^{\mathrm{tot}}_{\mathrm{input},F}$)**.** *Consider a (possibly infinite) set F of symbolic cost bounds containing $f \in F$ such that $\forall n.f(n) \geq n+1$. Also, let g be a computable symbolic bound such that $\forall n.g(n) > f(n)$. Then the decision problem $\mathrm{HALT}^{\mathrm{tot}}_{\mathrm{input},F}$ is undecidable.*

Proof. The proof goes by reducing the decision problem $\mathrm{HALT}^{\mathrm{tot}}_{f}$, which is undecidable (Thm. 6.3.4), to the decision problem $\mathrm{HALT}^{\mathrm{tot}}_{\mathrm{input},F}$. □

Beyond the time metric In formulating and proving undecidability of resource analysis, I have so far focused on the time metric of multi-tape Turing machines. Many existing works on undecidability focus on the time metric [87, 88, 112]. Fundamentally, because resource analysis under the time metric is similar to halting problems, the time metric makes resource analysis amenable to reduction from halting problems, thereby proving undecidability. By contrast, the space metric of Turing machines is less similar to halting problems than the time metric: having finite space usage does not imply termination of Turing machines.

Nonetheless, some of the undecidability proofs can be adapted to space usage of Turing machines. For example, Lem. 6.3.2, which is the crux of the undecidability of $\mathrm{HALT}^{\mathrm{tot}}_{f}$ (Thm. 6.3.4), can be modified by replacing the time metric with the space metric of multi-tape Turing machines. However, it is unclear whether, for example, Thm. 6.3.5 still holds for the space metric.

Finally, since the time metric is a special case of the more general tick metric, resource analysis for the tick metric is also undecidable under some conditions on target symbolic bounds. Furthermore, if the user is allowed to specify an arbitrary resource metric using the construct tick, then we may no longer need requirements such as $\forall n.f(n) \geq n+1$ on a target symbolic f in $\mathrm{HALT}^{\mathrm{par}}_{f}$ and $\mathrm{HALT}^{\mathrm{tot}}_{f}$. This condition on the symbolic bound f is imposed to prevent Turing machines from becoming trivial to analyze (e.g., failing to read entire inputs as in Lem. 6.3.1).

6.3.3 Soundness and Completeness of Resource Analysis

This section discusses soundness and completeness of static, data-driven, and interactive resource analyses.

Defn. 6.3.1 defines *soundness* and *completeness* of algorithms for resource-analysis decision problems.

Definition 6.3.1 (Soundness and completeness of resource analysis)**.** *Consider a resource-analysis algorithm A that aims to solve a resource-analysis decision problem L. The algorithm A is said to be sound if*

$$\forall M.A \text{ returns yes} \implies \langle M \rangle \in L. \tag{6.3.13}$$

Dually, the algorithm A is said to be complete if

$$\forall M.\langle M \rangle \in L \implies A \text{ returns yes}. \tag{6.3.14}$$

Thm. 6.3.7 relates soundness, completeness, and decidability of resource analysis.

Theorem 6.3.7. *There exists a total, sound, and complete algorithm A that solves a resource-analysis decision problem L if and only if the problem L is decidable.*

Proof. It follows from the definitions of soundness, completeness, and decidability. If a resource-analysis decision problem L is decidable, it means there exists a total (i.e., terminating) Turing machine/algorithm A such that, for any input Turing machine $\langle M \rangle$, we have

$$\langle M \rangle \in L \implies A \text{ returns yes on } \langle M \rangle \tag{6.3.15}$$
$$\langle M \rangle \notin L \implies A \text{ returns no on } \langle M \rangle. \tag{6.3.16}$$

Therefore, the (total) algorithm A is sound and complete (Defn. 6.3.1). The other direction of the bi-implication can be proved similarly. □

Static and data-driven resource analyses As a consequence of Thm. 6.3.7, if a resource-analysis decision problem L is undecidable, then a total resource-analysis algorithm A cannot be both sound and complete. In the face of this impossibility result, static and data-driven resource analyses make different trade-offs between soundness and completeness.

Static resource-analysis techniques (e.g., AARA [112, 113, 116] and COSTA [9, 11]) are usually sound but incomplete for the six resource-analysis decision problems we have discussed: Defns. 6.2.1–6.2.3 for partial computation and Defns. 6.2.4–6.2.6 for total computation. These static techniques reason about worst-case behaviors of programs by examining their source code. Hence, whenever these techniques successfully verify cost bounds, they come with proofs of soundness.

On the other hand, data-driven resource-analysis techniques (e.g., trend profiler [94], algorithmic profiling [234], and input-sensitive profiling [60]) are usually complete but unsound. For instance, to solve $\text{HALT}_f^{\text{tot}}$ with a target symbolic bound f, data-driven analysis runs a given program P on finitely many program inputs x and checks whether the program P halts within $f(|x|)$ steps. However, data-driven analysis cannot verify that the program P has a cost bound f by repeatedly testing program inputs x. The analysis can only refute the cost bound.

Remark 6.3.8 (Completeness of Bayesian data-driven analysis)**.** *Data-driven analysis is complete for the decision problem* $\text{HALT}_f^{\text{tot}}$ *as long as the analysis simply checks the target cost bound f with respect to the (finitely many) observed cost measurements of an input Turing machine. However, some data-driven techniques add an additional buffer on top of observed costs. For example, BayesWC (§7.3.3) first runs Bayesian inference to infer worst-case costs for those input sizes present in observed runtime-cost data $\mathcal{D}$. These inferred worst-case costs can be strictly higher than the maximum observed costs in the observed data $\mathcal{D}$. BayesWC then solves a constrained optimization problem to infer a symbolic bound that lies above all the inferred worst-case costs. Thus, if a ground-truth cost bound f is too close to maximum observed costs, BayesWC may reject it as a candidate cost bound, rendering BayesWC incomplete as a decision procedure for* $\text{HALT}_f^{\text{tot}}$.

In fact, for BayesWC, as long as its probabilistic model is carefully designed (i.e., it satisfies the robustness condition (7.3.9)), a ground-truth cost bound f has a positive probability density in the posterior distribution. However, if a fixed buffer is added on top of maximum observed costs and then the resulting costs are tested with a target cost bound f, such data-driven analysis is not complete. «

Remark 6.3.9 (Mathematical soundness of data-driven analysis)**.** *The word "sound" can be overloaded in data-driven analysis. Although data-driven analysis is not sound as a decision procedure, it can still be a mathematically sound technique. For example, in Bayesian data-driven resource analysis (§7.3), Bayesian inference is conducted to infer cost bounds using (i) a (user-customizable) probabilistic model π; and (ii) observed cost measurements $\mathcal{D}$ of an input program. As Bayesian inference relies on Bayes' rule, its inference results are mathematically sound with respect to the underlying statistical model π and observed data $\mathcal{D}$.* «

Interactive resource analysis Defn. 6.3.1 of soundness and completeness is inapplicable to interactive resource analysis, where the user collaborates with an interactive theorem prover to prove a cost bound in a program logic (e.g., **calf** [180] and Iris with time credits [168]). Defn. 6.3.1 is only applicable to *algorithms*, while interactive resource analysis is a *proof system*.

Proof systems have their own definitions of soundness and completeness. Soundness means that any statement provable in the proof system (e.g., Peano arithmetic) is valid in relevant semantic models (e.g., standard and non-standard models of natural numbers). Dually, completeness means any valid statement in relevant models has a proof in the proof system.

Thm. 6.3.10 states that computably axiomatized proof systems for resource analysis must be incomplete. Here, a proof system is said to be *computably axiomatized* if a Turing machine can enumerate all theorems provable in the proof system [25]. The theorem follows from undecidability of resource-analysis decision problems.

Theorem 6.3.10 (Incompleteness of resource-analysis proof system)**.** *Given a resource-analysis decision problem L, if it is undecidable, then any computably axiomatized proof system for the decision problem L is incomplete.*

Proof. The proof goes by contradiction. Suppose a resource-analysis decision problem L is undecidable. Let T be a computably axiomatized proof system for the decision problem L. Assume that the proof system T is complete for the decision problem L: the proof system T has a correct proof for either $\langle M \rangle \in L$ or $\langle M \rangle \notin L$ for any Turing machine M. Then for any Turing machine M, we can solve the decision problem L by simultaneously searching for proofs of $\langle M \rangle \in L$ and $\langle M \rangle \notin L$. The search terminates because (i) one of the two statements $\langle M \rangle \in L$ and $\langle M \rangle \notin L$ holds; and (ii) any valid statement is provable in the proof system T thanks to its completeness. This contradicts the assumption that the decision problem L is undecidable. □

The proof of Thm. 6.3.10 is analogous to the proof of Gödel's incompleteness theorem via undecidability of a halting problem [25, §12].

Completeness of program logics, particularly Hoare logic for imperative programs, has been extensively studied in the literature [16, 17]. Notable results include relative completeness of Hoare logic by Cook [59] and the impossibility of sound and relatively complete Hoare-like proof systems in the presence of some programming constructs by Clarke [58]. However, as far as I know, no existing works on completeness of program logics are specific to resource analysis.

6.4 Polynomial-Time Completeness of AARA

Despite the incompleteness of AARA, the typable fragment of AARA is *polynomial-time complete* [186]. It means, for any function $f : \mathbb{N} \to \mathbb{N}$, if it is polynomial-time (i.e., there exists a polynomial-time Turing machine that computes the function f), there is a program that (i) computes the same function f while preserving the input-output behavior and resource usage; and (ii) is typable in AARA.

§6.4.1 states the theorem of polynomial-time completeness and illustrates its proof idea. §6.4.2 describes in detail how to construct a list of polynomial length such that AARA can infer its polynomial bound. Lastly, §6.4.3 describes in detail how to translate a polynomial-time Turing machine to a functional program.

6.4.1 Theorem Statement

Thm. 6.4.1 formally states the polynomial-time completeness of the typable fragment of AARA.

Theorem 6.4.1 (Polynomial-time completeness of AARA)**.** *Let M be a polynomial-time one-tape Turing machine:*

$$M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}}). \tag{6.4.1}$$

Let $p(n) : \mathbb{N} \to \mathbb{N}$ be a degree-d polynomial time bound of the machine M that can be expressed as a non-negative linear combination of binomial coefficients $\binom{n}{i}$ ($i = 0, \ldots, d$). Then there exists a functional program $P_M : L(\Sigma) \to L(\Sigma)$ such that

- *For every input $x \in \Sigma^*$, we have $M(x) = P_M(x)$;*
- *The program P_M has the same cost as the running time of the Turing machine M, where the resource metric of the program P_M is the number of function calls;*
- *AARA can infer a resource-annotated type of the program P_M, which encodes a polynomial larger than $p(n)$.*

That is, the set of programs typable in AARA (under the resource metric of the number of function calls) is complete *with respect to polynomial-time functions.*

To prove Thm. 6.4.1, given a polynomial-time one-tape Turing machine M with a polynomial time bound $p(n)$, where n is the length of an input string, I construct a program P_M that simulates the machine M. The resource metric of the program P_M is the number of functions calls (including all recursive calls and helper functions). This resource metric serves as a good approximation of the running time of the program P_M.

Alg. 1 describes the operational workings of the program P_M. The program P_M first creates a list $\ell_{\text{potential}}$ whose length is $p(n)$ (line 4). Without knowing the polynomial time bound $p(n)$ in advance, the program P_M cannot be constructed. The list $\ell_{\text{potential}}$ acts as a reservoir of potential, storing one unit of potential per element. The list $\ell_{\text{potential}}$ is constructed by a recursive function, where each recursive call runs tick 1 to incur the cost of 1.

The program P_M then simulates the Turing machine M (line 6), deleting the head element of the list $\ell_{\text{potential}}$ every time the Turing machine M moves its tape head (line 7). The program P_M runs the expression tick 1 (line 8), whose cost is paid by the potential stored in the list $\ell_{\text{potential}}$.

The program P_M constructs the list $\ell_{\text{potential}}$ of length $p(n)$ in such a way that AARA's type system can infer a polynomial cost bound for constructing the list. The polynomial bound

Algorithm 1 Program P_M simulating the Turing machine M

Require: Input $x : L(\Sigma)$

1: **procedure** $P_M(w)$
2: Create the list $\ell_{\text{left}} := [\,]$ and a list $\ell_{\text{right}} := [\bot, \ldots, \bot]$ of length $p(|x|)$
3: Prepend the input x to the list ℓ_{right}
4: Create a list $\ell_{\text{potential}}$ of length $p(|x|)$ ▷ Reservoir of potential
5: $s \leftarrow q_0$ ▷ Initialize the current state
6: **while** $s \neq q_{\text{accept}} \wedge s \neq q_{\text{reject}} \wedge \ell_{\text{potential}} \neq [\,]$ **do**
7: $\ell_{\text{potential}} \leftarrow \text{tail}\ \ell_{\text{potential}}$ ▷ Potential is released
8: Run tick 1 to consume one unit of potential
9: Compute $\delta(s, \ell_{\text{right}}[0])$
10: Update s and $\ell_{\text{right}}[0]$ appropriately
11: Update the tape head's position by moving the head of ℓ_{left} or ℓ_{right} to the other
12: **return** append (reverse $\ell_{\text{left}}, \ell_{\text{right}}$)

inferred by AARA is (strictly) larger than the polynomial time bound $p(n)$ of the Turing machine M. This is because, to construct the list $\ell_{\text{potential}}$, we must pay for both (i) the cost of function calls while constructing the list; and (ii) the potential stored in the output list $\ell_{\text{potential}}$ after construction. Since the list $\ell_{\text{potential}}$'s construction is typable in AARA, so is the program P_M.

The idea of adding a list to a program that explicitly encodes a known cost bound is later exploited in the development of the second hybrid resource analysis, resource decomposition (§8). In resource decomposition, an extra numeric variable is added to an original program to encode a quantity that cannot be expressed or inferred by one of the constituent analysis techniques (e.g., logarithmic recursion depth of MergeSort, which cannot be expressed in AARA).

6.4.2 Generating Lists of Polynomial Length

This section describes how to generate a list of polynomial length in a way that enables AARA to infer the polynomial length. Defn. 6.4.1 introduces a function

$$\texttt{poly}_{d,s} : L(\tau) \times L(\tau) \to L(\tau), \tag{6.4.2}$$

where $d \in \mathbb{N}$ is a polynomial degree, s is a symbol to fill the list with, and τ is the type of the symbol s. It takes two input lists $x : L(\tau)$ and $a : L(\tau)$, creates a new list $y : L(\tau)$ of polynomial length $\binom{|x|}{d}$, and then prepends the list y to the accumulator a.

Definition 6.4.1 (Function $\texttt{poly}_{d,s}$)**.** *For a polynomial degree $d \in \mathbb{N}$ and a symbol $s : \tau$, the function $\texttt{poly}_{d,s} : L(\tau) \times L(\tau) \to L(\tau)$ is defined as follows:*

$$\text{fun}\ \texttt{poly}_{0,s}\ (x, a) = \text{let}\ \_ = \text{tick}\ 1\ \text{in}\ s :: a \tag{6.4.3}$$

$$
\begin{aligned}
\text{fun}\ \mathtt{poly}_{d+1,s}\,(x,a) = \text{let}\ \_ = \text{tick}\ 1\ \text{in}\\
\text{case}\ x\ \{[\,] \hookrightarrow a \mid (x_1 :: x_2) \hookrightarrow\\
\text{share}\ x_2\ \text{as}\ x_{2,1}, x_{2,2}\ \text{in}\\
\text{let}\ a_2 = \mathtt{poly}_{d,s}\,(x_{2,1}, a)\ \text{in}\\
\mathtt{poly}_{d+1,s}\,(x_{2,2}, a_2)\}.
\end{aligned}
\tag{6.4.4}
$$

The underscore in Eqs. (6.4.3) and (6.4.4) means it does not matter what symbol goes in there.

The function $\mathtt{poly}_{d,s}$ runs tick 1 at the start of the function body (Eqs. (6.4.3) and (6.4.4)) to increment a cost counter by one for each function call.

The intuition behind Eq (6.4.4) is that we create two lists: (i) a list of length $\binom{n-1}{d}$ by computing $\mathtt{poly}_d\,(x_{2,1}, a)$; and (ii) a list of length $\binom{n-1}{d+1}$ by computing $\mathtt{poly}_{d+1,s}\,(x_{2,2}, x_3)$. We then combine the two lists, resulting in a list of length $\binom{n}{d+1}$ due to the identity $\binom{n}{d+1} = \binom{n-1}{d} + \binom{n-1}{d+1}$.

Lem. 6.4.1 establishes the correctness of the implementation of the function $\mathtt{poly}_{d,s}$ (Defn. 6.4.1).

Lemma 6.4.1 (Correctness of $\mathtt{poly}_{d,s}$)**.** *The computation of* $\mathtt{poly}_{d,s}\,(x,a)$ *produces a list of length* $\binom{|x|}{d} + |a|$*, where* $\binom{n}{k} := 0$ *whenever* $n < k$*.*

Proof. The proof goes by nested induction: outer induction on d and inner induction on $|x|$. For the base case where $d = 0$, the output length is indeed $\binom{n}{0} + |a| = 1 + |a|$.

For the inductive case, suppose that the claim holds when $d = k$ for some $k \geq 0$. The proof proceeds by (inner) induction on $|x|$. When $|x| = 0$, we have

$$|\mathtt{poly}_{k+1,s}\,(x,a)| = 0 + |a|$$

according to the first branch of pattern matching in Eq (6.4.4). Hence, the claim holds when x is empty.

Conversely, for the inductive case of the inner induction, given $x = x_1 :: x_2$, we have

$$
\begin{aligned}
|\mathtt{poly}_{k+1,s}\,(x,a)| &= |\mathtt{poly}_{k+1,s}\,(x_2, \mathtt{poly}_{k,s}\,(x_2,a))| && \text{by Eq (6.4.4)} && (6.4.5)\\
&= \binom{|x_2|}{k+1} + \binom{|x_2|}{k} + |a| && \text{by the inductive hypothesis} && (6.4.6)\\
&= \binom{|x|-1}{k+1} + \binom{|x|-1}{k} + |a| && \text{because } x = x_1 :: x_2 && (6.4.7)\\
&= \binom{|x|}{k+1} + |a|. && && (6.4.8)
\end{aligned}
$$

Therefore, the claim holds for the inductive case of the outer induction. □

Lemma 6.4.2 (Typability of $\mathtt{poly}_{d,s}$)**.** *The function* $\mathtt{poly}_{d,s}$ *($d \in \mathbb{N}$) can be typed as follows in AARA:*

$$\mathtt{poly}_{0,s} : \langle L^0(\tau) \times L^1(\tau), 2\rangle \to \langle L^1(\tau), 0\rangle \tag{6.4.9}$$

$$\mathtt{poly}_{d,s} : \langle L^{\vec{q}_d}(\tau) \times L^1(\tau), 1\rangle \to \langle L^1(\tau), 0\rangle \qquad \text{for } d \geq 1, \tag{6.4.10}$$

where the tuple $\vec{q}_d := (\underbrace{2,\ldots,2}_{d-1\ \text{times}}, 3) \in \mathbb{Q}^d_{\geq 0}$ *($d \geq 1$) is a resource annotation for the polynomial* $n \mapsto 3\binom{n}{d} + \sum_{i=1}^{d-1} 2\binom{n}{i}$.

Proof. The proof goes by induction on the polynomial degree d. For the base case where $d = 0$, the function $\texttt{poly}_0$ (Eq (6.4.3)) runs tick 1 and then prepends the element $s : \tau$ to the list $a : L(\tau)$. Thus, the function $\texttt{poly}_{0,s}$ must be supplied with two units of constant potential: (i) one unit of potential to pay for tick 1; and (ii) one unit of potential to store in the element s, which is prepended to the accumulator $a : L^1(\tau)$. Hence, by the typing rules T:Tick:Pos and T:List:Cons of AARA (Listing 4.2), the function can be typed as Eq (6.4.9).

For the next base case where $d = 1$, variables appearing in the function definition $\texttt{poly}_1$ (Eq (6.4.4)) are assigned resource-annotated types as follows:

$$x : L^3(\tau) \quad a : L^1(\tau) \quad x_2 : L^3(\tau) \quad x_{2,1} : L^0(\tau) \quad x_{2,2} : L^3(\tau). \tag{6.4.11}$$

Applying the shift operator $\triangleleft$ (Eq (4.2.8)) to the resource annotation in the typing judgment $x_1 : L^3(\tau)$ yields the judgment $x_2 : L^3(\tau)$. The resource annotation in the judgment $x_2 : L^3(\tau)$ is then shared between $x_{2,1} : L^0(\tau)$ and $x_{2,2} : L^1(\tau)$ by the typing rule T:Share (Listing 4.2). The two recursive calls to $\texttt{poly}_{0,s}$ and $\texttt{poly}_{1,s}$ in Eq (6.4.4) are typed as

$$x_{2,1} : L^0(\tau), a : L^1(\tau); 2 \vdash \texttt{poly}_0\ (x_{2,1}, a) : \langle L^1(\tau), 0\rangle \tag{6.4.12}$$

$$x_{2,2} : L^3(\tau), a_2 : L^1(\tau); 1 \vdash \texttt{poly}_1\ (x_{2,2}, a_2) : \langle L^1(\tau), 0\rangle. \tag{6.4.13}$$

Eqs. (6.4.12) and (6.4.13) contain the input constant potential 1 and 2, respectively. The total constant potential 3 is obtained from the pattern matching on the list $x : L^3(\tau)$, which is split into the head $x_1 : \tau$, the tail $x_2 : L^3(\tau)$, and the constant potential 3 by the typing rule T:Case:List (Listing 4.2). Thus, the function definition of $\texttt{poly}_{0,s}$ is well-typed in AARA.

For the inductive case, suppose the claim holds for $d = k$ for some $k \geq 1$. To prove the claim for the case $d = k + 1$, variables appearing in the function definition $\texttt{poly}_{k+1,s}$ (Eq (6.4.4)) are assigned resource-annotated types as follows:

$$x : L^{\vec{q}_{k+1}}(\tau) \quad a : L^1(\tau) \quad x_2 : L^{\triangleleft(\vec{q}_{k+1})}(\tau) \quad x_{2,1} : L^{\vec{q}_k}(\tau) \quad x_{2,2} : L^{\vec{q}_{k+1}}(\tau). \tag{6.4.14}$$

The resource-annotated judgment $x_2 : L^{\triangleleft(\vec{q}_{k+1})}(\tau)$, which is obtained by the pattern matching on the list $x : L^{\vec{q}_{k+1}}(\tau)$ (T:Case:List), is shared between $x_{2,1} : L^{\vec{q}_k}(\tau)$ and $x_{2,2} : L^{\vec{q}_{k+1}}(\tau)$ by the typing rule T:Share (Listing 4.2). The sharing of the resource annotation works because

$$\triangleleft\vec{q}_{k+1} = \triangleleft(\underbrace{2, \ldots, 2}_{k \text{ times}}, 3) = (\underbrace{4, \ldots, 4}_{k-1 \text{ times}}, 5, 3) = \vec{q}_k + \vec{q}_{k+1}, \tag{6.4.15}$$

where $\vec{q}_i \coloneqq (\underbrace{2, \ldots, 2}_{i-1 \text{ times}}, 3)$. The two recursive calls to $\texttt{poly}_k$ and $\texttt{poly}_{k+1}$ in Eq (6.4.4) are typed as

$$x_{2,1} : L^{\vec{q}_k}(\tau), a : L^1(\tau); 1 \vdash \texttt{poly}_k\ (x_{2,1}, a) : \langle L^1(\tau), 0\rangle \tag{6.4.16}$$

$$x_{2,2} : L^{\vec{q}_{k+1}}(\tau), a_2 : L^1(\tau); 1 \vdash \texttt{poly}_{k+1}\ (x_{2,2}, a_2) : \langle L^1(\tau), 0\rangle, \tag{6.4.17}$$

where Eq (6.4.16) follows from the inductive hypothesis. The input constant potential 2 (i.e., the constant potential 1 in each of Eqs. (6.4.16) and (6.4.17)) is obtained from the pattern matching on the list $x : L^{\vec{q}_{k+1}}(\tau)$. Thus, the function definition of $\texttt{poly}_{k+1,s}$ is well-typed in AARA. □

6.4.3 Translation of Polynomial-Time Turing Machines

This section details how to translate a polynomial-time one-tape Turing machine to a program typable in AARA while preserving the input-output behavior and the polynomial time bound.

Theorem 6.4.1 (Polynomial-time completeness of AARA)**.** *Let M be a polynomial-time one-tape Turing machine:*

$$M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}}). \tag{6.4.1}$$

Let $p(n) : \mathbb{N} \to \mathbb{N}$ be a degree-d polynomial time bound of the machine M that can be expressed as a non-negative linear combination of binomial coefficients $\binom{n}{i}$ $(i = 0, \ldots, d)$. Then there exists a functional program $P_M : L(\Sigma) \to L(\Sigma)$ such that

- *For every input $x \in \Sigma^*$, we have $M(x) = P_M(x)$;*
- *The program P_M has the same cost as the running time of the Turing machine M, where the resource metric of the program P_M is the number of function calls;*
- *AARA can infer a resource-annotated type of the program P_M, which encodes a polynomial larger than $p(n)$.*

That is, the set of programs typable in AARA (under the resource metric of the number of function calls) is complete *with respect to polynomial-time functions.*

Proof. Given a one-tape polynomial-time Turing machine M with a polynomial time bound $p(n)$ of degree $d \in \mathbb{N}$, it is translated to a program P_M that simulates the Turing machine M while preserving the input-output behavior and the polynomial cost bound. Without loss of generality, I assume that $p(n) = \binom{n}{d}$.

The program P_M is defined as

$$\begin{aligned}
\mathsf{fun}\, P_M\, x = {} & \mathsf{let}\ _ = \mathsf{tick}\ 1\ \mathsf{in} \\
& \mathsf{share}\ x\ \mathsf{as}\ x_1, x_2\ \mathsf{in}\ \mathsf{share}\ x_2\ \mathsf{as}\ x_{2,1}, x_{2,2}\ \mathsf{in} \\
& \mathsf{let}\ \ell_{\text{left}} = [\,]\ \mathsf{in}\ \mathsf{let}\ \ell_{\text{blank}} = \mathtt{poly}_{d,\perp}\ (x_1, [\,])\ \mathsf{in} \\
& \mathsf{let}\ \ell_{\text{right}} = \mathsf{append}\ (x_{2,1}, \ell_{\text{blank}})\ \mathsf{in} \\
& \mathsf{let}\ \ell_{\text{potential}} = \mathtt{poly}_{d,\langle\rangle}\ (x_{2,2}, [\,])\ \mathsf{in} \\
& \mathtt{simulate}\ (q_0, \ell_{\text{left}}, \ell_{\text{right}}, \ell_{\text{potential}}).
\end{aligned} \tag{6.4.18}$$

The program P_M first creates three lists: (i) $\ell_{\text{left}} : L(\Gamma)$ for the Turing machine's tape to the left of the tape head; (ii) $\ell_{\text{right}} : L(\Gamma)$ for the Turing machine's tape to the right of the tape head; and (iii) $\ell_{\text{potential}} : L(\mathtt{unit})$ storing one unit of potential per element. The functions $\mathtt{poly}_{d,\perp}$ and $\mathtt{poly}_{d,\perp}$ construct lists of length $\binom{n}{d}$ filled with the blank symbol $\perp$ and the unit element $\langle\rangle$, respectively. They are described in Defn. 6.4.1.

The function application $\mathsf{append}\ (x_{2,1}, \ell_{\text{blank}})$ prepends the list $x_{2,1}$ to the list ℓ_{blank}. The computational cost (i.e., the number of function calls) of this function application is $1 + |x_{2,1}|$.

The function `simulate` simulates the Turing machine M with the initial state $q_0 \in Q$. Every time the Turing machine M moves its tape head, the function `simulate` removes a list element from the list $\ell_{\text{potential}}$, freeing up one unit of potential. The function `simulate` then runs `tick 1` to increment a cost counter, and this cost is paid by the potential stored in the list $\ell_{\text{potential}}$.

It follows from Lem. 6.4.2 that the function $\mathtt{poly}_{d,\langle\rangle}$ can be typed in AARA. The potential stored inside the output of $\mathtt{poly}_{d,\langle\rangle}$ is then used to pay for the cost of running the function

`simulate`. All other functions (i.e., $\texttt{poly}_{d,\perp}$, `append`, and `simulate`) called in the program P_M are typable in AARA's type system as well. Therefore, AARA can infer the polynomial bound $p(n)$ for the program P_M. □

In Thm. 6.4.1, I assume that a polynomial bound $p(n)$ can be expressed as a non-negative linear combination of binomial coefficients. If we have a polynomial of the form $a_0 + a_1 n + \cdots a_d n^d$, where $a_i \geq 0$ $(i = 0, \ldots, d)$, then the polynomial can be expressed as a non-negative linear combinations of binomial coefficients. Lem. 6.4.3 formally shows that, given a polynomial degree d, the polynomial n^d can be expressed as a non-negative linear combination of binomial coefficients $\binom{n}{i}$ $(i = 0, \ldots, d)$.

Lemma 6.4.3. *For any polynomial degree $d \in \mathbb{N}$, the polynomial function n^d can be expressed as $\sum_{i=0}^{d} q_i \binom{n}{i}$, where $q_i \in \mathbb{Q}_{\geq 0}$ for all $i = 0, \ldots, d$.*

Proof. The proof goes by induction on the polynomial degree $d \in \mathbb{N}$. For the base case $d = 0$, the claim holds because $n^0 = \binom{n}{0}$. For the inductive case, an inductive hypothesis for the case $d = k$ states that $n^k = \sum_{i=0}^{k} p_i \cdot \binom{n}{i}$, where $p_i \in \mathbb{Q}_{\geq 0}$ $(i = 0, \ldots, k)$. We obtain

$$n^{k+1} = n \cdot \sum_{i=0}^{k} p_i \binom{n}{i} = \sum_{i=0}^{k} \frac{p_i}{i!} n \cdot (n-1) \cdots (n-i+1) \cdot n \tag{6.4.19}$$

$$= \sum_{i=0}^{k} \frac{p_i}{i!} n(n-1) \cdots (n-i+1) \cdot (n - (i+1) + 1 + i) \tag{6.4.20}$$

$$= \sum_{i=0}^{k} \frac{p_i}{i!} n(n-1) \cdots (n-i+1) \cdot (n - (i+1) + 1) + \sum_{i=0}^{k} \frac{i \cdot p_i}{i!} n(n-1) \cdots (n-i+1) \tag{6.4.21}$$

$$= \sum_{i=0}^{k} (i+1) p_i \binom{n}{i+1} + \sum_{i=0}^{k} i \cdot p_i \binom{n}{i}, \tag{6.4.22}$$

where the two summations in the last line can be unified. Also, all coefficients in the last line are non-negative since p_i are all non-negative $(i = 0, \ldots, k)$. Therefore, the claim also holds for the inductive case $d = k + 1$. □

Chapter 7

Hybrid AARA

This chapter presents the first hybrid resource analysis—Hybrid AARA—that integrates different (and complementary) resource-analysis methods compositionally via a user-adjustable interface. A user annotates a program to indicate which code fragments are to be analyzed by which analysis methods. The analysis methods are performed on their respective code fragments, and their inference results are combined into an overall cost bound for the whole program. The goal of hybrid resource analysis is to retain the strengths of the constituent analysis methods while mitigating their respective weaknesses.

In addition to Hybrid AARA, this chapter presents Bayesian resource analysis, which performs Bayesian inference to infer cost bounds. Compared to optimization-based techniques prevalent in the literature of data-driven resource analysis, Bayesian techniques offer greater customizability, greater robustness, and richer information about statistical uncertainty.

Hybrid AARA specifically integrates (i) Conventional AARA (§4), a type-based static analysis method; and (ii) optimization-based and Bayesian data-driven resource analyses. For the interface between constituent analysis methods, Hybrid AARA adopts the resource-annotated type from Conventional AARA. Just like any other type, the resource-annotated type is naturally compositional, lending itself to hybrid resource analysis.

First, §7.1 gives the basics of Bayesian inference and illustrates it with linear regression. §7.2 sets the stage for data-driven and hybrid resource analyses by describing code annotations and data collection. §7.3 then formulates an optimization-based data-driven analysis technique (Opt) and two Bayesian data-driven analysis techniques (BayesWC and BayesPC). §7.4 describes Hybrid AARA, which integrates Conventional AARA and the three data-driven analyses. §7.5 describes a prototype implementation of Hybrid AARA, which is then evaluated in §7.6. Finally, §7.7 discuses the design and limitations of Hybrid AARA.

7.1 Bayesian Inference

This section introduces the basics of Bayesian inference, which is a statistical-inference method centering on Bayes' rule. §7.1.1 describes a high-level idea of Bayesian inference, and §7.1.2 illustrates Bayesian inference through an example of linear regression. Bayesian inference is applied to data-driven resource analysis in §7.3.

7.1.1 Overview

Workflow Bayesian inference consists of three steps:

1. Define a probabilistic model $\pi(\theta, y)$ over a latent variable θ and an observed variable y;
2. Collect observed data $\mathcal{D}$;
3. Compute or approximate the posterior distribution $\pi(\theta \mid y = \mathcal{D})$.

In the first step, the user defines a probabilistic model $\pi(\theta, y)$, which is a joint probability distribution of two random variables θ and y. The random variable θ, called a *latent variable*[1] [216], is a quantity the user wants to infer. In the context of data-driven resource analysis, the latent variable θ represents a symbolic cost bound (e.g., coefficients of polynomial cost bounds) of a program. The random variable y, called an *observed variable* [89], represents a quantity that the user can measure and observe. In data-driven resource analysis, the observed variable y represents a dataset of cost measurements of a program P obtained by running the program P on many inputs.

The probabilistic model $\pi(\theta, y)$ captures the user's understanding of how values of the random variables θ and y are probabilistically generated. The model is often expressed in the form

$$\pi(\theta, y) = \pi(\theta)\pi(y \mid \theta), \tag{7.1.1}$$

where the probability distribution $\pi(\theta)$ is called a *prior distribution* and the conditional distribution $\pi(y \mid \theta)$ is called a *likelihood*. The prior distribution $\pi(\theta)$ encodes the user's prior belief about the latent variable θ's distribution without observing the random variable y. The likelihood $\pi(y \mid \theta)$ encodes the user's understanding of how the observed variable y is sampled, given a fixed latent variable θ.

In the second step, the user collects observed data $\mathcal{D}$, which is a sample of the observed variable y.

In the third step, the user computes or approximates the *posterior distribution* $\pi(\theta \mid y = \mathcal{D})$, which is given by Bayes' rule:

$$\pi(\theta \mid y = \mathcal{D}) = \frac{\pi(\theta, y = \mathcal{D})}{\pi(y = \mathcal{D})} = \frac{\pi(\theta, y = \mathcal{D})}{\int_\theta \pi(\theta, y = \mathcal{D})\,\mathrm{d}\theta}. \tag{7.1.2}$$

The posterior distribution $\pi(\theta \mid y = \mathcal{D})$ represents the user's updated belief about the latent variable θ, given the observed data $\mathcal{D}$.

Once we obtain a posterior distribution of the model parameter θ, we compute a *posterior predictive distribution* [89] to predict a future observation $\tilde{y}$:

$$\pi(\tilde{y} \mid y = \mathcal{D}) = \int_\theta \pi(\tilde{y}, \theta \mid y = \mathcal{D})\,\mathrm{d}\theta \tag{7.1.3}$$

$$= \int_\theta \pi(\tilde{y} \mid \theta, y = \mathcal{D})\pi(\theta \mid y = \mathcal{D})\,\mathrm{d}\theta \tag{7.1.4}$$

$$= \int_\theta \pi(\tilde{y} \mid \theta)\pi(\theta \mid y = \mathcal{D})\,\mathrm{d}\theta. \tag{7.1.5}$$

[1] The latent variable θ is also called an unobserved variable [216] and a parameter [44, 89].

Probabilistic inference In practice, it is usually infeasible to directly compute the posterior distribution by Bayes' rule (7.1.2). This is because the denominator $\int_\theta \pi(\theta, y = \mathcal{D})\,\mathrm{d}\theta$ in Eq (7.1.2) involves an integral over the space of the latent variable θ, which is typically a high-dimensional space.

Instead, sampling-based probabilistic inference is performed to approximate the posterior distribution $\pi(\theta \mid y = \mathcal{D})$. We run a sampling algorithm to draw a finite set of M posterior samples

$$\theta_1, \ldots, \theta_M \sim \pi(\theta \mid y = \mathcal{D}). \tag{7.1.6}$$

We then use these finitely many posterior samples as an approximation of the true posterior distribution.

A commonly used family of sampling algorithms is Markov chain Monte Carlo (MCMC). Given a probabilistic model $\pi(\theta, y)$ and observed data $\mathcal{D}$, MCMC constructs a Markov chain over the space of θ such that the Markov chain's stationary distribution is the posterior distribution $\pi(\theta \mid y = \mathcal{D})$. MCMC then runs the Markov chain for sufficiently many steps (such that it converges to a stationary distribution) and then draws samples $\theta_1, \ldots, \theta_M$ of the latent variable θ from the chain.

7.1.2 Example: Bayesian Linear Regression

This section illustrates Bayesian inference through an example of linear regression. Given a set of two-dimensional coordinates $(x_i, y_i) \in \mathbb{R}^2$ $(i = 1, \ldots, N)$, the goal of (Bayesian) linear regression is to infer a linear function relating x_i and y_i $(i = 1, \ldots, N)$. For data-driven resource analysis, we can view x_i as sizes of program inputs and y_i as computational costs.

Mathematical formulation Fix a vector $\mathbf{x} \coloneqq (x_1, \ldots, x_N) \in \mathbb{R}^N$ of x-coordinates. Let a probabilistic model $\pi_{\mathbf{x}}(\theta_0, \theta_1, \sigma_{\text{noise}}, \mathbf{y})$, where $\theta_0, \theta_1, \sigma_{\text{noise}} \in \mathbb{R}$ are latent variables and $\mathbf{y} \coloneqq (y_1, \ldots, y_N)$ is a vector of observed variables, be defined as

$$\theta_0, \theta_1 \sim \text{Normal}(0, \sigma_{\text{coeff}}) \qquad \sigma_{\text{noise}} \sim \text{InverseGamma}(\alpha, \beta) \tag{7.1.7}$$

$$\hat{y}_i \coloneqq \theta_0 + \theta_1 x_i \qquad y_i \sim \text{Normal}(\hat{y}_i, \sigma_{\text{noise}}) \qquad (i = 1, \ldots N), \tag{7.1.8}$$

where the hyperparameters are

$$\sigma_{\text{coeff}} \coloneqq 5 \qquad \alpha = \beta \coloneqq 1. \tag{7.1.9}$$

In Eq (7.1.7), the first sampling statement states that the coefficients θ_0 and θ_1 of a linear function follow a normal distribution with mean 0 and standard deviation $\sigma_{\text{coeff}} \coloneqq 5$. The second sampling statement in Eq (7.1.7) states that the random variable σ_{noise} follows an inverse gamma distribution with the shape parameter $\alpha \coloneqq 1$ and the scale parameter $\beta \coloneqq 1$. In Eq (7.1.8), random variables $\hat{y}_i$ $(i = 1, \ldots, N)$ are defined as the predictions of a linear function $x \mapsto \theta_0 + \theta_1 x$. Finally, the observed variables y_i follow normal distributions with mean $\hat{y}_i$ and standard deviation σ_{noise} $(i = 1, \ldots, N)$.

```
1  data {
2      int <lower=0> N;
3      vector<lower=0>[N] x;
4      vector[N] y;
5
6      real<lower=0> coefficient_sigma;
7      real<lower=0> alpha;
8      real<lower=0> beta;
9  }
10 parameters {
11     vector[2] coefficients;
12     real<lower=0> noise_sigma;
13 }
14 model {
15     vector[N] predictions;
16
17     coefficients ~ normal(0, coefficient_sigma);
18     noise_sigma ~ inv_gamma(alpha, beta);
19     predictions = coefficients[1] + coefficients[2] * x;
20     y ~ normal(predictions, noise_sigma);
21 }
```

Lst. 7.1: Stan code of the probabilistic model (7.1.7)–(7.1.8) for Bayesian linear regression.

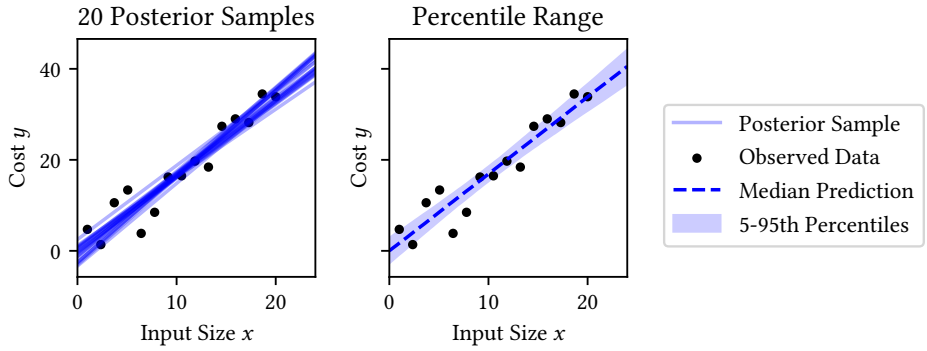


Figure 7.1: Posterior distribution of Bayesian linear regression modeled in Listing 7.1.

Probabilistic program To formally specify a probabilistic model and compute (or approximate) its posterior distribution, we use a probabilistic programming language (PPL). A PPL is a domain-specific language equipped with constructs for specifying probabilistic models and an inference engine for computing posterior distributions. Examples of PPLs include Church [96], Anglican [229], Stan [44], Pyro [33], Gen [63], PyMC [4], and Turing.jl [78].

Listing 7.1 displays the probabilistic model $\pi_{\mathbf{x}}(\theta_0, \theta_1, \sigma_{\text{noise}}, \mathbf{y})$ written in a PPL Stan [44]. The `data` block declares observed variables and the probabilistic model's hyperparameters. A programmer defines these data in a host language (e.g., Python), and they are passed to an inference engine of Stan. The program variables `N`, `x`, and `y` (lines 2 to 4) represent N, $\mathbf{x}$, and $\mathbf{y}$, respectively. Lines 6 to 8 declare the model hyperparameters (Eq (7.1.9)). In the `parameters` block, the program variable `coefficients` (line 11) represents the vector (θ_0, θ_1) of latent variables, and the program variable `sigma_noise` (line 12) represents the latent variable σ_{noise}. Finally, in the `model` block (line 14), the joint probability distribution $\pi_{\mathbf{x}}(\theta_0, \theta_1, \sigma_{\text{noise}}, \mathbf{y})$ is specified as formulated in Eqs. (7.1.7)–(7.1.8).

Fig. 7.1 displays a posterior distribution of the probabilistic model $\pi_{\mathbf{x}}$ given the black dots as observed data. I run Stan's inference engine, which implements a gradient-based sampling algorithm Hamiltonian Monte Carlo (HMC) [40, 111], to draw $M = 1000$ posterior samples. In the left plot, the blue lines represent 20/1000 randomly selected posterior samples of the coefficients (θ_0, θ_1) of linear functions. In the right plot, the blue dashed line indicates the median, and the blue shade indicates the 5–95$^{\text{th}}$ percentile range of the posterior distribution.

Linear constraints on random variables In Eq (7.1.8), the observed variables y_i are allowed to be smaller than the predictions $\hat{y}_i$ ($i = 1, \ldots, N$). Hence, if linear regression is viewed as data-driven analysis of linear cost bounds, the probabilistic model $\pi_{\mathbf{x}}$ infers an *average* symbolic cost bound. However, in this thesis, resource analysis's goal is to infer a *worst-case* symbolic cost bound, rather than an average one.

To adapt the probabilistic model $\pi_{\mathbf{x}}$ to infer a worst-case bound, Eq (7.1.8) should use a *truncated* normal distribution $\mathrm{Normal}_{[0,\hat{y}_i]}(\hat{y}_i, \sigma_{\mathrm{noise}})$, where the density is zero if y_i falls outside the interval $[0, \hat{y}_i]$. Such truncated distributions can be expressed in many PPLs, including Stan. However, if the region of non-zero densities is small, a sampling-based probabilistic inference algorithm struggles to draw posterior samples that have non-zero densities, failing to converge to the true posterior distribution quickly enough.

A more efficient sampling strategy from a truncated distribution is to restrict the search space of a sampling algorithm to the region of non-zero densities such that the algorithm never exits the region in the first place. In the case of Bayesian linear regression, the constraints

$$0 \leq y_i \leq \hat{y}_i = \theta_0 + \theta_1 x_i \qquad (i = 1, \ldots, N), \tag{7.1.10}$$

where x_i and y_i ($i = 1, \ldots, N$) are constants, are linear constraints over latent variables (θ_0, θ_1). However, many PPLs, including Stan, do not support arbitrary linear constraints on latent variables. Stan only supports box constraints (i.e., $c_1 \leq \theta \leq c_2$ for constants $c_1, c_2 \in \mathbb{R}$ and a latent variable θ), which are special cases of linear constraints.

7.2 Code Annotations and Data Collection

Hybrid AARA integrates static and data-driven resource analyses. This section discusses how to (i) specify which code fragments should be analyzed by which analysis method; and (ii) collect cost measurements to be used in the data-driven part.

Code annotations To specify which code fragments are to be analyzed by data-driven analysis, the user annotates them. Let $\mathcal{L}$ be a countable set of labels. To indicate that an expression e inside the source code is subject to data-driven analysis, the user annotates the expression as

$$\mathrm{stat}_\ell\ e, \tag{7.2.1}$$

where $\ell \in \mathcal{L}$ is a label that uniquely identifies a site of data-driven analysis. The annotation[2]stands for "statistical." The construct stat_ℓ is the only extension of the programming language's syntax (§3.1) necessary for Hybrid AARA. An expression $\mathrm{stat}_\ell\ e$ executes the expression e and records its computational cost in a dataset.

Any code fragment that lies outside an annotation stat_ℓ is analyzed by static resource analysis AARA. Given a target program $P(x) = e$ for resource analysis, if the entire function body e is annotated as $\mathrm{stat}_\ell\ e$, then hybrid resource analysis by Hybrid AARA reduces to fully data-driven resource analysis.

[2]I later realized that the annotation stat_ℓ could also stand for "static," although I intended it to stand for "statistical." However, it was too late to change the syntax in the prototype implementation of Hybrid AARA.

Monotone resource metrics Throughout this chapter, for simplicity, I focus on monotone resource metrics (e.g., running time) where every tick q in the source code satisfies $q \geq 0$. Under monotone resource metrics, the cost-semantics judgment (3.2.5) has the form $V \vdash e \Downarrow v \mid (c, 0)$, where $c \geq 0$ is the high-water-mark and net cost. This judgment is abbreviated as

$$V \vdash e \Downarrow v \mid c. \tag{7.2.2}$$

Data collection Consider a program $P(x)$ where code fragments subject to data-driven analysis are annotated with stat_ℓ for $\ell \in \mathcal{L}$. Let $L \subset \mathcal{L}$ denote the finite set of labels ℓ that appear inside annotations stat_ℓ throughout the program $P(x)$.

To construct a dataset $\mathcal{D}$ of cost measurements, we prepare $N \geq 1$ many program inputs u_i $(i = 1, \ldots, N)$ and run the target program $P(x)$ on each input u_i $(i = 1, \ldots, N)$. During its execution, we record the outputs and cost measurements of all annotated code fragments $\mathrm{stat}_\ell\ e_\ell$ $(\ell \in L)$ in the program $P(x)$. The construction of the runtime-cost dataset $\mathcal{D}$ is given by a judgment

$$(\{x : u_i\} \vdash P(x) \Downarrow \tilde{u}_i \mid c_i)_{i=1}^{N} \mid \mathcal{D}. \tag{7.2.3}$$

The data-collection judgment (7.2.3) means that, if we run the target program $P(x)$ on inputs $u_1, \ldots, u_N$, we obtain (i) an output $\tilde{u}_i$ and a (high-water-mark and net) cost $c_i \in \mathbb{Q}_{\geq 0}$ from each run of $P(u_i)$ $(i = 1, \ldots, N)$; and (ii) a dataset $\mathcal{D}$ of cost measurements from code fragments $\mathrm{stat}_\ell\ e_\ell$ $(\ell \in L)$. Unless data-driven analysis is performed on the whole target program $P(x)$, its outputs $\tilde{u}_i$ and costs c_i $(i = 1, \ldots, N)$ do not play a role in data-driven analysis.

During the execution of $P(u_i)$, suppose we encounter an annotated code fragment $\mathrm{stat}_\ell\ e_\ell$ for $N_i^\ell \geq 0$ times. Let the cost-semantics judgments (§3.2) of the expression e_ℓ while evaluating $P(u_i)$ be

$$V_{i,j}^\ell \vdash e_\ell \Downarrow \tilde{v}_{i,j}^\ell \mid c_{i,j}^\ell \qquad (i = 1, \ldots, N,\ j = 1, \ldots, N_i^\ell). \tag{7.2.4}$$

A dataset $\mathcal{D}_\ell$ of cost measurements for the code fragment $\mathrm{stat}_\ell\ e_\ell$ is constructed by recording three components: inputs $V_{i,j}^\ell$, outputs $\tilde{v}_{i,j}^\ell$, and costs $c_{i,j}^\ell$. An overall dataset $\mathcal{D}$ of cost measurements is given by aggregating all $\mathcal{D}_\ell$ for $\ell \in L$. Formally, we define

$$\mathcal{D}_\ell := \{(\ell, V_{i,j}^\ell, \tilde{v}_{i,j}^\ell, c_{i,j}^\ell) \mid 1 \leq i \leq N, 1 \leq j \leq N_i^\ell\} \quad (\ell \in L) \qquad \mathcal{D} := \bigcup_{\ell \in L} \mathcal{D}_\ell. \tag{7.2.5}$$

Remark 7.2.1 (Context dependency of data collection)**.** *Cost measurements collected as described above are* context-dependent *(i.e., dependent on the context in which a code fragment* $\mathrm{stat}_\ell\ e_\ell$ *runs), enabling data-driven analysis to infer tighter cost bounds. For instance, consider an annotated code fragment* $\mathrm{stat}_\ell\ e_\ell$ $(\ell \in \mathcal{L})$ *where the expression* e_ℓ *performs insertion sort. Although the worst-case time complexity for insertion sort is* $O(n^2)$*, its complexity becomes* $O(n)$ *if input lists are (almost) sorted. Thus, if inputs to the expression* $\mathrm{stat}_\ell\ e_\ell$ *are almost sorted, then the dataset* $\mathcal{D}_\ell$ *of cost measurements capture* $O(n)$ *costs, rather than* $O(n^2)$ *costs, of insertion sort.*

Context dependency would not hold if we recorded cost measurements of an expression $\mathrm{stat}_\ell\ e_\ell$ *in an isolated context. For instance, if we ran insertion sort on randomly generated lists, as opposed to sorted lists, we would instead collect* $O(n^2)$ *costs. This would result in less tight cost bounds from data-driven resource analysis.* «

Remark 7.2.2 (Data collection of all recursive calls). *For data-driven resource analysis, it is critical to use all cost measurements in a dataset $\mathcal{D}$, instead of dropping some of them. This is important for the soundness guarantee of data-driven analysis with respect to the inputs u_i ($i = 1, \ldots, N$) to the target program $P(x)$. For example, consider an annotated code fragment $\mathrm{stat}_\ell\ e_\ell$ ($\ell \in \mathcal{L}$) where the expression e_ℓ is the body of a recursive function. If we did not record cost measurements of all recursive calls e_ℓ, then a cost bound inferred by data-driven analysis for the entire recursive function would not be guaranteed to be sound with respect to the inputs u_i ($i = 1, \ldots, N$).* «

Remark 7.2.3 (Higher-order functions). *In the judgment (7.2.4), the input u_i to the function P, all values in the environment $V^\ell_{i,j}$ for the expression e^ℓ, and the output value $\tilde{v}^\ell_{i,j}$ are all required to have non-arrow types. Otherwise, higher-order functions would pose technical challenges in program-input generation and data collection in data-driven (and hybrid) resource analysis.*

Consider a target higher-order function $P(x)$ that takes another function as input. The first challenge is that, if a dataset of cost measurements for the function P is not readily available, the user needs to generate functions $F_1, \ldots, F_N$ to be used as inputs to the function P. Because the space of functions is vast, it is non-trivial to generate a reasonably diverse set of arrow-typed inputs. QuickCheck [57], a property-based testing tool for Haskell, lets the user write generators for arrow-typed inputs.

The second challenge is that the data-collection procedure is complicated. During the execution of the program $P(x)$, suppose the higher-order function P is executed N times, each with input F_i ($i = 1, \ldots, N$). We need to record not only cost measurements of the function P but also cost measurements of each input function F_i ($i = 1, \ldots, N$).

Dually, if the higher-order function P returns a function as output (say G), then the function G may never be executed during data collection. An example is partial application of a curried function P with multiple inputs. Without having runtime-cost data of the output function G, we cannot statistically infer a cost bound of the function P. «

7.3 Bayesian Data-Driven Resource Analyses

This section presents *Bayesian data-driven resource analysis* [188], which infers a symbolic cost bound by Bayesian inference (§7.1) on a dataset of cost measurements. Bayesian data-driven analysis offers two advantages over existing data-driven resource analyses in the literature, which are mostly optimization-based [60, 94, 234]:

1. Bayesian inference lets the user express their domain knowledge (e.g., how conservative inferred cost bounds should be with respect to the observed costs) in the form of probabilistic models. By contrast, most data-driven analyses in the literature, which are optimization-based, do not let the user customize statistical models for data analysis.
2. Bayesian inference returns a distribution of inferred cost bounds, providing greater robustness and richer information about statistical uncertainty than optimization-based methods.

§7.3.1 sets the stage for data-driven resource analysis by introducing notation. §7.3.2 presents an optimization-based method Opt, variants of which are prevalent in the literature of data-driven analysis. §7.3.3 and §7.3.4 then introduce two Bayesian data-driven resource-analysis

techniques: BayesWC and BayesPC. Finally, §7.3.5 discusses generalizations of the three data-driven analyses.

7.3.1 Setting the stage

To set the stage for data-driven analysis, consider a program $P(x) = e$. Let the dataset of cost measurements of the function body e be $\mathcal{D} \coloneqq \{(V_i, \tilde{v}_i, c_i)\}_{i=1}^{N}$. To simplify the presentation, I assume that P takes as input an integer list. Since $V_i \equiv \{x : v_i\}$ holds for each $i, \ldots, N$, I denote the measurements more concisely as $(v_i, \tilde{v}_i, c_i)$. A cost bound of P is described by a resource-annotated typing judgment

$$\{x : L^{\vec{p}}(\texttt{int})\}; p_0 \vdash P\, x : \langle L^{\vec{q}}(\texttt{int}), q_0\rangle, \tag{7.3.1}$$

where $\vec{p}$ and $\vec{q}$ are vector of polynomial coefficients (except for degree-zero coefficients) of input and output potential functions, respectively, and p_0 and q_0 are constant potential in the input and output, respectively. This typing judgment is sound if, for all lists $v : L(\texttt{int})$ such that $\{x : v\} \vdash P\, x \Downarrow \tilde{v} \mid c$, the input potential is enough to pay for the cost and output potential:

$$[\Phi(v : L^{\vec{p}}(\texttt{int})) + p_0] - [\Phi(\tilde{v} : L^{\vec{q}}(\texttt{int})) + q_0] \equiv \left[\Psi(|v|; p_0, \vec{p}) - \Psi(|\tilde{v}|; q_0, \vec{q})\right] \geq c, \tag{7.3.2}$$

where I have introduced the function

$$\Psi(n; p_0, \vec{p}) \coloneqq p_0 + \sum_{i=1}^{|\vec{p}|} p_i \binom{n}{i} \qquad (n \in \mathbb{N}) \tag{7.3.3}$$

to evaluate the amount of potential for input size n with coefficients p_0 and $\vec{p}$ of polynomial potential functions.

Unlike Conventional AARA, which derives (7.3.1) by static analysis of e and linear programming, data-driven resource analysis infers the parameters $(p_0, \vec{p})$ and $(q_0, \vec{q})$ using the dataset $\mathcal{D}$ of cost measurements. An inferred symbolic cost bound for the net cost (and also the high-water-mark cost) of $P(x)$ is given by the function

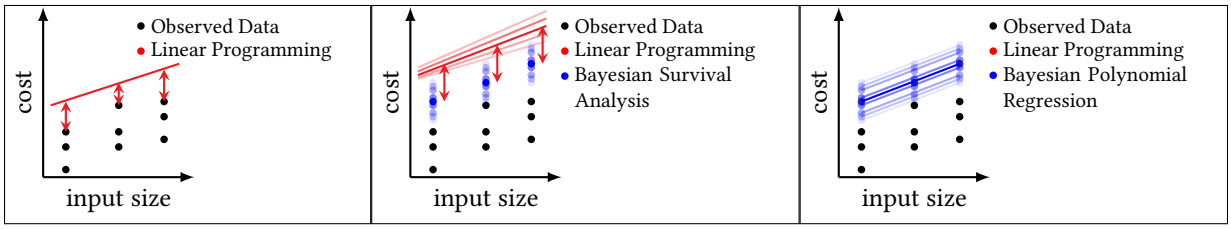
$$x \mapsto \Psi(|x|; p_0, \vec{p}). \tag{7.3.4}$$

To obtain a tighter bound for the net cost, we could take the function $x \mapsto \Psi(|x|; p_0, \vec{p}) - \Psi(|P(x)|; q_0, \vec{q})$ as a cost bound, although it is parametric in not only the input x but also the output $P(x)$ (see Remark 4.1.1).

7.3.2 Optimization-Based Data-Driven Analysis

Before presenting Bayesian inference, I consider a simple optimization-based baseline (adapted from the literature [60, 94, 234]) to ensure that (7.3.2) is satisfied with respect to the runtime-cost data $\mathcal{D}$:

$$\forall i = 1, \ldots, N. \Psi(|v_i|; p_0, \vec{p}) \geq c_i + \Psi(|\tilde{v}_i|; q_0, \vec{q}). \tag{7.3.5}$$



(a) Opt (b) BayesWC (c) BayesPC

Figure 7.2: Three approaches to data-driven resource analysis. (a) Opt uses linear programming to fit a polynomial curve that lies above the runtime data while minimizing the distance to the observed worst-case cost at each input size. (b) BayesWC uses a two-step approach: first, Bayesian survival analysis is used to infer a posterior distribution over the worst-case cost at each input size; second, linear programming is used to fit polynomial curves with respect to samples from the inferred distribution of worst-case costs. (c) BayesPC uses Bayesian polynomial regression to infer the coefficients of polynomial curves that lie above the observed runtime data.

We seek the tightest bound among all $p_0, \vec{p}, q_0, \vec{q}$ that minimizes the nonnegative cost gaps between the predicted and observed costs in the dataset $\mathcal{D}$. Letting

$$N_{\mathcal{D}} \coloneqq \{|v_i| \mid i = 1, \ldots, N\} \qquad \text{set of unique input sizes appearing in } \mathcal{D} \tag{7.3.6}$$

$$\hat{c}_n^{\max} \coloneqq \max\{c_i \mid i = 1, \ldots, N, |v_i| = n\} \qquad \text{max. observed cost for input size } n \in N_{\mathcal{D}} \tag{7.3.7}$$

$$c_n^{\max} \coloneqq \max\{\mathrm{cost}(P(v)) \mid v : L(\texttt{int}), |v| = n\} \qquad \text{true worst-case cost for input size } n \in N_{\mathcal{D}}, \tag{7.3.8}$$

I define the following linear program:

$$\begin{aligned} \text{minimize} \quad & \textstyle\sum_{i=1}^{N} \left[\Psi(|v_i|; p_0, \vec{p}) - \Psi(|\tilde{v}_i|; q_0, \vec{q})\right] - \hat{c}_{|v_i|}^{\max} && \text{(Opt-LP)} \\ \text{subject to} \quad & \Psi(|v_i|; p_0, \vec{p}) \geq \Psi(|\tilde{v}_i|; q_0, \vec{q}) + \hat{c}_{|v_i|}^{\max} && (i = 1, \ldots, N) \\ & p_0, p_1, \ldots, p_{|\vec{p}|}, q_0, q_1, \ldots, q_{|\vec{q}|} \geq 0. \end{aligned}$$

Equivalently, in (Opt-LP), I can replace $\hat{c}_{|v_i|}^{\max}$ with c_i $(i = 1, \ldots, N)$ without changing the space of feasible and optimal solutions. This optimization-based approach to data-driven resource analysis is dubbed Opt and is illustrated in Fig. 7.2a.

Opt has a shortcoming that its inference result may have zero probability of being a sound cost bound. That is, Opt fails to satisfy *robustness*. While any solution $\hat{p}_0, \hat{\vec{p}}, \hat{r}_0, \hat{\vec{q}}$ to (Opt-LP) is guaranteed to satisfy (7.3.5), an inferred cost bound $\Psi(n; \hat{p}_0, \hat{\vec{p}})$ may lie below the true value $c_n^{\max}$ in Eq (7.3.8) (which I assume is finite). This shortcoming occurs because Opt uses the point estimate $\hat{c}_n^{\max}$ given in Eq (7.3.7) as a proxy for $c_n^{\max}$, which is not robust in cases where the data $\mathcal{D}$ is such that $\hat{c}_n^{\max} < c_n^{\max}$ for some $n \in N_{\mathcal{D}}$.

To *partially* fix this issue, we can let the user adjust the objective function and constraints in (Opt-LP). However, no optimization-based techniques in the literature do so. Furthermore,

even with this partial fix, any optimization-based technique only returns a single inferred cost bound, failing to achieve robustness and capture statistical uncertainty.

7.3.3 Bayesian Inference on Worst-Case Costs (BayesWC)

Overview The first approach to addressing the aforementioned limitation of Opt is *Bayesian inference on worst-case costs* (BayesWC). Whereas Opt uses the data $\mathcal{D}$ to form a point estimate $\hat{c}_n^{\max}$ of the worst-case cost $c_n^{\max}$ for each input size $n \in N_{\mathcal{D}}$ in the linear program, BayesWC instead leverages $\mathcal{D}$ to learn an entire probability distribution μ_n that characterizes our uncertainty about $c_n^{\max}$. I identify two requirements that the inferred worst-case cost distributions μ_n must satisfy:

$$\mu_n([\hat{c}_n^{\max}, \infty)) = 1 \qquad \forall \epsilon > 0, w > \hat{c}_n^{\max}.\, \mu_n([w-\epsilon, w+\epsilon]) > 0. \tag{7.3.9}$$

The left expression guarantees *soundness* (7.3.5) with respect to runtime-cost data $\mathcal{D}$, and the right expression ensures *robustness* with respect to the true worst-case cost $c_n^{\max}$. Robustness means that an inference result has a positive probability of being a sound cost bound even if the dataset $\mathcal{D}$ does not contain worst-case inputs. Opt satisfies soundness with respect to the dataset $\mathcal{D}$, but not robustness due to usage of point estimates.

If we have access to probability distributions μ_n ($n \in N_{\mathcal{D}}$) over worst-case costs, we can use them to robustly estimate bounds by generating $|N_{\mathcal{D}}|$ batches of $M > 0$ i.i.d. samples

$$(c'_{n,1}, \ldots, c'_{n,M}) \sim \mu_n \qquad (n \in N_{\mathcal{D}}). \tag{7.3.10}$$

Reorganizing these $|N_{\mathcal{D}}| \times M$ samples into M lists $\mathbf{c}'_j := (c'_{n,j}; n \in N_{\mathcal{D}})$ $(j = 1, \ldots, M)$ each of length $N_{\mathcal{D}}$, we obtain posterior samples of coefficients $p_0, \vec{p}, q_0, \vec{q}$ by solving M linear programs parametrized by the random samples $\mathbf{c}'_j$:

$$\begin{aligned}
\text{minimize} \quad & \textstyle\sum_{i=1}^{N} \left[\Psi(|v_i|; p_0, \vec{p}) - \Psi(|\tilde{v}_i|; q_0, \vec{q})\right] - c'_{|v_i|,j} && \text{(BayesWC-LP)}\\
\text{subject to} \quad & \Psi(|v_i|; p_0, \vec{p}) \geq \Psi(|\tilde{v}_i|; q_0, \vec{q}) + c'_{|v_i|,j} && (i = 1, \ldots, N)\\
& p_0, p_1, \ldots, p_{|\vec{p}|}, q_0, q_1, \ldots, q_{|\vec{q}|} \geq 0.
\end{aligned}$$

Fig. 7.2b illustrates BayesWC, where the blue dots above a given input size n represents the samples $c'_{n,j}$ from the worst-case cost distribution μ_n. The solutions of the corresponding linear programs (BayesWC-LP) are shown in red. Whereas Opt delivers a single bound using from one LP, BayesWC delivers posterior samples of bounds using multiple randomly generated LPs.

Sampling worst-case costs via Bayesian inference To obtain distributions μ_n over worst-case costs that satisfy soundness and robustness in Eq (7.3.9), I perform Bayesian inference using a probabilistic generative model and observed costs of the target program P. Define the following notation:

$$\mathbf{v} := (v_1, \ldots, v_N) \qquad \text{observed inputs in runtime-cost data } \mathcal{D} \tag{7.3.11}$$

$$\mathbf{c} := (c_1, \ldots, c_N) \qquad \text{observed costs in runtime-cost data } \mathcal{D} \tag{7.3.12}$$

$$\mathbf{y} := (y_1, \ldots, y_N) \qquad \text{random variables of the costs } y_i \text{ of running } P \text{ on inputs of length } |v_i| \text{ for } i = 1, \ldots, N. \tag{7.3.13}$$

We design a (yet-to-be-specified) probabilistic model indexed by $\mathbf{v}$:

$$\pi_{\mathbf{v}}(\theta, \mathbf{y}) \coloneqq h(\theta) \prod_{i=1}^{N} g(y_i; \theta, |v_i|), \tag{7.3.14}$$

where $h(\theta)$ is a prior distribution of latent parameters θ and $g(y_i; \theta, |v_i|)$ is the likelihood of a cost y_i given the latent parameters θ and the input size $|v_i|$. The probabilistic model encodes the user's domain knowledge about how conservative inferred cost bounds should be relative to maximum observed costs in the dataset $\mathcal{D}$. Conditioned on the observed costs $\mathbf{c}$ in the dataset $\mathcal{D}$, the posterior distribution of the latent parameters θ is given by Bayes' rule as

$$\pi_{\mathbf{v}}(\theta \mid \mathbf{y} = \mathbf{c}) = \frac{\pi_{\mathbf{v}}(\theta, \mathbf{y} = \mathbf{c})}{\int_{\theta} \pi_{\mathbf{v}}(\theta, \mathbf{y} = \mathbf{c}) \, \mathrm{d}\theta} \propto h(\theta) \prod_{i=1}^{N} g(y_i = c_i; \theta, |v_i|) \tag{7.3.15}$$

Suppose that, given $\mathcal{D}$, we are able to infer the posterior $\pi_{\mathbf{v}}(\theta \mid \mathbf{y} = \mathbf{c})$ as defined in Eq (7.3.15). We generate samples $(c'_{n,1}, \ldots, c'_{n,M})$ in Eq (7.3.10) from the worst-case cost distribution μ_n by sampling:

$$\theta_j \sim \pi_{\mathbf{v}}(\theta \mid \mathbf{y} = \mathbf{c}) \qquad c'_{n,j} \sim \widetilde{g}(y; \theta_j, n, [\hat{c}_n^{\max}, \infty)) \qquad (j = 1, \ldots, M; n \in N_{\mathcal{D}}), \tag{7.3.16}$$

where a truncated distribution $\widetilde{g}$ is defined as the restriction of g to an interval $U \subset \mathbb{R}$:

$$\widetilde{g}(x; \theta, n, U) \coloneqq \frac{g(x; \theta, n)\mathbf{I}[x \in U]}{\int_{x \in U} g(x; \theta, n) \, \mathrm{d}x} \qquad (x \in \mathbb{R}), \tag{7.3.17}$$

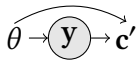
where

$$\mathbf{I}[x \in U] \coloneqq \begin{cases} 1 & \text{if } x \in U \\ 0 & \text{otherwise.} \end{cases} \tag{7.3.18}$$

Proposition 7.3.1 (Soundness and robustness of BayesWC)**.** *If the likelihood $g(y; \theta, n)$ has full support over $[0, \infty)$, then the inferred worst-case-cost distribution μ_n defined by Eq (7.3.16) satisfies the soundness and robustness properties (7.3.9).*

Remark 7.3.1. *The reader may be concerned that the distributions of the M simulated worst-case costs $\mathbf{c}' \coloneqq (c'_n; n \in N_{\mathcal{D}})$ in Eq (7.3.16) are defined in terms of $\hat{c}_n^{\max}$, which are observation-specific quantities. In Bayesian inference, unless we adopt empirical Bayes [45], prior distributions must be independent of observations. Otherwise, the prior distributions would not faithfully reflect the user's "prior" belief before collecting observed data. Therefore, it is reasonable to be wary of the dependence of the worst-case costs $\mathbf{c}'$ on observations.*

Nonetheless, the random variable c'_n has a well-defined prior distribution $h(\theta)$ that is independent of observed data $\mathcal{D}$. First of all, the probabilistic model $\pi_{\mathbf{v}}$ is indexed by a fixed vector of input instances $\mathbf{v}$ that uniquely define the sizes $N_{\mathcal{D}}$. Hence, $N_{\mathcal{D}}$ is not part of observed variables. Furthermore, the random variables θ, $\mathbf{y}$, and $\mathbf{c}'$ are related by the following factorization and graphical representation:



$$\pi_{\mathbf{v}}(\theta, \mathbf{y}, \mathbf{c}') \coloneqq h(\theta) \textstyle\prod_{i=1}^{N} g(y_i; \theta, |v_i|) \prod_{n \in N_{\mathcal{D}}} \widetilde{g}(c'_n; \theta, n, [\max_{i \in [N]; |v_i| = n} y_i, \infty)).$$

The conditional independence structure in the model is now obvious, and the M replicates of $\mathbf{c}'_j := (c'_{n,j}; n \in N_{\mathcal{D}})$ $(i = 1, \ldots, M)$ drawn in Eq (7.3.16) are valid posterior inferences conditioned on the observed costs $\mathbf{c}$ in $\mathcal{D}$. «

Survival analysis for worst-case costs To obtain the distribution $\pi_{\mathbf{v}}(\theta \mid \mathbf{y} = \mathbf{c})$ (7.3.15), we design a domain-general probabilistic model grounded in survival analysis [15] for predicting "time-to-occurrence" data beyond an observed horizon. Survival analysis is a widely used family of statistical methods applied in diverse areas such as hardware failure [1], clinical trials [2], and customer analytics [129].

The probabilistic model has three parameters $\theta = \{\beta_0, \beta_1, \sigma\}$ with i.i.d. normal prior h; a hyperparameter γ_0; a likelihood model $g(y \mid \theta)$ over observable costs y that is defined implicitly through a variable transformation; and a noise distribution g_{noise}:

$$\beta_0, \beta_1, \sigma \overset{\text{iid}}{\sim} \text{Normal}(0, \gamma_0) \qquad \epsilon_i \sim g_{\text{noise}}(0, 1) \qquad (i = 1, \ldots, N) \tag{7.3.19}$$

$$x_i := \beta_0 + \beta_1 |v_i| + |\sigma| \epsilon_i \qquad y_i = \exp(x_i) \qquad (i = 1, \ldots, N). \tag{7.3.20}$$

Possible choices for the noise distribution g_{noise} include the standard normal, logistic, or Gumbel distributions, which in turn imply that the likelihood model g is a log-normal, log-logistic, or Weibull distribution each with scale parameter $\exp(\beta_0 + \beta_1 |v_i|)$ and shape parameters $|\sigma|$, $|\sigma|^{-1}$, and $|\sigma|^{-1}$, respectively. The reference implementation by my collaborators and me (§7.5) sets g_{noise} to be a Gumbel distribution, for its relatively heavier tails as compared to other choices.

7.3.4 Bayesian Inference on Polynomial Coefficients (BayesPC)

Overview Whereas BayesWC performs Bayesian inference on worst-case costs and composes the results with (BayesWC-LP) to deliver symbolic cost bounds, we develop another approach that bypasses LP solving and directly performs Bayesian inference over the unknown coefficients $p_0, \vec{p}, q_0, \vec{q}$ in the judgment (7.3.1).

In this approach, dubbed Bayesian inference on polynomial coefficients (BayesPC), a Bayesian model is indexed by the input instances $\mathbf{v}$ and output instances $\tilde{\mathbf{v}}$ and defines a probability distribution

$$\pi_{\mathbf{v},\tilde{\mathbf{v}}}(\theta, p_0, \vec{p}, q_0, \vec{q}, \mathbf{y}) \tag{7.3.21}$$

over a set of auxiliary latent parameters θ, polynomial coefficients $p_0, \vec{p}, q_0, \vec{q}$ of potential functions in the input and output, and observable costs $\mathbf{y}$. Conditioned on observed costs $\mathbf{c}$, we sample

$$p'_0, \vec{p}', q'_0, \vec{q}' \sim \pi_{\mathbf{v},\tilde{\mathbf{v}}}(p_0, \vec{p}, q_0, \vec{q} \mid \mathbf{y} = \mathbf{c}), \tag{7.3.22}$$

which define the posterior bound $\lambda n.\Psi(n; p'_0, \vec{p}')$. Fig. 7.2c illustrates this idea: the blue curves represent posterior samples of cost bounds and the blue dots show samples c'_n of inferred worst-case costs that estimate the true value $c_n^{\max}$ at each input size $n \in N_{\mathcal{D}}$. As in BayesWC, BayesPC delivers posterior samples of both worst-case costs and cost bounds, but it rests on a different modeling and inference approach that bypasses linear programming entirely.

Bayesian polynomial regression To define a probabilistic model $\pi_{\mathbf{v},\tilde{\mathbf{v}}}(\theta, p_0, \vec{p}, q_0, \vec{q}, \mathbf{y})$ in BayesPC, I adopt Bayesian polynomial regression, which is a polynomial extension of Bayesian linear regression (§7.1.2). I model the observable costs $(c_1, \ldots, c_N)$ as random variables where each cost c_i takes values in a bounded interval $[0, c'_{|v_i|,|\tilde{v}_i|}]$, where $|v_i|$ is an input size and $|\tilde{v}_i|$ is an output size $(i = 1, \ldots, N)$. For each $n \in N_{\mathcal{D}}$, the endpoints $c'_{n,\tilde{n}}$ of these intervals are themselves random variables defined as polynomial regression outputs that capture uncertainty in the true worst-case cost $c_n^{\max}$. The probabilistic model in BayesPC is given by:

$$(p_j)_{j=0}^{|\vec{p}|}, (q_j)_{j=0}^{|\vec{q}|} \overset{\text{iid}}{\sim} \text{Normal}_{\geq 0}(0, \gamma_0) \qquad \theta \sim h_{\text{noise}}(\gamma_1) \tag{7.3.23}$$

$$c'_{n,\tilde{n}} := \Psi(n; p_0, \vec{p}) - \Psi(\tilde{n}; q_0, \vec{q}) \qquad (n \in N_{\mathcal{D}}; \tilde{n} \in \tilde{N}_{\mathcal{D}}) \tag{7.3.24}$$

$$\epsilon_i \sim \widetilde{g}_{\text{noise}}(\cdot; \theta, [0, c'_{|v_i|,|\tilde{v}_i|}]) \qquad y_i = c'_{|v_i|,|\tilde{v}_i|} - \epsilon_i \qquad (i = 1, \ldots, N; \text{i.i.d.}), \tag{7.3.25}$$

where γ_0, γ_1 are hyperparameters. In Eq (7.3.24), $\tilde{N}_{\mathcal{D}}$ is the set $\{|\tilde{v}_i| \mid i = 1, \ldots, N\}$ of output sizes in the dataset $\mathcal{D}$.

Eq. (7.3.23) samples (i) polynomial coefficients p_i, q_j $(i = 1, \ldots, |\vec{p}|, j = 1, \ldots, |\vec{q}|)$ of potential functions from a normal distribution; and (ii) a model parameter θ from a prior distribution h_{noise}. Eq. (7.3.24) then defines an inferred worst-case cost $c'_{n,\tilde{n}}$ for input size n $(n \in N_{\mathcal{D}})$ and output size $\tilde{n}$ $(\tilde{n} \in \tilde{N}_{\mathcal{D}})$. Finally, Eq (7.3.25) samples a noise ϵ_i $(i = 1, \ldots, N)$ from a truncated distribution $\widetilde{g}_{\text{noise}}(\cdot; \theta, [0, c'_{n,\tilde{n}}])$. The term $\widetilde{g}_{\text{noise}}(\cdot; \theta, [0, c'_{n,\tilde{n}}])$ is the truncation of an underlying noise distribution $g_{\text{noise}}(\cdot; \theta)$ that has full support over $[0, \infty)$ (c.f., Eq (7.3.17)).

In Eq (7.3.25), the noise ϵ_i represents the gap between inferred worst-case costs $c'_{|v_i|,|\tilde{v}_i|}$ and observed costs c_i $(i = 1, \ldots, N)$, where the latter may be smaller than the former. The noise ϵ_i is constrained to the interval $[0, c'_{|v_i|,|\tilde{v}_i|}]$ because the goal of resource analysis in this thesis to infer a *worst-case* (rather than average-case) cost bound: $0 \leq c_i \leq c'_{|v_i|,|\tilde{v}_i|}$ must hold for $i = 1, \ldots, N$.

The model parameter θ for the truncated distribution $\widetilde{g}_{\text{noise}}(\cdot; \theta, [0, c'_{n,\tilde{n}}])$ has prior h_{noise}. In an implementation of AARA (§7.5), my collaborators and I take g_{noise} to be a Weibull distribution with scale and shape parameters $\theta := (\theta_0, \theta_1)$.

Remark 7.3.2. *The main challenge to posterior inference in BayesPC is the fact that the polynomial coefficients $p_0, \vec{p}, q_0, \vec{q}$ are constrained to the linear regions (i.e., convex regions defined by linear constraints)*

$$c_i \sqsubseteq \Psi(|v_i|; p_0, \vec{p}) - \Psi(|\tilde{v}_i|; q_0, \vec{q}) \tag{7.3.26}$$

for $i = 1, \ldots, N$. That is, the coefficients must be sound at least with respect to the cost measurements in the dataset $\mathcal{D}$ such that $c_i \leq c'_{|v_i|,|\tilde{v}_i|}$ holds for all $i = 1, \ldots, N$. Coefficients outside this region have zero posterior probability density.

As discussed in §7.1.2, many probabilistic programming languages (PPLs) do not support arbitrary linear constraints on latent variables. Stan [44], for example, only supports box constraints (i.e., $c_1 \leq \theta \leq c_2$ for constants $c_1, c_2 \in \mathbb{R}$ and a latent variable θ).

Whereas traditional Markov chain Monte Carlo (MCMC) algorithms struggle in this setting, we leverage "Reflective" Hamiltonian Monte Carlo (ReHMC) sampling [47, 49, 50, 171] for posterior inference in BayesPC, where simulated trajectories reflect at the boundaries of the convex polytopes. A high-quality implementation is available in the C++ library Volesti [48]. «

7.3.5 Generalizations

I describe generalizations of data-driven resource analysis that relax the simplifying assumptions made at the beginning of this section and are needed to describe hybrid resource analysis in §7.4. I specifically discuss the following generalizations: (i) more complicated data types (e.g., tuples and nested lists) that have multiple size measures; (ii) typing contexts containing more than one variable; (iii) non-monotone resource metrics (e.g., memory); and (iv) probabilistic models besides the ones presented so far.

Size measures of values A value can have multiple measures of sizes. Suppose that a target program $P(x)$ takes as input an integer list $v : L(\texttt{int})$, as already assumed in the typing judgment (7.3.1). Let $d \in \mathbb{N}$ be a user-specified polynomial degree to be used in data-driven resource analysis. Then the resource-annotated typing judgment (7.3.1) contains polynomial coefficients $p_0, p_1, \ldots, p_d$ in the typing context, where each coefficient p_i is multiplied with a binomial coefficient $\binom{|v|}{i}$ $(i = 0, \ldots, d)$. The quantities $\binom{|v|}{i}$ for $i = 1, \ldots, d$ (and possibly the constant $\binom{|v|}{0} = 1$ as well) can each be viewed as distinct sizes of the input list $v : L(\texttt{int})$.

Furthermore, a value of a more complicated data type (e.g., tuples and nested lists) has even more sizes. For instance, in multivariate AARA [116], given a maximum polynomial degree $d = 2$, a nested list

$$v \coloneqq [v_1, v_2] \qquad v_1 \coloneqq [1, 2] \qquad v_2 \coloneqq [3, 4, 5] \tag{7.3.27}$$

has the following sizes:

1. the degree-1 outer-list length (i.e., $|v| = 2$);
2. the degree-2 outer-list length (i.e., $\binom{|v|}{2} = 1$); and
3. the combined length of all inner lists[3](i.e., $\sum_i |v_i| = 5$).

The last size, $\sum_i |v_i|$, is unique to nested lists—it does not exist for non-nested lists.

Opt and BayesPC can easily be generalized to input and output data types that have multiple sizes (e.g., nested lists). The linear program (Opt-LP) for Opt and the probabilistic model (7.3.23)–(7.3.25) for BayesPC treat all coefficients $(p_i)_{i=0}^d$ and $(q_i)_{i=0}^d$ in the same manner, where $d \in \mathbb{N}$ is a user-specified polynomial degree. Similarly, for input and output data types that have multiple sizes (e.g., nested lists), their coefficients are aggregated into tuples $(p_i)_{i=0}^d$ and $(q_i)_{i=0}^d$.

BayesWC (§7.3.3), on the other hand, requires more care than Opt and BayesPC. Survival analysis (7.3.19)–(7.3.20) infers worst-case costs $c'_{|v|}$, which are indexed by the degree-1 list length $|v|$, even if a user-specified polynomial degree is $d \geq 2$ and hence yields multiple sizes $\binom{|v|}{i}$ for $i = 1, \ldots, d$. This means only the degree-1 list length $|v|$ is used as a feature (also called an independent variable or a covariate) to infer worst-case costs $c'_{|v|}$. It might be theoretically more elegant and consistent to index inferred worst-case costs c' by a vector $\vec{n} \in \mathbb{R}^d$ of all sizes. Nonetheless, in a prototype implementation of BayesWC (§7.5), I only use the degree-1 list

[3]Multivariate AARA associates the size $\sum_i |v_i|$ with polynomial degree 2. Intuitively, this is because the combined length of all inner lists is approximated (i.e., bounded above) by a product of the outer-list length $|v|$ and the maximum inner-list length $\max_i |v_i|$. Hence, this size exists for nested lists only when a user-specified maximum polynomial degree is $d \geq 2$.

length $|v|$ to index inferred worst-case costs. This is because, for integer lists, supplying all binomial coefficients $\binom{v}{i}$ $(i = 1, \ldots, d)$ as features to survival analysis would be redundant—these sizes can all be obtained from the degree-1 size $|v|$. More generally, for arbitrary non-arrow data types, I use a vector of all degree-1 sizes to index inferred worst-case costs, but not a vector of sizes across all degrees. This means that, in benchmarks with nested lists (e.g., Concat in §7.6), worst-case costs are not indexed by the combined length of all inner lists, which has degree 2. Hence, survival analysis cannot access full information about the sizes and shapes of nested lists, since the degree-1 size of nested lists does not determine their combined length of inner lists (i.e., these two sizes are not fully correlated to each other). This design decision for survival analysis is ad hoc—generally, the user is free to choose different probabilistic models in their implementations of BayesWC.

General typing judgments For fully data-driven resource analysis, the resource-annotated typing judgment (7.3.1) assumes that the typing context of $P(x)$ is $\{x : L^{\vec{p}}(\texttt{int})\}$, which consists only of one variable. For hybrid resource analysis, we would like to generalize this resource-annotated typing judgment to

$$\{x_1 : a_1, x_2 : a_2, \ldots, x_m : a_m\}; p_0 \vdash e : \langle a, q_0 \rangle. \tag{7.3.28}$$

This is because, in hybrid resource analysis, data-driven analysis is performed on an annotated code fragment $\text{stat}_\ell\ e_\ell$ $(\ell \in \mathcal{L})$, and its typing context in general has the form $\{x_1 : a_1, x_2 : a_2, \ldots, x_m : a_m\}$ for $m \in \mathbb{N}$.

In Eq (7.3.28), each resource-annotated type a_i is associated with symbolic coefficients $\vec{p}_i$ $(i = 1, \ldots, m)$, and the output type a with coefficients $\vec{q}$. The constant potentials p_0 and q_0 are unchanged. The linear programs (Opt-LP) and (BayesWC-LP) and the BayesPC probabilistic model (7.3.23)–(7.3.25) are then defined over this expanded collection of coefficients. Simulations in Eq (7.3.22) produced from BayesPC, for example, may be written as equivalently

$$(p_0', \{\vec{p}_i'\}_{i=1}^m, \vec{q}, q_0') \sim \pi_{\mathbf{v},\tilde{\mathbf{v}}}(\cdot \mid \mathbf{y} = \mathbf{c}) \quad \text{or} \quad (p_0', \Gamma', a', q_0') \sim \pi_{\mathbf{v},\tilde{\mathbf{v}}}(\cdot \mid \mathbf{y} = \mathbf{c}). \tag{7.3.29}$$

In the latter case, the sampled typing environment Γ' is obtained by using the sampled coefficient $\vec{p}_i'$ in place of the symbolic coefficient within each resource-annotated type a_i $(i = 1, \ldots, m)$, and similarly for a' and $\vec{q}'$.

Non-monotone resource metrics So far, our discussion of data-driven resource analysis has assumed monotone resource metrics (e.g., running time), where resources are consumed but never freed up. To extend data-driven resource analysis to non-monotone resource metrics (e.g., memory), we modify the data-collection procedure by collecting not only net costs $c_{i,j}^\ell$ but also high-water-mark costs $h_{i,j}^\ell$ in Eq (7.2.4).

In Opt, the linear program (Opt-LP) imposes linear constraints on net costs c_i $(i = 1, \ldots, N)$, which are equal to high-water-mark costs under monotone resource metrics. Under non-monotone resource metrics, the linear program is augmented with additional linear constraints

$$\Psi(|v_i|; p_0, \vec{p}) \geq h_i \qquad (i = 1, \ldots, N). \tag{7.3.30}$$

As for the objective function of the linear program (Opt-LP), no choice seems to be a clear winner. The objective function of (Opt-LP), which minimizes the difference between observed net costs and predicted net-cost bounds, may remain unchanged. But the user is allowed to modify or extend the objective function by including high-water-mark costs h_i. For example, we may opt for the objective function

$$\sum_{i=1}^{N} \Psi(|v_i|; p_0, \vec{p}) - \hat{h}^{\max}_{|v_i|}, \tag{7.3.31}$$

which minimizes the difference between observed high-water-mark costs and their predicted bounds. §7.5.1 provides a holistic discussion of optimization objectives in data-driven and hybrid resource analyses.

For BayesWC to handle non-monotone resource metrics, we infer worst-case high-water-mark costs $h'_n \geq 0$ ($n \in N_{\mathcal{D}}$) by Bayesian inference, in addition to worst-case net costs c'_n. Furthermore, $h'_n \geq c'_n$ must hold for all $n \in N_{\mathcal{D}}$. The linear program (BayesWC-LP) is then modified by incorporating the inferred worst-case high-water-mark costs h'_n into linear constraints (and also the objective function if the user wishes).

Finally, for BayesPC, the Bayesian model is modified by incorporating observed high-water-mark costs $h^{\ell}_{i,j}$ into (i) the joint probability distribution and (ii) the linear constraints defining the region of positive density to draw posterior samples from.

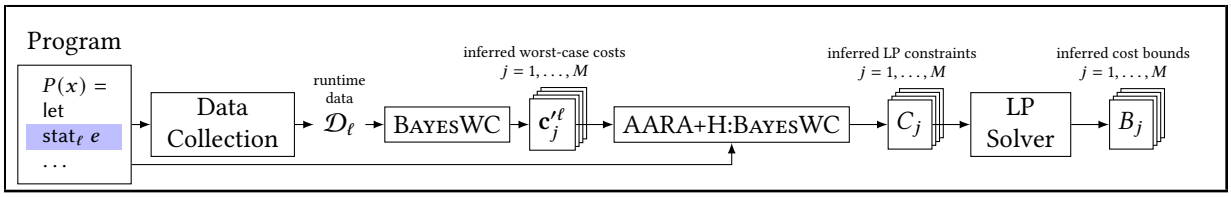
Probabilistic models Although I have presented concrete probabilistic models for BayesWC (Eqs. (7.3.19)–(7.3.20)) and BayesPC (Eqs. (7.3.23)–(7.3.25)), it is not my intention to advocate for these particular models. BayesWC and BayesPC are intended as *general* approaches to Bayesian data-driven resource analysis. Their high-level idea is that BayesWC performs Bayesian inference to infer worst-case costs c'_n ($n \in N_{\mathcal{D}}$), while BayesPC performs Bayesian inference to directly infer polynomial coefficients $p_0, \vec{p}, q_0, \vec{q}$ of input and output potential functions. Beyond this high-level idea, the concrete probabilistic models should be chosen according to the user's domain knowledge.

For instance, in BayesPC, survival analysis (Eqs. (7.3.19)–(7.3.20)) is used to obtain a distribution μ_n (Eq (7.3.10)) of inferred worst-case costs of input size $n \in N_{\mathcal{D}}$. Survival analysis is a reasonable statistical method for resource analysis because computational costs of programs, particularly their running time, can be likened to time to events (namely termination of programs), which is the subject of study in survival analysis. Nonetheless, the user is free to choose a different statistical method from survival analysis to construct distributions μ_n ($n \in N_{\mathcal{D}}$).

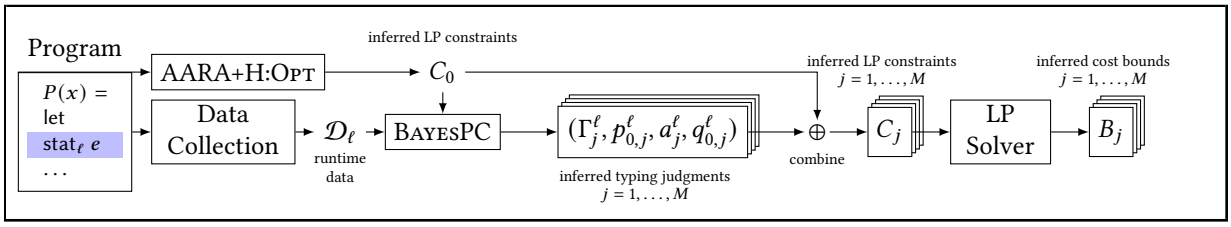
7.4 Hybrid Resource Analyses

This section presents the first hybrid resource-analysis technique, dubbed Hybrid AARA, which integrates data-driven analysis (§7.3) for code fragments annotated by stat_ℓ with static AARA analysis (§4.3) on the rest of the source code. Hybrid AARA is based on a formal typing system that extends AARA with a new typing judgment:

$$\Gamma; p_0 \vdash_{\mathcal{D}} \mathsf{stat}_\ell\ e : \langle a, q_0 \rangle. \tag{7.4.1}$$



(a) Hybrid BayesWC



(b) Hybrid BayesPC

Figure 7.3: Two hybrid resource-analysis techniques for composing static and data-driven resource analysis. Subexpression e in a function P cannot be analyzed using AARA: it is instead analyzed using Bayesian inference.

The judgment (7.4.1) extends the typing judgment (4.3.1) with a dataset $\mathcal{D}$ of runtime measurements that are collected using the procedure described in §7.2.

I also describe novel type-inference algorithms and a technical challenge in combining Bayesian and AARA-based cost-bound inference. To infer cost bounds, BayesPC runs a sampling-based probabilistic inference algorithm, while Conventional AARA runs a constrained optimization algorithm. Thus, the two constituent analyses combined by Hybrid AARA have distinct nature of cost-bound inference, making their interface challenging to design.

Interface of inference results Hybrid AARA uses the resource-annotated type as an interface between inference results of constituent analysis techniques. A cost bound inferred by the data-driven part is expressed as a resource-annotated typing judgment J_{anno} (7.3.1), where the input and output are assigned resource-annotated types. A cost bound inferred by the static part, meanwhile, is expressed as a resource-annotated typing tree T_{anno}, where annotated code fragments $\text{stat}_\ell\ e$ ($\ell \in \mathcal{L}$) are leaf nodes. To compose the two inference results in Hybrid AARA, a resource-annotated typing judgment J_{anno} from the data-driven part is inserted into the corresponding leaf node in the statically inferred typing tree T_{anno} from the latter, provided that the typing judgments do not violate linear constraints in the typing tree.

7.4.1 Hybrid BayesWC and Opt

Typing rules Given an annotated code fragment $\text{stat}_\ell\ e$ ($\ell \in \mathcal{L}$), suppose its dataset of cost measurements is

$$\mathcal{D}_\ell = \{(V_i^\ell, \tilde{v}_i^\ell, c_i^\ell) \mid i = 1, \ldots, |\mathcal{D}_\ell|\}. \tag{7.4.2}$$

Define the following notation:

$$\mathbf{V}^\ell := (V_i^\ell; i = 1, \ldots, |\mathcal{D}_\ell|) \qquad \text{tuple of all environments in } \mathcal{D}_\ell \tag{7.4.3}$$

$$\mathbf{c}^\ell := (c_i^\ell; i = 1, \ldots, |\mathcal{D}_\ell|) \qquad \text{tuple of all costs in } \mathcal{D}_\ell. \tag{7.4.4}$$

Opt and BayesWC are integrated into the AARA type system (§4.3) by adding the following rules for annotated code fragments $\mathsf{stat}_\ell\ e$ ($\ell \in \mathcal{L}$):

$$\text{H:Opt}\quad \frac{p_0 + \Phi(V_i^\ell : \Gamma) \geq q_0 + \Phi(\tilde{v}_i^\ell : a) + c_i^\ell \qquad (i = 1, \ldots, |\mathcal{D}_\ell|)}{\Gamma; p_0 \vdash_{\mathcal{D}} \mathsf{stat}_\ell\ e : \langle a, q_0 \rangle}$$

$$\text{H:BayesWC}\quad \frac{p_0 + \Phi(V_i^\ell : \Gamma) \geq q_0 + \Phi(\tilde{v}_i^\ell : a) + c'^\ell_{|V_i^\ell|} \qquad c'^\ell_n \sim \pi^\ell_{\mathbf{V}^\ell}(\cdot \mid \mathbf{y} = \mathbf{c}^\ell) \qquad (i = 1, \ldots, |\mathcal{D}_\ell|; n \in N_{\mathcal{D}_\ell})}{\Gamma; p_0 \vdash_{\mathcal{D}} \mathsf{stat}_\ell\ e : \langle a, q_0 \rangle}$$

The rule H:Opt states that the conclusion holds whenever the input potential $p_0 + \Phi(V : \Gamma)$ is large enough to cover a cost c and leftover potential $q_0 + \Phi(v : a)$ for every measurement $(V, \tilde{v}, c) \in \mathcal{D}_\ell$. Likewise, the rule H:BayesWC states that the conclusion holds whenever the input potential is large enough to cover an inferred worst-case cost $c'^\ell_{|V_i^\ell|}$ for input size $|V_i^\ell|$ and the leftover potential. The quantity $c'^\ell_{|V_i^\ell|}$ is inferred using a probabilistic model $\pi^\ell_{\mathbf{V}^\ell}$ (e.g., Eqs. (7.3.19)–(7.3.20)), which is indexed by the tuple $\mathbf{V}_\ell$ of input environments in the dataset $\mathcal{D}$. H:BayesWC is similar to H:Opt, except that each observed cost c within a measurement $(V, \tilde{v}, c)$ is replaced with a posterior sample $c'^\ell_{|V|}$ from BayesWC (7.3.10) that captures inferential uncertainty about the true worst-case cost $c^{\ell,\max}_{|V|}$.

Type inference Because the premises of H:Opt and H:BayesWC are linear constraints over the resource coefficients in e, type inference operates similarly to Conventional AARA. The only difference from Conventional AARA is that Hybrid Opt and Hybrid BayesWC treat a code fragment $\mathsf{stat}_\ell\ e$ as a leaf in a typing tree, instead of a subtree, even if the expression e is compound. The resource-annotated typing judgment for this leaf is then inferred by data-driven analysis Opt or BayesWC, instead of Conventional AARA.

Fig. 7.3a shows the type-inference workflow of Hybrid BayesWC. Given runtime-cost data $\mathcal{D}$, we first perform data-driven BayesWC inference to produce M batches of posterior samples

$$\mathbf{c}'^\ell_j := (c'^\ell_{n,j}; n \in N_{\mathcal{D}_\ell}) \qquad (j = 1, \ldots, M; \ell \in L), \tag{7.4.5}$$

which define M versions of H:BayesWC for each code fragment $\mathsf{stat}_\ell\ e$. In Eq (7.4.5), $L \subset \mathcal{L}$ is a finite set of all labels appearing in the source code of a target program $P(x)$.

Next, for each $j = 1, \ldots, M$, we perform a static pass, denoted AARA+H:BayesWC in Fig. 7.3a, that constructs a template typing tree according to the Conventional-AARA type system, except that subexpressions $\mathsf{stat}_\ell\ e$ are treated as leaves in the typing tree. That is, Conventional AARA's typing rules are only applied to traditional expressions, and the typing

rule H:BayesWC is applied to $\text{stat}_\ell\ e$ subexpressions. This process produces M systems of linear constraints C_j ($j = 1, \ldots, M$), where the linear constraints within each C_j are derived from two provenances: those from the Conventional-AARA type system and those from the typing rule H:BayesWC.

Finally, each C_j is provided to a linear-program (LP) solver to provide a resource-annotated typing judgment J_j for the root node's typing context. The judgment J_j's input polynomial potential function translates to an inferred cost bound.

Linear-programming objective When solving the overall linear program in AARA+H:BayesWC or AARA+H:Opt, an LP solver first minimizes the sum of cost gaps (i.e., difference between observed costs and predicted cost bounds) from the data-driven components (i.e., (Opt-LP) or (BayesWC-LP)). The LP solver then performs minimization of input coefficients at the root. As with Conventional AARA, it is possible to either minimize a weighted sum of coefficients at the root [112] or lexicographically minimize coefficients in the descending order of polynomial degrees [117, 118]. The prototype implementation of Hybrid AARA (§7.5) allows users to make either choice.

§7.5.1 discusses design decisions about optimization objectives in Hybrid AARA.

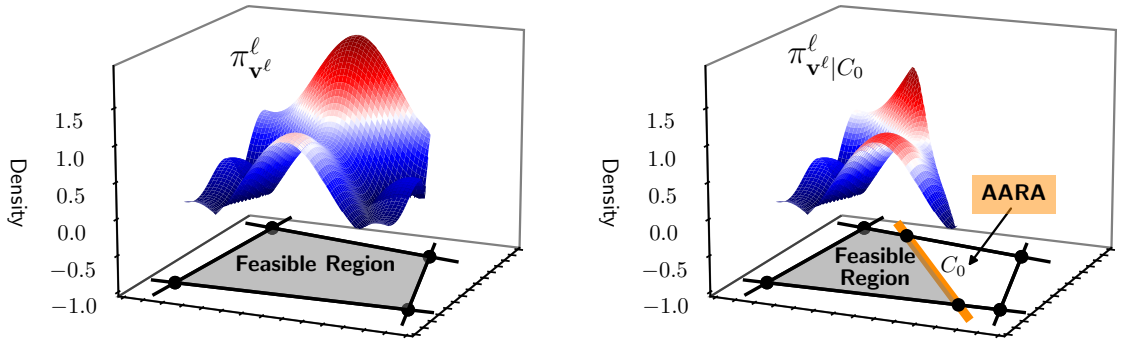
7.4.2 Hybrid BayesPC

Key challenge Integrating BayesPC into Conventional AARA is fundamentally more difficult as compared to integrating Opt and BayesWC. Resource coefficients in a resource-annotated typing judgment $\Gamma; p_0 \vdash_{\mathcal{D}} \text{stat}_\ell\ e : \langle a, q_0 \rangle$ are sampled using Bayesian inference in BayesPC, while they are optimized by LP solving in both Opt and BayesWC. As Conventional AARA also solves linear programs to infer cost bounds, Opt and BayesWC are easier to integrate with Conventional AARA.

The integration of BayesPC with Conventional AARA poses a challenge because we do not know in advance how much potential should be stored in $\langle a, q_0 \rangle$. Unlike in fully data-driven resource analysis, in hybrid resource analysis, the output of $\text{stat}_\ell\ e$ may be used in a subsequent computation that also consumes potential. The typing context Γ and constant p_0 should store enough potential to pay for both the cost of evaluating e and the cost of subsequent computation. Naïvely sampling resource annotations (Γ, p_0, a, q_0) for the $\text{stat}_\ell\ e$ subexpressions using BayesPC and providing them to a Conventional-AARA pass over the remaining source code will likely produce a linear program with no solution. Hence, we need an interface between sampling-based probabilistic inference over some coefficients and linear programming over other coefficients.

Type inference My collaborators and I address this challenge by adding linear constraints to the BayesPC probabilistic models that encode feasible regions of linear programs computed by Conventional AARA. This approach guarantees that resource-annotated judgments from BayesPC do not violate the linear constraints from Conventional AARA.

Fig. 7.3b shows the type-inference workflow of Hybrid BayesPC. We start by performing a static analysis pass through a program $P(x)$ using the Conventional-AARA type system to



(a) BayesPC (b) Hybrid BayesPC

Figure 7.4: Posterior distributions over resource coefficients restricted to convex polytopes using BayesPC.

obtain a set of linear constraints C_0, treating any $\text{stat}_\ell\ e$ using the typing rule H:Opt to ensure soundness with runtime-cost data $\mathcal{D}_\ell$. Next, for each subexpression $\text{stat}_\ell\ e$ encountered in the first pass, we apply a variant of BayesPC that combines both the runtime-cost data $\mathcal{D}_\ell$ and constraints C_0 from the first pass to infer a resource-annotated typing judgment of the form $\Gamma; p_0 \vdash_{\mathcal{D}} \text{stat}_\ell\ e : \langle a, q_0 \rangle$.

Let the BayesPC probabilistic model for $\text{stat}_\ell\ e$ (e.g., Eqs. (7.3.23)–(7.3.25)) be

$$\pi^{\ell}_{\mathbf{V}^{\ell},\tilde{\mathbf{v}}^{\ell}}(\theta, \Gamma, p_0, q_0, a, \mathbf{y}). \tag{7.4.6}$$

The linear constraints C_0 are used to construct a modified probabilistic model

$$\pi^{\ell}_{\mathbf{V}^{\ell},\tilde{\mathbf{v}}^{\ell}|C_0}(\varepsilon, \theta, \Gamma, p_0, q_0, a, \mathbf{y}), \tag{7.4.7}$$

where ε is a collection of symbolic variables that do not appear in Γ, p_0, a, q_0, but in the AARA constraints C_0. Letting $h(\varepsilon)$ denote an uninformative prior (e.g., uniform if C_0 is bounded), the modified probabilistic model is restricted to the convex polytope defined by the linear constraints C_0:

$$\pi^{\ell}_{\mathbf{V}^{\ell},\tilde{\mathbf{v}}^{\ell}|C_0}(\varepsilon, \theta, \Gamma, p_0, q_0, a, \mathbf{y}) \propto h(\varepsilon)\pi^{\ell}_{\mathbf{V}^{\ell},\tilde{\mathbf{v}}^{\ell}}(\varepsilon, \theta, \Gamma, p_0, q_0, a, \mathbf{y})\mathbf{I}[(\varepsilon, \ldots) \in C_0], \tag{7.4.8}$$

where $\mathbf{I}[(\varepsilon, \ldots) \in C_0]$ is 1 if the resource coefficients in the arguments to $\pi^{\ell}_{\mathbf{V}^{\ell},\tilde{\mathbf{v}}^{\ell}|C_0}$ satisfy constraints C_0 and 0 otherwise.

Fig. 7.4 illustrates how the probabilistic model (7.4.6) gives rise to the modified probabilistic model (7.4.7). Fig. 7.4a shows a probabilistic model (7.4.7), where the feasible region (colored gray) contains all solutions (i.e., resource coefficients) that are sound with respect to the runtime-cost data $\mathcal{D}$. Fig. 7.4b shows a modified probabilistic model (7.4.7) obtained from Fig. 7.4a by further constraining the feasible region by the linear constraints C_0 (colored orange) obtained from the Conventional-AARA type system. The probabilistic distribution in Fig. 7.4b is then re-normalized appropriately. In fact, the linear constraints C_0 already contains the linear constraints induced by the runtime-cost data $\mathcal{D}$ (i.e., gray feasible region in Fig. 7.4a).

Eq. (7.4.8) is then used to sample judgments

$$(\Gamma_j^\ell, p_{0,j}^\ell, a_j^\ell, q_{0,j}^\ell) \sim \pi^\ell_{\mathbf{V}^\ell,\tilde{\mathbf{v}}^\ell|C_0}(\Gamma, p_0, q_0, a \mid \mathbf{y} = \mathbf{c}^\ell) \qquad (j = 1, \ldots, M; \ell \in L), \tag{7.4.9}$$

which are shown as the output of BayesPC in Fig. 7.3a. Each sampled typing judgment in Eq (7.4.9) corresponds to a concrete realization of symbolic LP variables in C_0 created by the corresponding H:Opt rule at label ℓ during the first pass. We can then obtain M new constraints

$$C_j \coloneqq C_0 \oplus \{(\Gamma_j^\ell, p_{0,j}^\ell, a_j^\ell, q_{0,j}^\ell) \mid \ell \in L\} \qquad (j = 1, \ldots, M) \tag{7.4.10}$$

by syntactically replacing the symbolic LP variables in C_0 with concrete variables at all labels $\ell \in L$. Each C_j is then fed to an LP solver to obtain M posterior samples $(B_1, \ldots, B_M)$ of cost bounds.

7.4.3 Soundness

My collaborators and I have formulated and proved two soundness theorems for Hybrid AARA.

Thm. 7.4.1 establishes that inferred cost bounds from Hybrid AARA are sound with respect to all measurements in runtime-cost data $\mathcal{D}$ (collected by the procedure in §7.2).

Theorem 7.4.1 (Soundness with respect to runtime-cost data)**.** *Given an expression e (that may contain annotations stat_ℓ inside), let $\mathcal{D}$ be the runtime data such that $(V_i \vdash e \Downarrow \tilde{v}_i \mid c_i)_{i=1}^N \mid \mathcal{D}$. The following property holds with probability 1: If $\Gamma; p_0 \vdash_{\mathcal{D}} e : \langle a, q_0\rangle$ holds in the type system of Hybrid AARA, then*

$$\Phi(V_i : \Gamma) + p_0 - \Phi(\tilde{v}_i : a) - q_0 \geq c_i \tag{7.4.11}$$

holds for any $i = 1, \ldots, N$.

Proof. The proof proceeds by nested induction on $(V_i \vdash e \Downarrow v_i \mid c_i)_{i=1}^N \mid \mathcal{D}$ (outer induction) and $\Gamma; p_0 \vdash_{\mathcal{D}} e : \langle a, q_0\rangle$ (inner induction), following the same structure as the soundness proof of Conventional AARA [112, 116]. The only necessary modification is proving the base case where $e \equiv \mathrm{stat}_\ell\ e_\ell$ for some expression e_ℓ, for the new inference rules H:Opt, H:BayesWC, and H:BayesPC.

I argue that, for each typing rule, the resource-annotated judgment $\Gamma; P \vdash_{\mathcal{D}} \mathrm{stat}_\ell\ e_\ell : \langle a, q_0\rangle$ is sound with respect to any measurement $(V, \tilde{v}, c) \in \mathcal{D}_\ell$, meaning that the typing rule ensures $\Phi(V : \Gamma) + p_0 - \Phi(\tilde{v} : a) - q_0 \geq c$ holds with probability 1. For H:OPT, the property follows immediately from the premise of the typing rule.

For H:BayesWC, Prop. 7.3.1 establishes a condition that guarantees, for any runtime sample $(V, \tilde{v}, c) \in \mathcal{D}_\ell$, the probabilistic model $\pi_{\mathbf{v}}$ used in BayesWC has zero probability of simulating a cost $c'^\ell_{|V|} < c$. The choice of the likelihood function g in Eq (7.3.17) satisfies this requirement. Using this fact, the premise of H:BayesWC implies

$$\Phi(V : \Gamma) + p_0 - \Phi(\tilde{v} : a) - q_0 \geq c'^\ell_{|V|} \geq \hat{c}^{\ell,\max}_{|V|} \geq c \qquad \text{almost surely,} \tag{7.4.12}$$

and the conclusion follows.

For H:BayesPC, Remark 7.3.2 establishes that the probabilistic model $\pi^\ell_{\mathbf{V},\tilde{\mathbf{v}}}$ assigns zero posterior probability density to any (Γ, p_0, a, q_0) that satisfies $\Phi(V : \Gamma) + p_0 - \Phi(\tilde{v} : a) - q_0 < c$ for some runtime sample $(V, \tilde{v}, c) \in \mathcal{D}_\ell$. As constraining $\pi^\ell_{\mathbf{V},\tilde{\mathbf{v}}}$ to the convex polytope C_0 in Eq (7.4.8) cannot possibly increase its support, the conclusion follows for H:BayesPC. □

The second soundness theorem Thm. 7.4.2 establishes statistical soundness: the probability (over the randomness of the collected runtime-cost data) that inferred cost bounds from Hybrid AARA are sound up to a given input-size limit converges to 1 as the dataset size N tends to infinity.

To formally state the statistical-soundness theorem, it is necessary to nail down the notion of the size of program inputs (i.e., environments). Given an environment V, let its size be denoted by $|V| \in \mathbb{N}$. Any notion of the size can be used here, as long as the set of environments up to a fixed size is finite. An example of a suitable notion of the size is the number of bits necessary to encode an environment. On the other hand, if we assign, for example, the size of 1 to all numbers $v \in \mathbb{N}$, this size measure is not suitable, because the set $\{v : \texttt{int} \mid |v| = 1\}$ is infinite.

Theorem 7.4.2 (Statistical soundness)**.** *Let $\mathcal{D}$ be runtime data such that $(V_i \vdash e \Downarrow \tilde{v}_i \mid c_i)_{i=1}^N \mid \mathcal{D}$, where $(V_i)_{i=1}^N$ are N i.i.d. samples from an (unknown) input distribution and e is an expression. Fix an integer $m \in \mathbb{N}$ and a finite set $\mathcal{V}_m$ of well-typed environments such that for all $V \in \mathcal{V}_m$: $|V| \leq m$ (i.e., the size of an environment V is smaller than or equal to m) and V has nonzero probability of appearing in $\mathcal{D}$. Then for any typing judgment $\Gamma; p_0 \vdash_{\mathcal{D}} e : \langle a, q_0 \rangle$ inferred from Hybrid AARA, the probability that it satisfies*

$$\Phi(V : \Gamma) + p_0 - \Phi(\tilde{v} : a) - q_0 \geq c \tag{7.4.13}$$

for all V such that $V \vdash e \Downarrow \tilde{v} \mid c$ and $|V| \leq m$ converges to 1 as $N \to \infty$.

Proof. The proof goes by a probabilistic argument. For each $n = 1, \ldots, m$, there exists a worst-case environment

$$V_n^{\max} := \arg\max_{V \in \mathcal{V}_m} \{c \mid |V| = n, V \vdash e \Downarrow \tilde{v} \mid c\} \tag{7.4.14}$$

with a maximal execution cost. Eq. (7.4.14) is well-defined because it is assumed that $\mathcal{V}_m$ is a finite set. Let $p_n^{\max} > 0$ be the probability $V_n^{\max}$ is selected. Let $W_{n,N}$ be the event that $V_n^{\max}$ is sampled within N i.i.d. draws. The probability that all worst-case inputs $V_1^{\max}, \ldots, V_m^{\max}$ occur in runtime dataset $\mathcal{D}$ of size N is

$$\mathbb{P}\left(\bigcap_{n=0}^m W_{n,N}\right) = 1 - \mathbb{P}\left(\bigcup_{n=0}^m \overline{W}_{n,N}\right) \geq 1 - \sum_{n=1}^m \mathbb{P}(\overline{W}_{n,N}) = 1 - \sum_{n=1}^m (1 - p_n^{\max})^N \tag{7.4.15}$$

where $\overline{W}_{n,N}$ denotes the complement of the event $W_{n,N}$. The inequality follows from the union bound. The last expression converges to one as $N \to \infty$ because $p_n^{\max} > 0$ for each n. The conclusion follows from Thm. 7.4.1, which establishes that resource-annotated typing judgments from Hybrid AARA are sound with respect to the environments used to generate finite runtime-cost data $\mathcal{D}$. □

7.5 Implementation

This section describes a prototype implementation of Hybrid AARA (§7.4). It integrates (optimization-based and Bayesian) data-driven resource analyses into Resource-Aware ML (RaML) [117, 118], an implementation of multivariate polynomial AARA for analyzing OCaml programs. The prototype is publicly available in Pham et al. [187].

7.5.1 Optimization Objectives

In data-driven and hybrid resource analyses that involve optimization, no objective function is a clear winner. In the linear programs (Opt-LP) and (BayesWC-LP) for data-driven analyses, three candidate objectives exist:

- Minimize the total cost gaps (i.e., differences between observed costs and predicted cost bounds);
- Minimize coefficients $p_0, \vec{p}$ in the input potential function, either lexicographically [117, 118] or by a weighted sum [112]; and
- Maximize coefficients $q_0, \vec{q}$ in the output potential function.

The last objective cannot be used alone, because an optimal solution would be unbounded (i.e., ∞) according to this objective. Hence, it must be combined with other objectives (e.g., minimizing the input coefficients while maximizing the output coefficients).

Additionally, if Opt and BayesWC are extended to non-monotone resource metrics (§7.3.5), where high-water-mark costs and net-costs are different, the space of possible objective functions grows larger (e.g., Eq (7.3.31)).

When Opt and BayesWC are integrated with Conventional AARA, the resulting linear programs should be optimized according to the objectives of *both* data-driven analyses (i.e., Opt and BayesWC) and Conventional AARA. Otherwise, if Hybrid AARA only uses one of the two objectives, it yields an undesirable solution. Opt and BayesWC minimize total cost gaps, while Conventional AARA minimizes coefficients in the typing tree's root node [112, 117, 118]. If Hybrid AARA minimizes total cost gaps but not coefficients, Conventional AARA, which is a special case of Hybrid AARA, has no objective function to optimize. This is because Conventional AARA does not use any runtime-cost data $\mathcal{D}$ and hence has no cost gaps.

Conversely, a pitfall emerges if Hybrid AARA minimizes coefficients (particularly by lexicographic minimization) but not cost gaps. In data-driven analyses Opt and BayesWC, which are special cases of Hybrid AARA, the lexicographic minimization of coefficients yields a constant bound

$$\max\{c_i^\ell \mid i = 1, \ldots, |\mathcal{D}_\ell|\}. \tag{7.5.1}$$

This is because lexicographic minimization of coefficients in data-driven analysis sets (i) all degree-d coefficients ($d \geq 1$) to zero; and (ii) the degree-0 coefficient to the maximum cost in the dataset $\mathcal{D}_\ell$. However, constant cost bounds are not what we want all the time from data-driven resource analysis.

To avoid this issue, the prototype implementation of Hybrid AARA combines the two optimization objectives from data-driven analyses and Conventional AARA. Specifically, Hybrid AARA first minimizes total cost gaps, followed by minimization of input coefficients in the typing tree's root node. In the minimization of inputs coefficients, the user can choose between lexicographic minimization and minimization of the sum of the coefficients.

7.5.2 Probabilistic Models for BayesWC and BayesPC

This section describes an empirical-Bayes approach [45] for automatically determining the hyperparameters of the probabilistic models in BayesWC (Eqs. (7.3.19)–(7.3.20)) and BayesPC

(Eqs. (7.3.23)–(7.3.25)). Empirical Bayes is a widely used approach in Bayesian data analysis wherein observed data is used to infer plausible values of model hyperparameters: see Rizzelli et al. [194] for a recent survey. This approach applies directly to most benchmarks: only one benchmark uses a hyperparameter that deviates from the automatic procedure my collaborators and I have developed.

BayesWC Recall from Eqs. (7.3.19)–(7.3.20) that BayesWC uses a probabilistic model

$$\beta_0, \beta_1, \sigma \overset{\text{iid}}{\sim} \text{Normal}(0, \gamma_0) \qquad \epsilon_i \sim \text{Gumbel}(0, 1) \tag{7.5.2}$$

$$x_i \coloneqq \beta_0 + \beta_1 |v_i| + |\sigma| \epsilon_i \qquad y_i = \exp(x_i), \tag{7.5.3}$$

where g_{noise} in Eq (7.3.19) has been set to a standard Gumbel. The only hyperparameter in this model is γ_0, which is set to $\gamma_0 \coloneqq 5.0$ for all benchmarks. Coupled with the exp component in the likelihood model for the costs y_i, this choice determines a broad prior distribution over observed datasets.

BayesPC Recall from Eq (7.3.23) that, in the probabilistic model for BayesPC, resource coefficients are drawn from a truncated normal distribution:

$$(p_j)_{j=0}^{|\vec{p}|}, (q_j)_{j=0}^{|\vec{q}|} \overset{\text{iid}}{\sim} \text{Normal}_{\geq 0}(0, \gamma_0), \tag{7.5.4}$$

where $\gamma_0 \in \mathbb{R}_{>0}$ is the scale hyperparameter (i.e., standard deviation).

In our prototype implementation, the hyperparameter γ_0 for each benchmark is determined from data as follows. We first perform (Data-Driven or Hybrid) Opt to infer a resource-annotated typing judgment for the entire program $P(x)$:

$$\Gamma, p_0 \vdash_{\mathcal{D}} P\, x : \langle a, q_0 \rangle, \tag{7.5.5}$$

where Γ is a resource-annotated typing context, $p_0 \in \mathbb{Q}_{\geq 0}$ is constant potential for the input, a is a resource-annotated output type, and $q_0 \in \mathbb{Q}_{\geq 0}$ is constant potential for the output. Let d be a user-specified maximum polynomial degree for polynomial cost bounds, and let $p_1, \ldots, p_D$ be the resource coefficients inside Γ that correspond to indices of degree d. In other words, $p_1, \ldots, p_D$ are the polynomial coefficients of the highest-degree terms inside the polynomial potential function. The degree-0 coefficient p_0 for constant potential in the typing context is never in the set $\{p_1, \ldots, p_D\}$, unless the user specifies the maximum polynomial degree of 0. For all benchmarks in our evaluation, the hyperparameter γ_0 of the prior distribution for resource coefficients in BayesPC is set to

$$\gamma_0 \coloneqq \frac{8}{15} \max\{p_1, \ldots, p_D\} + \frac{4}{5}. \tag{7.5.6}$$

The cost gap ϵ_i in Eq (7.3.25) follows a truncated Weibull distribution:

$$c'_{|v_i|,|\tilde{v}_i|} - c_i =: \epsilon_i \sim \text{Weibull}_{[0, c'_{|v_i|,|\tilde{v}_i|}]}(\theta_0, \theta_1), \tag{7.5.7}$$

where $\theta_0 \in \mathbb{R}_{\geq 0}$ is the shape hyperparameter, $\theta_1 \in \mathbb{R}_{>0}$ is the scale hyperparameter, and $[0, c'_{|v_i|}]$ is the interval of truncation.

The two hyperparameters θ_0 and θ_1 are determined as follows. The shape θ_0 ranges from 1.0 to 1.5 across benchmarks:

- $\theta_0 \coloneqq 1.25$ in Data-Driven BayesPC of MapAppend and Hybrid BayesPC of ZAlgorithm;
- $\theta_0 \coloneqq 1.5$ in Data-Driven BayesPC of BubbleSort, Data-driven and Hybrid BayesPC of Concat, and Data-Driven BayesPC of ZAlgorithm;
- $\theta_0 \coloneqq 1.0$ for the remaining cases of BayesPC.

The scale hyperparameter θ_1 is set as follows. We first perform (Data-Driven or Hybrid) Opt to obtain a resource-annotated typing judgment around an annotated code fragment $\mathrm{stat}_\ell\ e$:

$$\Gamma, p_0 \vdash_{\mathcal{D}} \mathrm{stat}_\ell\ e : \langle a, q_0 \rangle, \tag{7.5.8}$$

where Γ is a resource-annotated typing context, $p_0 \in \mathbb{Q}_{\geq 0}$ is constant potential for the input, a is a resource-annotated output type, and $q_0 \in \mathbb{Q}_{\geq 0}$ is constant potential for the output. We then calculate the cost gap ϵ_i of $\mathrm{stat}_\ell\ e$:

$$\epsilon_i \coloneqq p_0 + \Phi(V_i^\ell : \Gamma) - q_0 - \Phi(v_i^\ell : a) \qquad (i = 1, \ldots, |\mathcal{D}_\ell|). \tag{7.5.9}$$

Let ε_α be the $\alpha \coloneqq 90^{\text{th}}$ percentile of these cost gaps $\epsilon_1, \ldots, \epsilon_{|\mathcal{D}_\ell|}$. The scale θ_1 is then

$$\theta_1 \coloneqq \frac{1100}{188.7}\varepsilon_\alpha + 100. \tag{7.5.10}$$

For Hybrid BayesPC of MedianOfMedians, the rule (7.5.10) suggests $\theta_1 \coloneqq 841.128$. However, this hyperparameter does not yield a posterior distribution whose median is close to the ground-truth cost bound of MedianOfMedians due to the inaccuracy of Opt. Therefore, Hybrid BayesPC analysis of MedianOfMedians is an exception, where my collaborators and I set $\theta_1 \coloneqq 41.128$ instead.

7.5.3 Prototype Implementation

Overview Each analysis run using the prototype requires:

- an OCaml program $P(x)$ annotated with `Raml.tick` q ($q \in \mathbb{Q}_{\geq 0}$) to indicate resource consumption and `Raml.stat` to indicate code fragments subject to data-driven resource analysis;
- a list of inputs $v_1, \ldots, v_N$ to the program $P(x)$ for runtime-cost data generation; and
- a configuration file specifying a polynomial degree, a data-driven analysis technique of choice, a probabilistic model, hyperparameters, etc.

The prototype is implemented in OCaml, built on top of the OCaml codebase of RaML [117, 118]. Additionally, the probabilistic programming language Stan [44] is used to perform Bayesian survival analysis in BayesWC. Although Stan offers an interface/binding with various host languages (e.g., Python, R, and Julia), Stan does not work with OCaml. Hence, from OCaml code, the prototype calls Python code through a Python-OCaml binding, which in turn calls Stan.

The inference engine of Stan implements the No-U-Turn Sampler (NUTS) [111]. NUTS is an example of Hamiltonian Monte Carlo (HMC) [40], which is a class of sampling-based probabilistic inference algorithms that calculate gradients of posterior distributions to guide the exploration of state spaces.

Limitations The current version of the prototype has the following limitations:

- It only supports monotone resource metrics (e.g., running time). Extension to non-monotone resource metrics is discussed in §7.3.5.
- It only supports limited probabilistic models (e.g., Weibull survival analysis for BayesWC).

Reflective Hamiltonian Monte Carlo In Data-Driven and Hybrid BayesPC, it is essential to restrict state spaces of sampling-based probabilistic inference to regions of positive posterior densities. Otherwise, without restricting the state spaces, sampling algorithms (e.g., NUTS [111] adopted in Stan [44]) would spend too much time exploring regions of zero posterior densities, repeatedly rejecting candidate posterior samples. Consequently, the sampling algorithms would fail to converge to the target posterior distribution quickly enough.

The regions of positive posterior densities in (Data-Driven and Hybrid) BayesPC are convex polytopes defined by linear constraints from two sources: the runtime-cost data $\mathcal{D}$ and the type system of Conventional AARA. In Bayesian polynomial regression (Eqs. (7.3.23)–(7.3.25)) for Data-Driven BayesPC, posterior samples have positive probability densities if and only if the posterior samples (i.e., inferred cost bounds) are sound with respect to runtime-cost data $\mathcal{D}$. This region of sound cost bounds with respect to $\mathcal{D}$ is defined by linear constraints (7.3.26) (Remark 7.3.2). In Hybrid BayesPC, resource annotations appearing in a typing tree are constrained by linear constraints from the Conventional-AARA typing rules, in addition to the linear constraints from the rule H:Opt (i.e., linear constraints induced by the dataset $\mathcal{D}$).

To restrict sampling algorithms' state spaces to convex polytopes, the Hybrid-AARA prototype leverages Reflective Hamiltonian Monte Carlo (ReHMC) [47, 49, 50, 171]. It is a sampling algorithm that runs Hamiltonian Monte Carlo (HMC) within a convex polytope defined by user-supplied linear constraints C. As the ReHMC sampler runs a Markov chain across a space of latent variables θ, whenever it hits a boundary of the linear constraints C's feasible region, the sampler "reflects" at the boundary. Thus, the sampler remains within the feasible region throughout the execution, guaranteeing that any samples drawn from ReHMC satisfy the linear constraints C.

ReHMC requires a user-specified convex polytope to be bounded. Meanwhile, linear constraints in Data-Driven and Hybrid BayesPC do not yield bounded convex polytopes. This is because these linear constraints, which come from runtime-cost data $\mathcal{D}$ or the Conventional-AARA typing rules, only specify how much leftover potential each subexpression should store. Hence, as long as the constraints have a solution, their feasible space is unbounded; that is, coefficients of potential functions can be arbitrarily large. To this end, the user of Hybrid AARA must provide a constant upper bound on resource annotations to make search spaces of ReHMC bounded.

ReHMC is implemented by the C++ library Volesti [48]. Because HMC requires gradients of posterior distributions, so does ReHMC. However, gradients are not automatically computed by Volesti. To this end, the Hybrid-AARA prototype uses another C++ library autodiff [153] to perform automatic differentiation.

Mixing time of ReHMC The mixing time (i.e., the time it takes for a Markov chain to get sufficiently close to its stationary distribution) of ReHMC has been known under the assump-

tion that target posterior distributions are log-concave [49]. ReHMC is applicable to any distributions truncated to convex polytopes: the sampler is guaranteed to converge to a target posterior distribution. However, the general mixing time of ReHMC is not known for arbitrary distributions.

Definition 7.5.1 (Log-concave distribution)**.** *A probability distribution $\pi(x)$ over the space of x is said to be log-concave if $\log(\pi(x))$ is concave; that is, $\pi(x) \propto e^{-f()x}$ for some convex function $f(x)$.*

Definition 7.5.2 (L-smooth function [41])**.** *A continuously differentiable function $f : \mathbb{R}^d \to \mathbb{R}$ is said to be L-smooth if*

$$\|\nabla f(x) - \nabla f(y)\| \leq L\|x - y\| \qquad (x, y \in \mathbb{R}^d). \tag{7.5.11}$$

Definition 7.5.3 (m-strongly convex function [41])**.** *A continuously differentiable function $f : \mathbb{R}^d \to \mathbb{R}$ is said to be m-strongly convex if*

$$f(x) - f(y) \leq \nabla f(x)^\top (x - y) - \frac{m}{2}\|x - y\|^2 \qquad (x, y \in \mathbb{R}^d). \tag{7.5.12}$$

Definition 7.5.4 (Sandwiching ratio of a convex polytope [49])**.** *Given a convex polytope $K \subseteq \mathbb{R}^d$, its sandwiching ratio at point $x^* \in \mathbb{R}^d$ is defined as*

$$\kappa \coloneqq \inf_{R>r>0} \{R/r \mid \mathbb{B}(x^*, r) \subseteq K \subseteq \mathbb{B}(x^*, R)\}, \tag{7.5.13}$$

where $\mathbb{B}(x^*, r)$ is the d-dimensional L_2 ball with radius r centered at x^*.

Theorem 7.5.1 (Mixing time of ReHMC [49])**.** *Let $\pi(x) \propto e^{-f(x)}$ be a log-concave target (posterior) distribution. Let $K \coloneqq \{\pi(x) > 0 \mid x \in \mathbb{R}^d\}$ be a convex polytope on which the distribution π has positive densities. Let $x^* \in K$ be the minimizer of $f(x)$ in K and γ be the sandwiching ratio of K at the minimizer x^*.*

Assume that the function $f(x)$ is twice differentiable, L-smooth, and m-strongly convex. A condition number is defined as $\kappa \coloneqq L/m$. Let ℓ be the maximum number of reflections made during the execution of ReHMC and ϵ be the target maximum distance (in total variation distance) that we desire between the target distribution π and a state distribution of the ReHMC sampler. The ReHMC algorithm with a step size $\eta \leq \frac{e^{-(\ell+1)^2/6}}{(2\pi e)^{1/2}(\ell+1)}$ mixes in

$$O(\kappa d^2 \ell^2 \log^2(\kappa/\epsilon) \log(d \log(\kappa/\epsilon) + d \log(\gamma/\epsilon)) \log(1/\epsilon)) \tag{7.5.14}$$

steps, given a starting point $x_0 \sim \mathrm{Normal}_K(x^*, \frac{I_d}{L})$. Here, $\mathrm{Normal}_K(x^*, \frac{I_d}{L})$ is a normal distribution truncated to the convex polytope K, and $I_d \in \mathbb{R}^{d \times d}$ is the identity matrix.

7.6 Evaluation

This section evaluates a prototype implementation of Hybrid AARA (§7.5) on challenging benchmark programs that Conventional AARA (or more generally static resource analysis) cannot solve.

7.6.1 Benchmark Suite

My collaborators and I have built a benchmark suite of 10 OCaml functional programs. Their source code is in §A.1. The resource metric of interest is the running time in all benchmark programs, although they each have slightly different notions of running time.

- MapAppend: Given two lists, x and y, for each element of x, run some function f that cannot be analyzed by static analysis and append its output to the cumulative result (whose initial value is y).
- Concat: Given a nested list, recursively append inner lists to the cumulative result.
- InsertionSort2: Sequentially run insertion sort twice on a list. The resource metric is the cost of comparisons in the second insertion sort.
- QuickSort: Run deterministic quicksort on lists with their heads as pivots for partition.
- QuickSelect: Given an integer i and a list, run deterministic quickselect and return the i^{th} smallest element in the list. Like QuickSort, it uses the heads of lists as pivots.
- MedianOfMedians: Given an integer i and a list, recursively compute the i^{th} smallest element in the list by first computing the median of medians then using it to partition the list.
- ZAlgorithm: Given a list x, return a list y such that $y[i]$ stores the maximum integer ℓ such that $x[0, \ldots, \ell - 1] = x[i, \ldots, i + \ell - 1]$.
- BubbleSort: Run bubble sort where pairs of adjacent out-of-order elements are repeatedly swapped until no such pairs exist (i.e., saturation).
- Round: Given a natural number x (represented as a list), compute a natural number y such that y is the largest power of two below x. Once y is computed, traverse y.
- EvenOddTail: Given a natural number x (represented as a list), first traverse the list and if x is even, divide it by two; otherwise, subtract one from it.

Conventional AARA cannot return a tight cost bound for any of these 10 programs. Specifically, it cannot analyze 7/10 programs at all as they contain code fragments that cannot be analyzed by static analysis.

- For BubbleSort, AARA cannot conclude its termination, let alone a quadratic cost bound. It is due to the non-size-decreasing recursion of BubbleSort: successive recursive calls to BubbleSort do not decrease the input size. BubbleSort terminates only when the input satisfies a semantic condition (i.e., the list is sorted), and this is beyond the reasoning power of AARA.
- MedianOfMedians [36] is a linear-time selection algorithm using the divide-and-conquer strategy. To determine its linear time complexity, it is necessary to reason about mathematical properties of medians, which is beyond the capability of AARA.
- For Round (taken from [112, §5.4.3]), to prove its linear complexity, AARA would need to derive an infinite set (4.4.1) of resource-annotated types.
- The four benchmarks MapAppend, Concat, QuickSort, and QuickSelect fail because they contain some complex function that Conventional AARA cannot analyze.

Table 7.1: Percentage of inferred cost bounds that are sound and analysis runtime for 10 benchmark programs.

Benchmark Program	Conventional AARA	Analysis Method	Fraction of Sound Inferred Bounds		Analysis Runtime	
			Data-Driven	Hybrid	Data-Driven	Hybrid
MapAppend	*Cannot Analyze*	Opt	0%	0%	0.01 s	0.01 s
		BayesWC	68.5%	100%	1.87 s	12.44 s
		BayesPC	75.5%	100%	51.83 s	360.80 s
Concat	*Cannot Analyze*	Opt	0%	0%	0.00 s	0.01 s
		BayesWC	67.3%	96.7%	2.54 s	14.73 s
		BayesPC	96%	100%	113.53 s	125.28 s
InsertionSort2	*Wrong Degree*	Opt	0%	0%	0.01 s	0.02 s
		BayesWC	57.6%	100%	1.53 s	5.46 s
		BayesPC	21%	57.5%	10.68 s	220.66 s
QuickSort	*Cannot Analyze*	Opt	0%	0%	0.01 s	0.11 s
		BayesWC	4%	96%	2.20 s	144.88 s
		BayesPC	0%	100%	13.72 s	274.51 s
QuickSelect	*Cannot Analyze*	Opt	0%	0%	0.02 s	0.19 s
		BayesWC	0.2%	98.2%	1.83 s	222.47 s
		BayesPC	0%	100%	12.39 s	277.20 s
MedianOfMedians	*Cannot Analyze*	Opt	0%	0%	0.17 s	0.21 s
		BayesWC	11.5%	71.3%	2.36 s	93.89 s
		BayesPC	0%	100%	70.39 s	896.98 s
ZAlgorithm	*Wrong Degree*	Opt	0%	0%	0.09 s	0.13 s
		BayesWC	13.7%	95.9%	1.96 s	72.21 s
		BayesPC	28%	100%	11.11 s	509.29 s
BubbleSort	*Cannot Analyze*	Opt	0%	*Cannot Analyze*	0.01 s	∅
		BayesWC	40.1%	*Cannot Analyze*	2.69 s	∅
		BayesPC	31.5%	*Cannot Analyze*	11.70 s	∅
Round	*Cannot Analyze*	Opt	0%	*Cannot Analyze*	0.01 s	∅
		BayesWC	58.3%	*Cannot Analyze*	1.91 s	∅
		BayesPC	81%	*Cannot Analyze*	12.87 s	∅
EvenOddTail	*Wrong Degree*	Opt	0%	*Wrong Degree*	0.01 s	∅
		BayesWC	65.1%	*Wrong Degree*	1.98 s	∅
		BayesPC	70%	*Wrong Degree*	11.79 s	∅

For the remaining 3/10 programs, to infer some polynomial cost bound, Conventional AARA requires a wrong polynomial degree that is too high. For example, InsertionSort2 has a linear cost bound, while AARA infers a quadratic bound. The second insertion sort runs in linear time because the input is already sorted by the first call to insertion sort. However, AARA infers a quadratic bound, which is a worst-case cost bound of insertion sort across all inputs.

Data-driven resource analysis also struggles to infer sound cost bounds for these benchmark programs. This is because atomic operations (e.g., integer comparison in QuickSort) are modified in such a way that their worst-case costs show up sporadically. For instance, in the source code of QuickSort (Listing A.9), the function `incur_cost` incurs a cost ranging from 0.5 to 1.0, and the worst-case cost of 1.0 only arises sporadically.

7.6.2 Experiment Results

Proportions of sound cost bounds Tab 7.1 shows (i) the proportions of inferred cost bounds that are sound and (ii) analysis runtimes for all 10 benchmarks using data-driven analysis (Opt, BayesWC, BayesPC) and their hybrid counterparts, where applicable. Optimization-based resource analyses (Data-Driven and Hybrid Opt) always return a single inferred cost bound, while Bayesian resource analyses return collections of samples of cost bounds drawn from posterior

probability distributions.

In all benchmarks, for both data-driven and hybrid analyses, BayesWC produces strictly higher proportions of sound cost bounds than Opt. This finding demonstrates that cost bounds inferred by BayesWC are more *robust* than those inferred by Opt in 10/10 cases (data-driven) and 7/7 cases (hybrid); that is, BayesWC has a positive probability of inferring a sound bound even though the runtime-cost dataset does not contain worst-case inputs. In contrast, in these benchmarks, Opt never returns a sound bound. Likewise, BayesPC returns more robust bounds than Opt in 7/10 (data-driven) and 7/7 cases (hybrid). These improvements highlight the benefits of probabilistic inference as compared to optimization.

Between data-driven Bayesian analyses (i.e., BayesWC and BayesPC) and their hybrid counterparts, the latter delivers a substantially larger fraction of sound cost bounds in all 7 benchmarks. These results illustrate the benefit of integrating data-driven analysis with static type inference on the remaining parts of the program not tagged with stat_ℓ expressions.

Analysis time The right two columns of Tab 7.1 show the analysis runtime. Opt uses an LP solver that is much faster than sampling algorithms: it returns answers in less than one second, whereas Bayesian resource analysis can take minutes. Between BayesWC and BayesPC, the former is faster in terms of analysis time per iteration. The main reason is that BayesWC uses HMC sampling without constraints, which is much simpler than Reflective HMC sampling over convex polytopes (Fig. 7.4) needed for inference in BayesPC. Furthermore, the HMC sampler used in BayesWC comes from the probabilistic programming language Stan [44], which has been developed and optimized for many years, while ReHMC in BayesPC is implemented in the C++ libraries Volesti [48] and autodiff [153], which have not been as optimized as Stan.

Distributions of estimation errors Fig. 7.5 shows relative estimation errors of inferred cost bounds with respect to the ground-truth worst-case bound in 5 benchmarks (full results in §A.1). In each benchmark, we fix three input sizes (10, 100, and 1000). For each size, we show the 5^{th}, 50^{th}, and 95^{th} percentiles of relative estimation errors. Because Opt infers a single bound, it has the same estimation error for all percentiles. Relative estimation errors below 0 (resp., above 0) indicate an underestimate (resp., overestimate) of the true bound. A cost bound is sound if its estimation error is at least 0.

Fig. 7.5 exhibits similar results to Tab 7.1:

1. Bayesian resource analyses are more robust than Opt: the former return posterior cost-bound distributions with positive probabilities of being sound bounds for each fixed input size, while the latter is either sound or unsound.
2. Hybrid analyses are more robust than data-driven analyses.

The QuickSort and QuickSelect panels in Fig. 7.5 show an interesting finding. At input size 10, the bounds from Data-Driven BayesWC and BayesPC are tighter than those from their Hybrid counterparts, but some of the former bounds are unsound. As the input size increases, the estimation errors from Hybrid AARA shrink but remain in the "Sound Region," whereas those from Bayesian data-driven analysis become unsound.

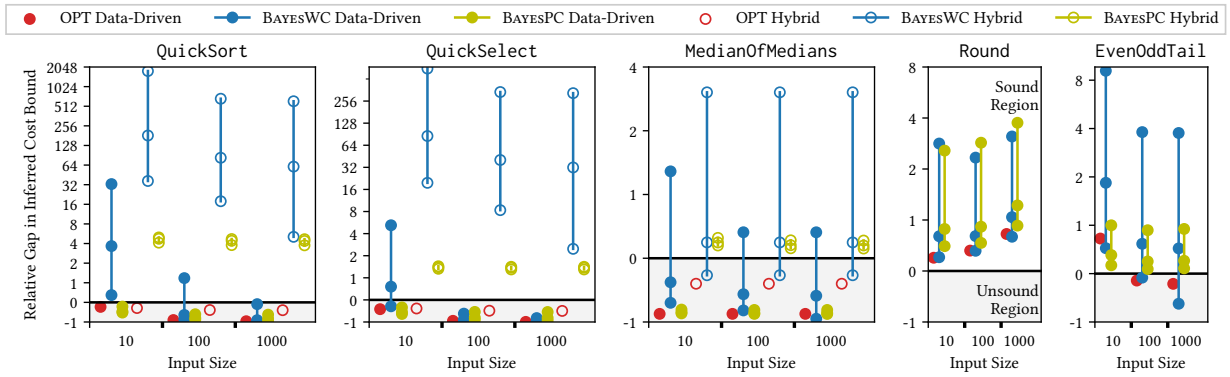


Figure 7.5: Relative estimation errors of inferred cost bounds with respect to ground-truth worst-case costs on 5 benchmarks. Each panel shows the estimation errors for a given benchmark program (subplot title), three input sizes (10, 100, 1000), and six resource-analysis methods (top legend). For BayesWC and BayesPC, the three markers on a vertical line show the 5^{th}, 50^{th}, and 95^{th} percentiles of estimation errors (computed over the posterior distribution of inferred cost bounds). Opt delivers a single estimation error, shown as a red marker. The ideal relative estimation error is zero: errors greater than zero are sound but conservative, and errors less than zero are unsound.

Posterior distributions of cost bounds Fig. 7.6 shows posterior distributions of cost bounds of 5 benchmarks. Red curves are the true bounds, and black dots show the runtime-cost data. Blue curves are median cost bounds, and light-blue shades around them show the 10–90^{th} percentile ranges. In both data-driven and hybrid analyses, Bayesian resource analysis delivers a sizeable fraction of correct bounds. Certain datasets such as MedianOfMedians in Fig. 7.5 are particularly challenging for fully data-driven analysis, however. Between data-driven and hybrid analyses, the cost bounds from the latter are always more *accurate* (i.e., closer to sound worst-case cost bounds).

Fig. 7.7 shows inferred cost bounds for MapAppend, which are multivariate. The median bounds for the Bayesian methods always lie above the ground-truth plane, whereas the single bounds for Data-Driven Opt and Hybrid Opt are both incorrect.

Summary Our evaluation has several key takeaways. First, in the 7/10 programs that Conventional AARA cannot solve (first 7 rows of Tab 7.1), Hybrid AARA successfully returns robust and accurate bounds using at least one of Hybrid BayesWC or Hybrid BayesPC. In the 3/10 programs that Hybrid AARA cannot solve (last 3 rows of Tab 7.1), fully data-driven analysis using BayesWC or BayesPC gives good proportions of sound bounds. Second, Bayesian resource analysis delivers a substantially higher percentage of sound cost bounds as compared to optimization-based analysis in both the fully data-driven and hybrid case. Between BayesWC and BayesPC, the former is more conservative: it gives a larger fraction of sound bounds in the fully data-driven case, but its cost gaps are higher. Third, for Hybrid AARA, incorporating static analysis (when possible) gives a higher proportion of sound bounds than fully data-driven analysis with a similar runtime.

We have no clear winner between Hybrid BayesWC and Hybrid BayesPC: the latter is

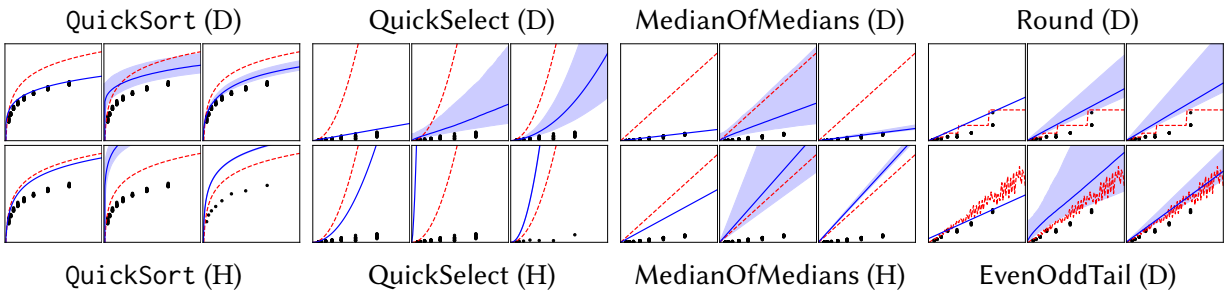
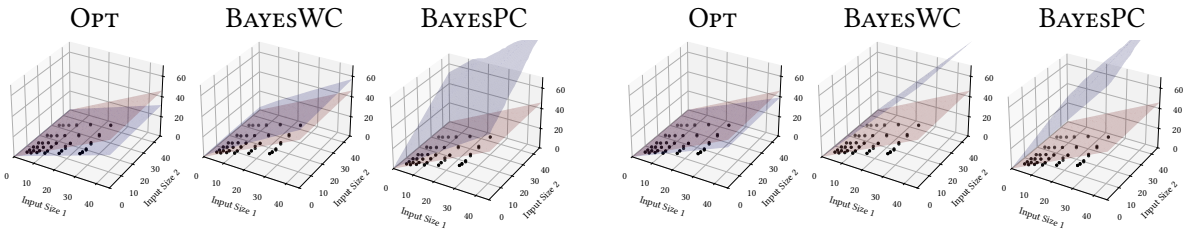


Figure 7.6: Plots of inferred bounds for various benchmarks and resource analysis methods shown in Fig. 7.5 (D=Data Driven; H=Hybrid). Each benchmark has three plots (left-to-right): Opt, BayesWC, and BayesPC.



(a) Data-Driven Resource Analysis

(b) Hybrid Resource Analysis

Figure 7.7: Plots of inferred multivariate cost bounds for MapAppend using data-driven resource analysis and Hybrid AARA. Red planes show ground-truth tight worst-case bounds, and blue planes show inferred bounds.

slower but more accurate than the former in 5/7 cases. Additionally, BayesPC enables richer probabilistic models than BayesWC since the probabilistic model of BayesPC models not only worst-case costs but also resource coefficients. In practice, depending on the application, one can take the looser or tighter result of the two methods.

7.7 Discussion

This chapter has introduced Hybrid AARA, which to the best of my knowledge is the first resource-analysis method that integrates data-driven and static resource analyses via a user-adjustable interface. In this section, I first reflect on the design of Hybrid AARA (§7.7.1). I next discuss usage of Hybrid AARA in practice (§7.7.2) and its application to other program-analysis tasks (§7.7.3). At the end, I describe the limitation of Hybrid AARA (§7.7.4), which motivate the second hybrid resource analysis, resource decomposition (§8).

7.7.1 Interface Design of Hybrid AARA

This section discusses the interface between constituent analysis techniques in Hybrid AARA. Specifically, I first describe what *hybrid* means in Hybrid AARA and how it is different from the sense in which some resource-analysis methods are hybrid. I then discuss why resource-annotated types are a natural choice of the interface in Hybrid AARA.

Hybrid nature of a user-adjustable interface The user-adjustable interface between constituent analysis techniques is a key characteristic of Hybrid AARA. The interface of Hybrid AARA allows the user to freely specify which code fragments are to be analyzed by which analysis techniques. Consequently, Hybrid AARA covers a spectrum ranging from fully data-driven analysis to fully static analysis.

The user-adjustable interface distinguishes Hybrid AARA from other resource-analysis techniques that are also *hybrid* (i.e., they integrate static and data-driven analyses in some way). Several existing resource-analysis techniques [131, 197, 198] sequentially compose data-driven and static analyses in their workflows: they first perform data-driven analysis to guess something, which is then processed by static analysis. For instance, Dynaplex [131] first infers a recurrence relation by data-driven analysis and then solves it statically by the master theorem. Likewise, Rustenholz et al. [197, 198] first infer candidate solutions to recurrence relations by data-driven analysis and then verify them by static analysis. Just like Hybrid AARA, these resource-analysis techniques are considered *hybrid* in that they integrate data-driven and static analyses. However, they differ from Hybrid AARA in the way analysis methods are combined: the former performs different analysis methods in different stages of the analysis workflow, while the latter performs different analysis methods in different code fragments.

Type-based interface The resource-annotated type is an enabler of the user-adjustable interface in Hybrid AARA. Type systems are prominent in the programming-language literature due to their compositionality: the type of a whole program can be derived from the types assigned to constituent code fragments. The compositionality of types contributes to the scalability of type-based program analysis: the analysis of a large, complex program can be broken down into the analysis of smaller code fragments. In Hybrid AARA, its user-adjustable interface must be able to connect two analysis techniques no matter where the user draws a line between the analysis techniques (i.e., which code fragments are analyzed by which techniques). Thus, the interface in Hybrid AARA is required to be compositional, making resource-annotated types a suitable interface.

7.7.2 Using Hybrid AARA in Practice

This section discusses questions about using Hybrid AARA in practice: (i) how to choose a data-driven analysis method in practice; and (ii) how to automate code annotations.

Choice of data-driven analysis This chapter offers three data-driven analyses: Opt (§7.3.2), BayesWC (§7.3.3), and BayesPC (§7.3.4). The choice of a data-driven analysis technique depends on the requirements and domain knowledge of a user. If the user does not need inference

results to be conservative, it is reasonable to use Opt. Also, it is fast because it does not run sampling-based probabilistic inference for Bayesian inference.

Conversely, if the user would like inferred cost bounds to be conservative, it is necessary to add a buffer on top of the maximum observed costs (see Figs. 7.2b and 7.2c). One approach is to first statistically infer how much buffer to add based on observed data $\mathcal{D}$ and then optimize a cost bound while accounting for the inferred buffer. If Bayesian inference is employed to infer the amount of necessary buffer, the resulting data-driven analysis is BayesWC. In Bayesian inference, the user writes a probabilistic model to express their domain knowledge and requirement about how large the buffer should be. If the user has little domain knowledge, they can use broad prior distributions (e.g., normal distributions with large standard deviations).

Alternatively, instead of Bayesian inference, frequentist-statistical methods can be used to infer the amount of buffer. However, frequentist statistics typically does not offer robustness of inferred cost bounds, since it usually returns point estimates (and perhaps confidence intervals), as opposed to posterior distributions returned by Bayesian inference.

As shown in Tab 7.1, BayesWC has longer analysis time than Opt. Since sampling-based probabilistic inference runs many iterations, it is slower than optimization, particularly linear programming. Furthermore, for each posterior sample returned by Bayesian inference, BayesWC solves an optimization problem, although this can be done in parallel.

If the user wishes a fully Bayesian approach, BayesPC is the way to go. In contrast to BayesWC, BayesPC models not only the amount of buffer but also resource annotations (i.e., polynomial coefficients of potential functions) in probabilistic models. To draw posterior cost bounds, BayesPC runs Reflective HMC (ReHMC) [47, 49, 50, 171], which runs Hamiltonian Monte Carlo (HMC) within bounded convex polytopes. ReHMC is implemented in Volesti [48].

Tab 7.1 shows that BayesPC has longer analysis time than BayesWC. This can be because ReHMC is fundamentally more computationally demanding than HMC (e.g., NUTS [111] in Stan) and/or because Volesti is not as optimized as Stan's inference engine. Furthermore, if a bounded convex polytope is narrow, ReHMC may struggle to explore the state space effectively, requiring a long time before converging to a target posterior distribution.

Automation of code annotations In Hybrid AARA, it is the user's responsibility to annotate the source code with stat_ℓ to indicate which code fragments are to be analyzed by data-driven analysis. Nonetheless, it is possible to devise a procedure to automatically insert annotations. A trivial idea is to perform fully data-driven analysis, which is a special case of Hybrid AARA and is fully automatic.

A more sensible approach is to incrementally expand the code fragments to be analyzed by data-driven analysis, until Hybrid AARA can infer a cost bound. In this approach, given a program P, we first run fully static analysis on the whole program P. If it fails, we then apply data-driven analysis to one of the function applications appearing in P's source code. We repeat the step, gradually increasing the number of functions f (and their function applications) inside the program P to be analyzed by data-driven analysis. In the worst case, eventually, the whole program P is analyzed by data-driven analysis, boiling down to fully data-driven analysis.

7.7.3 Hybrid Approaches to Program Analysis beyond Resource Analysis

This section discusses extending the idea of Hybrid AARA beyond resource analysis: (i) hybrid type checking; and (ii) hybrid inference of discrete program properties, particularly types.

Gradual and hybrid typing User-adjustable type-based interfaces show up in the literature of gradual types (also known as hybrid types) [79, 145, 204]. Gradual typing integrates static and dynamic typing, which complement each other. Static typing ensures type safety once and for all at compile time, and produces efficient machine code because it does not require runtime tag-checking of values. On the other hand, dynamic typing for refinement types can check more complex properties of programs than static typing. In gradual typing, a user annotates some (but not necessarily all) code fragments with types. These code fragments are statically checked at compile time, while the rest of the source code is dynamically checked at runtime. Instructions for runtime tag-checking of values are only inserted for dynamically typed code fragments.

Gradual typing and Hybrid AARA are similar in that they both involve types. However, they differ in (i) the goals of program analysis; and (ii) how types are used. In gradual typing, the goal is to *check* some kind of safety (e.g., absence of crashes) of programs. The notion of program safety is encoded by types, which can be standard functional types (e.g., base types and arrow types) [204] or refinement types (i.e., types augmented with predicates) [79, 145]. Thus, types serve as specifications of programs, and they are *checked* by either static analysis or by dynamic analysis (i.e., looking at runtime values). On the other hand, the goal of Hybrid AARA is to *infer* symbolic cost bounds of programs. Hybrid AARA use resource-annotated types to encode potential functions. They are *inferred* by either static analysis or data-driven analysis (i.e., using observed data collected by running programs on concrete inputs). In summary, gradual typing concerns program safety, while Hybrid AARA concerns symbolic cost bounds. Additionally, gradual typing is about type checking, while Hybrid AARA is about type inference.

Inference of discrete program properties One might wonder about developing Hybrid-AARA-like approaches to program-analysis tasks beyond resource analysis. As the search space of such program-analysis tasks is typically discrete, two challenges arise:

1. It is difficult to justify the use of data-driven methods (and hybrid ones);
2. Effective gradient-based inference algorithms are no longer applicable.

In Hybrid AARA, the search space consists of symbolic cost bounds (or more precisely, polynomial coefficients of potential functions). The continuity of this search space makes it easy to justify the use of data-driven methods. For example, consider a program P whose ground-truth cost bound is n. If data-driven analysis infers a close but unsound bound (e.g., $0.9n$ or $0.99n$), the user will likely tolerate it, unless the program P is a safety-critical program. This is fundamentally because the space of symbolic cost bounds is continuous: as the error between inferred bounds and ground-truth bounds is continuous, the user does not outright reject unsound bounds as long as they are close to the ground truths.

Meanwhile, in many program-analysis tasks, such as traditional type inference, the search

space is discrete, potentially making the user less willing to accept unsound inference results. For instance, given a program P whose ground-truth type is $\texttt{int} \to \texttt{int}$, suppose a type-inference engine infers an unsound type (e.g., $\texttt{int} \to \texttt{unit}$). Then the user may reject it outright even if it is unknown whether the inferred bound is sound or unsound. This is because the error between $\texttt{int} \to \texttt{int}$ and $\texttt{int} \to \texttt{unit}$ is discrete: it is either correct or wrong, and we have no alternative inference results in between.

Alternatively, data-driven analysis can return a distribution of inferred types, as does Bayesian inference. Again, the user may be unhappy with the distribution returned by data-driven analysis, unless the analysis offers the absolute (i.e., 100%) guarantee of sound type inference. Otherwise, the 99% guarantee of soundness may not be enough to satisfy the user's demand.

Furthermore, continuous search spaces have the benefit that efficient gradient-based inference algorithms (e.g., HMC [111] adopted in Stan [44]) are readily available. On the other hand, discrete search spaces are not supported in some probabilistic programming languages, including Stan [44]. Gen [63] supports discrete as well as continuous random variables, but a user usually needs to customize probabilistic-inference algorithms to quickly converge to target posterior distributions.

Nonetheless, data-driven inference of discrete program properties (e.g., types and names of variables) have been actively investigated. For example, Raychev et al. [193] first train probabilistic graphical models using code repositories and then use the trained graphical models to infer identifiers' names and types in JavaScript programs. Also, numerous works employ neural networks to infer types in dynamically typed programming languages (e.g., Python and JavaScript) [12, 13, 107, 183, 184, 225].

7.7.4 Limitations of Hybrid AARA

Hybrid AARA has two limitations:

1. It can only express and infer polynomial cost bounds;
2. Its constituent analysis techniques must all infer quantities of the same resource metric.

These limitations stem from the use of resource-annotated types as an interface between constituent analysis techniques' inference results.

Resource-annotated types only encode polynomial potential functions. It is challenging to extend Conventional AARA to infer symbolic bounds where polynomials and logarithm coexist. This is fundamentally because it is difficult to define a set of symbolic bounds that (i) admit both polynomials and logarithm; (ii) are closed under some operations, such as resource-annotation sharing $\curlyvee$ (Listing 4.3) and the shift operator $\triangleleft$ (Eq (4.2.8)); and (iii) are amenable to automated reasoning. Consequently, Hybrid AARA cannot express, let alone infer, an asymptotically tight $O(n \log n)$ cost bound of MergeSort.

The second limitation arises because, in Hybrid AARA, the constituent analyses' inference results are encoded by resource-annotated types of code fragments and are composed by their substitution. Otherwise, if Analysis A inferred a running-time bound of some code fragment and Analysis B inferred a memory-usage bound of the rest of the source code, it would be impossible to compose Analyses A's and B's resource-annotated types.

Hybrid AARA is not the only way to integrate resource analyses. Another idea for hybrid

resource analysis is to break down an overall cost bound of a recursive program P into a product of: (i) the cost of a single recursive step of P; and (ii) the number of recursive calls to P. These two quantities have different resource metrics: the first is the cost of a code fragment, while the second is the number of function calls. This alternative idea for hybrid resource analysis leads to the development of the second hybrid resource analysis: resource decomposition (§8).

Chapter 8

Resource Decomposition

This chapter introduces the second hybrid resource analysis—resource decomposition—that integrates resource-analysis techniques via a different interface from Hybrid AARA (§7).

Fig. 2.2 shows the workflow of resource decomposition. Given a target program $P(x)$, a user annotates it to specify $n \geq 1$ many custom quantities called *resource components* (e.g., the recursion depth of a function and the cost of a code fragment). One resource-analysis technique analyzes the annotated program $P_{\mathrm{rd}}(x)$, called a *resource-decomposed program*, to infer high-water-mark bounds $g_i(x)$ $(i = 1, \ldots, n)$ on the resource components. The resource-decomposed program $P_{\mathrm{rd}}(x)$ is (automatically) extended with numeric input variables $\mathbf{r} = (r_1, \ldots, r_n)$, called *resource guards*, which track the user-specified resource components. The extended program $P_{\mathrm{rg}}(x, \mathbf{r})$ is called a *resource-guarded program*. Another resource-analysis technique analyzes the program $P_{\mathrm{rg}}(x, \mathbf{r})$ to infer its cost bound $f(x, r_1, \ldots, r_n)$ parametric in both the original input x and the resource guards $\mathbf{r}$. Finally, the inferred bounds $f(x, r_1, \ldots, r_n)$ and $g_i(x)$ $(i = 1, \ldots, n)$ are composed by substitution, resulting in $f(x, g_1(x), \ldots, g_n(x))$.

Thanks to the numeric-variable interface, resource decomposition overcomes[1] the two drawbacks of Hybrid AARA in §7.7.4. Furthermore, resource decomposition is more versatile than Hybrid AARA in terms of the diversity of resource analyses that can be integrated. While Hybrid AARA is more or less tied to Conventional AARA, resource decomposition can integrate analysis techniques beyond Conventional AARA.

§8.1 introduces a programming language RPCF that extends PCF with monadic effects for computational cost and resource components. §8.2 then describes the domain-theoretic denotational semantics of RPCF. §8.3 formalizes the automatic program transformation of resource-decomposed programs $P_{\mathrm{rd}}(x)$ to corresponding resource-guarded programs $P_{\mathrm{rg}}(x, \mathbf{r})$. §8.4 formulates a soundness theorem of the program transformation, which is then proved by a logical-relation argument in the domain-theoretic denotational semantics. §8.5–8.7 describe three instantiations of the resource-decomposition framework, which each integrate different pairs of static, data-driven, and interactive resource analyses. Finally, §8.8 discusses the design of resource composition and compares it with Hybrid AARA.

[1] Resource decomposition has its own drawbacks that Hybrid AARA does not have, as discussed in §8.8.3. Hence, it does not strictly improve on Hybrid AARA.

$A ::= \mathtt{unit} \mid \mathtt{int} \mid A_1 + A_2 \mid A_1 \times A_2 \mid L(A) \mid A_1 \to A_2$	value types
$\mid \mathsf{F}A$	computation type
$e ::= \langle\rangle \mid z$	unit and integer; $z \in \mathbb{Z}$
$\mid x$	variable; $x \in \mathcal{X}$
$\mid \mathsf{left} \cdot e \mid \mathsf{right} \cdot e \mid \mathsf{case}\ e\ \{\mathsf{left} \cdot x_1 \hookrightarrow e_1 \mid \mathsf{right} \cdot x_2 \hookrightarrow e_2\}$	sums
$\mid \langle e_1, e_2\rangle \mid \mathsf{case}\ e\ \{\langle x_1, x_2\rangle \hookrightarrow e_1\}$	products
$\mid [\,] \mid e_1 :: e_2 \mid \mathsf{case}\ e\ \{[\,] \hookrightarrow e_1 \mid (x_1 :: x_2) \hookrightarrow e_2\}$	lists
$\mid \mathsf{fun}\, f\, x = e \mid e_1\, e_2$	functions; $f \in \mathcal{X}$
$\mid \mathsf{return}\, e$	return statement
$\mid \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$	let-binding
$\mid \mathsf{tick}\ q$	resource consumption; $q \in \mathbb{Q}$
$\mid \mathsf{mark}_i \mid \mathsf{unmark}_i \mid \mathsf{reset}_i$	resource components; $i = 1, \ldots, n$

Lst. 8.1: Types A and expressions e in $\textsc{Rpcf}_n$.

8.1 Programming Language

To formulate the hybrid resource analysis resource decomposition, this section introduces a resource-sensitive programming language Rpcf. Rpcf extends call-by-value Pcf by Plotkin [191] with lists and two monads for tracking: (i) computational cost indicated by the construct tick; and (ii) user-defined resource components indicated by constructs mark, unmark, and reset. These two quantities are treated as computational effects captured by monads.

I describe the syntax (§8.1.1), a type system (§8.1.2), and domain-theoretic denotational semantics of Rpcf (§8.2). The denotational semantics is used in the formalization and soundness proof of resource decomposition (§8.3).

8.1.1 Syntax

Suppose an input program has $n \in \mathbb{N}$ many resource components $r_1, \ldots, r_n$. I define a programming language $\textsc{Rpcf}_n$ that extends the vanilla Pcf by Plotkin [191] with lists and constructs for indicating resource usage and resource components. The language $\textsc{Rpcf}_n$ is parametrized by $n \in \mathbb{N}$ because, in its denotational semantics, the number n determines dimensions of vectors tracking resource components. The syntax and type system of $\textsc{Rpcf}_n$ (or Rpcf more generally), on the other hand, need not be parametrized by n.

Fix a countable set $\mathcal{X}$ of variable symbols. Listing 8.1 defines types A and expressions e of $\textsc{Rpcf}_n$ ($n \in \mathbb{N}$). The language is adapted from the call-by-value language $\mathrm{PCF}_{\mathrm{v}}$ in Harper [106].

Types The language $\textsc{Rpcf}_n$ distinguishes between values and computational by types. Value types (e.g., $\mathtt{int}$ and $A_1 \to A_2$) capture *values*, namely expressions that do not further evaluate.

On the other hand, a computation type $\mathsf{F}A$, where F is a modality and A is a type, captures *effectful computations*, namely expressions that spontaneously evaluate and give rise to computational effects (e.g., non-termination and computational costs). In resource decomposition, the monadic type $\mathsf{F}A$ captures three computational effects: non-termination, computational cost, and resource components r_i ($i = 1, \ldots, n$).

Expressions The construct $\mathsf{return}\ e$ is a constructor of a computation type $\mathsf{F}A$: it encapsulates an expression e of type A inside a computation. Dually, the let-binding $\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$ acts as a destructor of the computation type $\mathsf{F}A$: it first runs a computation e_1, then binds its value (if any) to a variable x, and then proceeds with a computation e_2. The constructs $\mathsf{return}\ e$ and $\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$ serve as the return and bind operators for the monad captured by F.

The construct $\mathsf{tick}\ q$ ($q \in \mathbb{Q}$) increments the cost counter by q and returns the unit element $\langle\rangle$. This construct has already been introduced by the programming language in §3, which serves as a basis for Conventional AARA (§4) and Hybrid AARA (§7).

Unlike the programming language introduced in §3, the language $\textsc{Rpcf}_n$ does not have a construct $\mathsf{share}\ x\ \mathsf{as}\ x_1, x_2\ \mathsf{in}\ e$ for variable sharing. Also, $\textsc{Rpcf}_n$ allows targets of pattern matching to be general expressions e, instead of restricting them to variables $x \in \mathcal{X}$.

The construct mark_i (resp., unmark_i) increases (resp., decreases) the counter of a resource component r_i ($i = 1, \ldots, n$) by one. The constructs mark_i and unmark_i must be inserted in such a way that their high-water mark (i.e., the highest value ever reached) is equal to a resource component r_i specified by the user. For example, to set a resource component to the recursion depth of a recursive function P, the user inserts (i) mark at the beginning of P's function body to increment the recursion-depth counter by one; and (ii) unmark at the end of the function body to decrement the recursion-depth counter by one.

Lastly, the construct reset_i resets the counter of a resource component r_i ($i = 1, \ldots, n$) to 0. This construct is useful when a resource component is defined for *each individual call* to the function. For example, given a function f inside a program P, consider a resource component r equal to the cost of the function f. If the function f is called multiple times during the execution of the program P, the resource component r has multiple measurements, one for each call to the function f. To indicate the resource component r in the source code, it is necessary to reset the resource component r's counter after each call to the function f. Otherwise, the resource component r would track the *combined cost across all calls* to the function f, which is different from the intended quantity.

More concretely, consider a target program P defined as

$$\mathsf{fun}\ P\ (\ell, t) = \mathsf{map}\ \ell\ (\mathsf{fun}\ f\ x = (\mathtt{lookup}\ (t, x); \mathsf{reset}_1)). \tag{8.1.1}$$

Suppose a resource component r_1 of interest is the cost of the function `lookup` that looks up an element x in a tree t. The program P in Eq (8.1.1) takes two inputs, a list ℓ and a tree t. For each element $x \in \ell$ in the list, the program P looks up the element x in the tree t by the function call $\mathtt{lookup}\ (t, x)$. Since the function `lookup` is called multiple times in an execution, we must reset the counter of the resource component r_1 to 0 at the end of each function call of `lookup`. To this end, we sequentially compose[2] the function application $\mathtt{lookup}\ (t, x)$ with reset_1.

[2] Sequential composition is indicated by the semicolon in Eq (8.1.1).

$$\frac{}{\Gamma \vdash \langle\,\rangle : \mathsf{unit}} \qquad \frac{z \in \mathbb{Z}}{\Gamma \vdash z : \mathsf{int}} \qquad \frac{\Gamma(x) = A}{\Gamma \vdash x : A} \qquad \frac{\Gamma \vdash e : A_1}{\Gamma \vdash \mathsf{left} \cdot e : A_1 + A_2} \qquad \frac{\Gamma \vdash e : A_2}{\Gamma \vdash \mathsf{right} \cdot e : A_1 + A_2}$$

$$\frac{\Gamma \vdash e : A_1 + A_2 \qquad \Gamma, x_1 : A_1 \vdash e_1 : A_3 \qquad \Gamma, x_2 : A_2 \vdash e_2 : A_3}{\Gamma \vdash \mathsf{case}\ e\ \{\mathsf{left} \cdot x_1 \hookrightarrow e_1 \mid \mathsf{right} \cdot x_2 \hookrightarrow e_2\} : A_3} \qquad \frac{\Gamma \vdash e_1 : A_1 \qquad \Gamma \vdash e_2 : A_2}{\Gamma \vdash \langle e_1, e_2 \rangle : A_1 \times A_2}$$

$$\frac{\Gamma \vdash e : A_1 \times A_2 \qquad \Gamma, x_1 : A_1, x_2 : A_2 \vdash e_1 : A_3}{\Gamma \vdash \mathsf{case}\ e\ \{\langle x_1, x_2 \rangle \hookrightarrow e_1\} : A_3} \qquad \frac{}{\Gamma \vdash [\,] : L(A)} \qquad \frac{\Gamma \vdash e_1 : A \qquad \Gamma \vdash e_2 : L(A)}{\Gamma \vdash e_1 :: e_2 : L(A)}$$

$$\frac{\Gamma \vdash e : L(A_1) \qquad \Gamma \vdash e_1 : A_2 \qquad \Gamma, x_1 : A_1, x_2 : L(A_1) \vdash e_2 : A_2}{\Gamma \vdash \mathsf{case}\ e\ \{[\,] \hookrightarrow e_1 \mid (x_1 :: x_2) \hookrightarrow e_2\} : A_2} \qquad \frac{\Gamma, f : A_1 \to \mathsf{F}A_2, x : A_1 \vdash e : \mathsf{F}A_2}{\Gamma \vdash \mathsf{fun}\ f\ x = e : A_1 \to \mathsf{F}A_2}$$

$$\frac{\Gamma \vdash e_1 : A_1 \to A_2 \qquad \Gamma \vdash e_2 : A_1}{\Gamma \vdash e_1\ e_2 : A_2} \qquad \frac{\Gamma \vdash e : A}{\Gamma \vdash \mathsf{return}\ e : \mathsf{F}A} \qquad \frac{\Gamma \vdash e_1 : \mathsf{F}A_1 \qquad \Gamma, x : A_1 \vdash e_2 : \mathsf{F}A_2}{\Gamma \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : \mathsf{F}A_2}$$

$$\frac{q \in \mathbb{Q}}{\Gamma \vdash \mathsf{tick}\ q : \mathsf{Funit}} \qquad \frac{1 \le i \le n}{\Gamma \vdash \mathsf{mark}_i : \mathsf{Funit}} \qquad \frac{1 \le i \le n}{\Gamma \vdash \mathsf{unmark}_i : \mathsf{Funit}} \qquad \frac{1 \le i \le n}{\Gamma \vdash \mathsf{reset}_i : \mathsf{Funit}}$$

Lst. 8.2: Type system of $\textsc{Rpcf}_n$.

The constructs tick, mark, and unmark have similar behaviors: they manipulate some kind of counters. Indeed, we can view mark as tick 1 and unmark as tick -1. However, they differ as follows. Firstly, they serve different purposes. The construct tick defines the computational cost of an input program, whose symbolic bound is the objective of resource analysis. Meanwhile, the constructs mark and unmark define resource components to decompose resource analysis into two analysis techniques via the resource-decomposition framework. Secondly, in a transformation from a resource-decomposed program to a resource-guarded one, the construct tick is left unchanged so that the target program of the transformation has the same cost as the source program. Meanwhile, mark and unmark are deleted by the program transformation.

Remark 8.1.1 (Integer-valued resource components)**.** *The construct* tick *q increments computational cost by rational numbers $q \in \mathbb{Q}$, whereas* mark *and* unmark *can only increment resource components by integers $z \in \mathbb{Z}$. This limitation is due to the fact that I use natural numbers, particularly their unary encoding as lists, to encode resource guards (§8.3). This design decision allows my collaborators and me to reuse RaML [117, 118] in the instantiations of resource decomposition that combine AARA and other analysis techniques (§8.5 and §8.6). To extend resource components' encodings from natural numbers to rational numbers, the implementation of RaML [117, 118] would need to be extensively modified.* «

8.1.2 Type System

A typing judgment of $\textsc{Rpcf}_n$ is given by

$$\Gamma \vdash e : A, \tag{8.1.2}$$

where Γ is a typing context (i.e., a mapping from variables to types), e is an expression, and A is a type. Listing 8.2 defines this typing judgment.

The typing rules for sums, products, and lists are standard [105]. The function definition $\mathsf{fun}\, f\, x = e$ has type $A_1 \to \mathsf{F}A_2$. The output type $\mathsf{F}A_2$ is monadic because a function may entail computational effects (i.e., non-termination, computational cost, and resource components) when applied to inputs. The expression $\mathsf{return}\, e$ creates a trivial computation that encapsulates the expression e. So if the expression e has type A, the expression $\mathsf{return}\, e$ has type $\mathsf{F}A$. Dually, the let-binding $\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$ runs a computation $e_1 : \mathsf{F}A_1$, binds its value to a variable $x : A_1$, and then runs a computation $e_2 : \mathsf{F}A_2$ that may use the variable x. The typing rules for functions, return expressions, and let-bindings are adapted from Harper [106].

Those constructs that manipulate cost counters and resource components' counters (e.g., $\mathsf{tick}\ q$ for $q \in \mathbb{Q}$ and mark_i for $i = 1, \ldots, n$) have the monadic type $\mathtt{Funit}$. This is reasonable because these constructs entail computational effects and return the unit element $\langle\rangle$.

8.2 Denotational Semantics

This section defines domain-theoretic denotational semantics of $\textsc{Rpcf}_n$ by extending the standard interpretation of $\textsc{Pcf}$ with the denotations of those constructs that manipulate costs and resource components (i.e., tick, mark, unmark, and reset) (§8.1.1).

§8.2.1 describes mathematical objects to capture the semantics of computational cost and resource components. Next, §8.2.2 gives an introduction to domain theory. Finally, §8.2.3 formulates the domain-theoretic denotational semantics of $\textsc{Rpcf}_n$.

8.2.1 Computational Cost and Resource Components

Computational cost Recall from Defn. 3.2.1 that computational cost is represented by elements of the resource monoid $\mathsf{RM} := (\mathbb{Q}^2_{\geq 0}, (0,0), \oplus)$. In a resource-monoid element $(h, r) \in \mathbb{Q}^2_{\geq 0}$, the value h is the high-water-mark cost, and r is the amount of remaining resources. The net cost is given by $h - r$.

Notation (Extracting high-water marks and remaining resources)**.** *Given a resource-monoid element $x \in \mathbb{Q}^2_{\geq 0}$, if $(h, r) = x$, I write $x.h$ for the high-water-mark cost h and $x.r$ for the remaining resources r.*

In the denotational semantics of the language $\textsc{Rpcf}_n$, the cost of an expression is denoted by an element from the resource monoid RM. To compose the costs of two sequentially composed expressions, we use the binary operator $\oplus$ of the resource monoid RM.

Resource components The semantics of resource components, which are defined by constructs mark, unmark, and reset, cannot be fully captured by the natural numbers' equivalent of the resource monoid: $(\mathbb{N}^2, (0,0), \oplus)$. The constructs mark and unmark have additive nature: they add some constants (1 and -1, respectively) to the counter of a resource component. This additive nature makes the two constructs amenable to the resource monoid. Meanwhile, the construct reset has non-additive nature: it resets a counter to zero regardless of the counter's current value.

To this end, a resource component' semantics is encoded by a function $f : \mathbb{N}^2 \to \mathbb{N}^2$, called a *resource effect*. Consider an expression e that entails computational effects, particularly effects arising from resource components. The input $(h_1, r_1) \in \mathbb{N}^2$ to the function f stores the initial reading of a resource component's counter: a high-water mark h_1 and the remaining resources r_1 of the counter before the expression e is executed. That is, the current value (i.e., net cost[3]) of the resource component's counter is $h_1 - r_1$, and the highest value reached so far (i.e., high-water-mark cost) before the execution of e is h_1.

The expression e is then executed, and the resource component's counter is manipulated by constructs mark_i, unmark_i, and reset_i ($i = 1, \ldots, n$). The construct mark_i increments the counter by one; i.e., it consumes and deducts one unit of the remaining resources r (if $r \geq 1$). If the counter's reading surpasses the highest value reached previously (i.e., if the remaining resources r are not enough to pay for mark), the high-water mark h is incremented. The construct unmark has the dual behavior: it decrements the counter by one (i.e., increments the remaining resources r by one). Finally, the output (h_2, r_2) of the resource effect f stores the final high-water mark h_2 and the remaining resources r_2 after the expression e has been executed.

Definition 8.2.1 (Resource effect)**.** *A function $f : \mathbb{N}^2 \to \mathbb{N}^2$ is a resource effect when it is inflationary in the first component, that is, $(f\,x).h \geq x.h$ for all $x \in \mathbb{N}^2$.*

Resource effects are required to be inflationary in the first argument to enforce the invariant that the high-water mark of a computation is monotone with respect to execution.

Definition 8.2.2 (Resource-effect monoid)**.** *The resource-effect monoid* REM *is*

$$\mathsf{REM} \coloneqq ([\mathbb{N}^2 \to \mathbb{N}^2]_{\mathrm{infla}}, \diamond, \circ), \tag{8.2.1}$$

where $[\mathbb{N}^2 \to \mathbb{N}^2]_{\mathrm{infla}}$ is the set of resource effects (i.e., functions $f : \mathbb{N}^2 \to \mathbb{N}^2$ that are inflationary in the first component), $\diamond : x \mapsto x$ is the identity function, and a binary operator $\circ$ is function composition.

Definition 8.2.3 (Resource effects for resource components)**.** *The constructs* mark, unmark, *and* reset *are interpreted as the following resource effects:*

$$\wr : \mathbb{N}^2 \to \mathbb{N}^2 \qquad \wr : \mathbb{N}^2 \to \mathbb{N}^2 \qquad \circlearrowleft : \mathbb{N}^2 \to \mathbb{N}^2 \tag{8.2.2}$$

$$\wr(h, r) = (h, r) \oplus (1, 0) \qquad \wr(h, r) = (h, r) \oplus (0, 1) \qquad \circlearrowleft(h, r) = (h, h). \tag{8.2.3}$$

In the denotational semantics, I interpret each of mark_i, unmark_i, and reset_i by a tuple $\sigma = (f_1, \ldots, f_n)$ of resource effects in which the i^{th} component is the respective function in Defn. 8.2.3 and the other components are the identity resource effect $\diamond$ (Defn. 8.2.3).

8.2.2 Domain Theory

This section introduces the basics of domain theory, which is used in the denotational semantics of PCF and RPCF_n. The material in this section is adapted from Abramsky and Jung [3], Pitts et al. [190], Streicher [212].

[3]The current value of a resource component's counter is considered as a net cost if we view a resource component (e.g., the recursion depth of a recursive function) as a resource metric. Indeed, the constructs mark and unmark are equivalent to tick 1 and tick -1. However, I do not use the terminology net costs and high-water-mark costs in the context of resource components, because they may misleadingly refer to the computational cost defined by a construct tick q ($q \in \mathbb{Q}$) in the source code.

Partially ordered sets I first introduce partially ordered sets (posets), which are sets equipped with partial orders.

Definition 8.2.4 (Partially ordered set)**.** *Given a set D, a binary relation $\sqsubseteq \subseteq D \times D$ is a partial order if it satisfies the following conditions:*

- *Reflexive:* $\forall d \in D.d \sqsubseteq d$*;*
- *Transitive:* $\forall d_1, d_2, d_3 \in D.d_1 \sqsubseteq d_2 \wedge d_2 \sqsubseteq d_3 \implies d_1 \sqsubseteq d_3$*; and*
- *Antisymmetric:* $\forall d_1, d_2 \in D.d_1 \sqsubseteq d_2 \wedge d_2 \sqsubseteq d_1 \implies d_1 = d_2$.

A pair $(D, \sqsubseteq)$ of a set D and a partial order $\sqsubseteq$ is called a partially ordered set (poset).

In denotational semantics of programs, partial orders capture the ordering between programs in terms of their amount of information content: how much of their input-output behaviors are defined. For example, a program P never terminates for all inputs, the program P carries no information when viewed as a (partial) function because, for any input x, the output $P(x)$ is undefined. Conversely, if a program Q terminates for all inputs, then this program carries more information than the program P. The ordering based on the amounts of information content is called information ordering [190, 212].

Definition 8.2.5 (Upper and lower bounds)**.** *Given a poset $(D, \sqsubseteq)$, consider a subset $A \subseteq D$. A lower bound $d \in D$ of the subset A satisfies $\forall a \in A.d \sqsubseteq a$. Dually, an upper bound $d \in D$ of the subset A satisfies $\forall a \in A.a \sqsubseteq d$.*

Definition 8.2.6 (Least and largest elements)**.** *Given a poset $(D, \sqsubseteq)$, consider a subset $A \subseteq D$. The least element (also known as the bottom element) of the subset A is $\bot_A \in A$ such that $\forall a \in A.\bot_A \sqsubseteq a$. If the (sub)set we are talking about is clear, it is customary to simply denote the least element by $\bot$. The largest element of the subset A is defined dually.*

Definition 8.2.7 (Least upper bound and greatest lower bound)**.** *Given a poset $(D, \sqsubseteq)$, consider a subset $A \subseteq D$. The least upper bound (also known as supremum and join) of the subset A, denoted by $\bigvee A$, is the least element among all upper bounds of A. The greatest lower bound (also known as infimum and meet), denoted by $\bigwedge A$, is defined dually.*

Directed complete partial orders Building on posets, I next introduce directed complete partial orders (dcpos), which are used to represent the semantics of programs.

Definition 8.2.8 (Pointed poset)**.** *Given a poset $(D, \sqsubseteq)$, it is said to be pointed if it has a least element $\bot \in D$.*

Definition 8.2.9 (Directed set)**.** *Given a poset $(D, \sqsubseteq)$, consider a subset $A \subseteq D$. The subset A is said to be directed if every pair of elements in A has an upper bound in A as well. If the subset A is directed, it must be non-empty because the empty set $\emptyset$ cannot contain an upper bound of its elements.*

A simple example of a directed set is an ω-chain [3], which is an infinite chain $(d_i)_{i \in \mathbb{N}}$ such that $d_i \sqsubseteq d_{i+1}$ for all $i \in \mathbb{N}$.

Definition 8.2.10 (Directed complete partial order)**.** *A poset $(D, \sqsubseteq)$ is a directed complete partial order (dcpo; also known as a predomain) if every directed subset $A \subseteq D$ has a least upper bound in D.*

Given an ω-chain $(d_i)_{i \in \mathbb{N}}$, which is a directed set, a dcpo must contain the supremum $\bigvee_{i \in \mathbb{N}} d_i$. This property is necessary for the denotational semantics of fixed-point expressions (i.e., gen-

eral recursion in functional programming and while-loops in imperative programming). Given a fixed-point expression, its semantics can be approximated by unrolling the fixed-point expression finitely many times. Put differently, the fixed-point expression's semantics is given by the supremum of the chain $(d_i)_{i\in\mathbb{N}}$, where d_i is the semantics of unrolling the fixed-point expression $i \in \mathbb{N}$ many times. Hence, it is crucial to use a poset $(D, \sqsubseteq)$ where every ω-chain has a supremum. Otherwise, no element from the poset $(D, \sqsubseteq)$ might be able to serve as a denotation of a fixed-point expression.

Finally, domains are dcpos equipped with least elements, which represent computation with undefined values (e.g., non-terminating computation).

Definition 8.2.11 (Domain)**.** *A dcpo $(D, \sqsubseteq)$ is called a domain if it is pointed (i.e., it has the least element $\bot \in D$). A domain is also simply known as a pointed dcpo.*

The literature of domain-theoretic denotational semantics uses various definitions of complete partial orders (cpos) in the development of denotational semantics. For example, Milner [169] defines cpos as posets that are (i) pointed (i.e., contain the least elements); and (ii) chain–complete (i.e., every ω-chain has a supremum). Chain-completeness is a weaker condition[4]than directed completeness (Defn. 8.2.10). Meanwhile, Plotkin [191] defines cpos as pointed dcpos (i.e., domains defined in Defn. 8.2.11). In more modern presentations of denotational semantics, Abramsky and Jung [3], de Jong [71], Niu [179], Streicher [212] use dcpos without requiring them to be pointed. Pitts et al. [190], on the other hand, use cpos defined as chain-complete posets without requiring them to be pointed.

The choice between chain-complete partial orders and directed complete partial orders is discussed in §2.2.4 of Abramsky and Jung [3]. Dcpos are more desirable than chain-complete partial orders because the latter uses non-constructive mathematical reasoning (e.g., the axiom of choice) [3, §2.2.4].

Continuous functions In denotational semantics, fixed-point expressions are interpreted as fixed points of functions from dcpos to dcpos. To guarantee the existence of fixed points, we should not consider arbitrary functions. Instead, we should narrow our scope to those functions that are *continuous*. Continuous functions turn out to be suitable mathematical objects capturing programs (i.e., computable functions).

Definition 8.2.12 (Fixed point)**.** *Given a poset $(D, \sqsubseteq)$, consider a function $f : D \to D$. A fixed point $d \in D$ of the function f is such that $f(d) = d$. The least fixed point $\mathsf{fix}(f) \in D$ of the function f is the least element among all fixed points of the function f.*

Definition 8.2.13 (Continuous function)**.** *Given two dcpos $(D_1, \sqsubseteq_1)$ and $(D_2, \sqsubseteq_2)$, a function $f : D_1 \to D_2$ is said to be (Scott-)continuous if, for every directed set $A \subseteq D_1$, we have*

$$f\left(\bigvee A\right) = \bigvee_{a\in A} f(a). \tag{8.2.4}$$

Definition 8.2.14 (Dcpo of continuous functions)**.** *Consider two dcpos $(D_1, \sqsubseteq_1)$ and $(D_2, \sqsubseteq_2)$. The set of all continuous functions from D_1 to D_2 forms a dcpo where the partial order is*

$$f_1 \sqsubseteq f_2 \iff (\forall d \in D_1. f_1(d) \sqsubseteq_2 f_2(d)). \tag{8.2.5}$$

[4]In fact, chain-completeness and directed completeness can be proved to be equivalent. But its proof requires the axiom of choice, making the proof non-constructive [3, 161, 232].

This dcpo is denoted by $[D_1 \to D_2]_{\text{cont}}$.

Theorem 8.2.1 (Fixed-point theorem)**.** *Consider a domain* $(D, \sqsubseteq)$ *with the least element* $\bot \in D$*. A continuous function* $f : D \to D$ *has the least fixed point*

$$\mathsf{fix}(f) = \bigvee_{i \in \mathbb{N}} f^i(\bot). \tag{8.2.6}$$

To prove a property P about a fixed point of a function f in denotational semantics, a common technique is to prove that the property P is *admissible* and that the property P is closed under the function f. This proof technique is known as *fixed-point induction* [3, 212].

Definition 8.2.15 (Admissible predicate)**.** *Consider a domain* $(D, \sqsubseteq)$ *with the least element* $\bot \in D$*. A subset (also called a predicate)* $P \subseteq D$ *is said to be admissible if* P *contains* $\bot$ *and* P *is closed under suprema of* ω*-chains: given a chain* $(d_i)_{i \in \mathbb{N}}$*, we have*

$$(\forall i \in \mathbb{N}. d_i \in P) \implies \left(\bigvee_{i \in \mathbb{N}} d_i \right) \in P. \tag{8.2.7}$$

Theorem 8.2.2 (Fixed-point induction)**.** *Given a domain* $(D, \sqsubseteq)$ *with the least element* $\bot \in D$*, and an admissible predicate* $P \subseteq D$*, a continuous function* $f : D \to D$*. If the predicate* P *is closed under the function* f *(i.e.,* $\forall x \in D. x \in P \implies f(x) \in P$*), then* $\mathsf{fix}(f) \in P$ *holds.*

Lifting To define the denotation of a monadic type $\mathsf{F}A$ capturing non-termination, the dcpo corresponding to the type $\mathsf{F}A$ is extended with a new least element $\bot$ to represent non-termination. The act of extending a dcpo this way is called lifting, and the resulting domain is called a lift.

Definition 8.2.16 (Lift)**.** *Given a dcpo* $(D, \sqsubseteq)$*, its lift is a domain* $(D_\bot, \sqsubseteq_\bot)$*, where*

$$D_\bot \coloneqq D \cup \{\bot\}. \tag{8.2.8}$$

The symbol $\bot$ *is assumed to be fresh (i.e., it is not used to denote any element in* D*). The partial order* $\sqsubseteq_\bot$ *of the lift is defined as*

$$x_1 \sqsubseteq_\bot x_2 \iff x_1 = \bot \vee (x_1, x_2 \in D \wedge x_1 \sqsubseteq x_2). \tag{8.2.9}$$

The lift defined in Defn. 8.2.16 entails the limited principle of omniscience (LPO), which is deemed non-constructive [71, §3.4]. To define the lift in a more constructive manner, the partial map classifier is used in the literature of topos theory, constructive domain theory, and constructive type theory [71, 179]. In this thesis, however, the focus is not to present a fully constructive formulation of resource decomposition. So I use Defn. 8.2.16 for simplicity.

Lifting can be framed as a monad in the category of dcpos [3, §3.2.5]. For those readers who are familiar with category theory, consider two categories:

1. the category **DCPO** formed by dcpos and continuous functions; and
2. the category $\mathbf{DCPO}_{\bot!}$ formed by domains (i.e., dcpos with bottom/least elements) and continuous strict functions, where functions from domains to domains are said to be *strict* if they preserve the least elements.

Lifting is a functor $\mathsf{L} : \mathbf{DCPO} \to \mathbf{DCPO}_{\perp!}$. Furthermore, together with the inclusion functor $\mathsf{U} : \mathbf{DCPO}_{\perp!} \to \mathbf{DCPO}$, the functor L forms a free-forgetful adjunction $\mathsf{L} \dashv \mathsf{U}$. Consequently, the composition $\mathsf{L} \circ \mathsf{U}$ of the functors yields the *lifting monad* $\mathbb{L} \coloneqq (\mathsf{L}, \eta_{\mathbb{L}}, \mu_{\mathbb{L}})$.

Definition 8.2.17 (Return and bind operators of the lifting monad)**.** *In light of lifting being a monad, given a dcpo D, I write $\eta_{\mathbb{L}} : D \to D_\perp$ for the monadic return operator, which is defined as*

$$\eta_{\mathbb{L}}(d) \coloneqq d. \tag{8.2.10}$$

This notation is overloaded because the dcpo D is not specified, although it is usually clear from the context. Additionally, I write $\leftarrow_{\mathbb{L}}$ for the monadic bind operator. Given a lift $D_\perp$, the bind operator is defined as

$$x \leftarrow_{\mathbb{L}} d; f \coloneqq \begin{cases} f[d/x] & \text{if } d \neq \perp \\ \perp & \text{otherwise.} \end{cases} \tag{8.2.11}$$

Here, x is a variable, $d \in D_\perp$ is an element in the lift, and f is a mathematical expression mentioning variable x. The notation $f[d/x]$ refers to the result of substituting d for variable x in the expression f.

8.2.3 Denotational Semantics of RPCF$_n$

This section introduces domain-theoretic denotational semantics of RPCF$_n$. It is adapted from the denotational semantics of call-by-value PCF presented in Winskel [228]. The notation and style of presentation are borrowed from Pitts et al. [190], Streicher [212].

Types In the denotational semantics of RPCF$_n$, the base types, namely `unit` and `int`, are assigned discrete dcpos.

Definition 8.2.18 (Discrete dcpo)**.** *Given a set X, its discrete dcpo is given by $(X, =)$, where the partial order is given by the equality $(=)$ of the elements in the set X.*

The denotations of compound value types (i.e., $A_1 + A_2$, $A_1 \times A_2$, $L(A)$, and $A_1 \to A_2$) are inductively constructed from the denotations of the constituent types. The denotation of the computation type $\mathsf{F}A$ extends the type A's denotation with (i) a bottom element $\perp$ to represent non-termination; (ii) the resource monoid RM to represent computational cost; and (iii) a tuple REM^n of n resource-effect monoids to track resource components.

Formally, types A of the programming language RPCF$_n$ are assigned the following dcpos:

$$\llbracket \texttt{unit} \rrbracket \coloneqq (\{\langle\rangle\}, =) \qquad \llbracket \texttt{int} \rrbracket \coloneqq (\mathbb{Z}, =) \tag{8.2.12}$$

$$\llbracket A_1 + A_2 \rrbracket \coloneqq \llbracket A_1 \rrbracket + \llbracket A_2 \rrbracket \qquad \llbracket A_1 \times A_2 \rrbracket \coloneqq \llbracket A_1 \rrbracket \times \llbracket A_2 \rrbracket \tag{8.2.13}$$

$$\llbracket L(A) \rrbracket \coloneqq \mu X. \llbracket \texttt{unit} \rrbracket + \llbracket A \rrbracket \times X \qquad \llbracket A \to B \rrbracket \coloneqq [\llbracket A \rrbracket \to \llbracket B \rrbracket]_{\text{cont}} \tag{8.2.14}$$

$$\llbracket \mathsf{F}A \rrbracket \coloneqq (\mathsf{REM}^n \times \mathsf{RM} \times \llbracket A \rrbracket)_\perp. \tag{8.2.15}$$

In Eq (8.2.13), the operator + on the right-hand side creates a dcpo of the disjoint union ([228, §8.3.5]), and the operator $\times$ creates a dcpo of the Cartesian product ([228, §8.3.2]). In Eq (8.2.14), the operator μ is the least-fixed-point operator of dcpos. More formally, solutions to recursive domain equations of the form $X \cong F(X)$ (e.g., $X \cong \llbracket \texttt{unit} \rrbracket + \llbracket A \rrbracket \times X$ for the list type $L(A)$) are

defined as initial F-algebras (see Smyth and Plotkin [211] and §5 of Abramsky and Jung [3]). The operator $[\cdot]_{\text{cont}}$ in Eq (8.2.14) creates a dcpo of continuous functions (Defn. 8.2.14).

Definition 8.2.19 (Bind operator of the combined monad of cost and resource effects)**.** *The mapping from a dcpo D to a domain $(\mathsf{REM}^n \times \mathsf{RM} \times D)_\perp$ is a monad in the category* **DCPO** *of dcpos. Its monadic bind operator is denoted by $\leftarrow_{\mathbb{T}}$. It is defined as*

$$x \leftarrow_{\mathbb{T}} d; f \coloneqq (\sigma_1, c_1, a_1) \leftarrow_{\mathbb{L}} d; (\sigma_2, c_2, a_2) \leftarrow_{\mathbb{L}} f(a_1); \eta_{\mathbb{L}}(\sigma_2 \circ \sigma_1, c_1 \oplus c_2, a_2), \tag{8.2.16}$$

where $\circ$ is the binary operator of the resource-effect monoid REM *(Defn. 8.2.2), and $\oplus$ is the binary operator of the resource monoid* RM *(Defn. 3.2.1).*

Expressions Consider a typing judgment $\Gamma \vdash e : A$ in $\textsc{Rpcf}_n$. A well-typed environment $V : \Gamma$ is interpreted as a tuple of all values $(x : v) \in V$. Hence, the denotation of the typing context Γ is given by

$$[\![\Gamma]\!] \coloneqq \{x : [\![A]\!] \mid (x : A) \in \Gamma\}. \tag{8.2.17}$$

The typing judgment $\Gamma \vdash e : A$ is interpreted as a continuous function from the dcpo $[\![\Gamma]\!]$ to the dcpo $[\![A]\!]$. The denotations of a few language constructs are listed below:

$$[\![x]\!](\gamma) \coloneqq \gamma(x) \qquad \text{where } \Gamma \vdash x : A \tag{8.2.18}$$

$$[\![\mathsf{fun}\, f\, x = e]\!](\gamma) \coloneqq \mathsf{fix}([\![e]\!](\gamma)) \qquad \text{where } \Gamma, f : A_1 \to \mathsf{F}A_2, x : A_1 \vdash e : \mathsf{F}A_2 \tag{8.2.19}$$

$$[\![\mathsf{let}\, x = e_1\, \mathsf{in}\, e_2]\!](\gamma) \coloneqq x \leftarrow_{\mathbb{T}} [\![e_1]\!](\gamma); [\![e_2]\!](\gamma)(x) \quad \text{where } (\Gamma \vdash e_1 : \mathsf{F}A_1); (\Gamma, x : A_1 \vdash e_2 : \mathsf{F}A_2), \tag{8.2.20}$$

where $\gamma \in [\![\Gamma]\!]$ is the denotation of an input environment of type Γ. In Eq (8.2.19), the operator $\mathsf{fix}(\cdot)$ computes the least fixed point of the function $[\![e]\!](\gamma) \in [\![(A_1 \to \mathsf{F}A_2) \to (A_1 \to \mathsf{F}A_2)]\!]$. In Eq (8.2.20), the operator $\leftarrow_{\mathbb{T}}$ is the monadic bind operator of the combined monad of computational cost and resource effects (Defn. 8.2.19). The denotations of language constructs in the vanilla call-by-value Pcf are presented in §11.3 (for product types and function types), §11.11 (for sum types), and §13.3 (for list types, or more generally, recursive types) of Winskel [228].

Cost and resource components The construct tick q ($q \in \mathbb{Q}$), which manipulates cost counters, has a denotation

$$[\![\mathsf{tick}\, q]\!](\gamma) \coloneqq \begin{cases} \eta_{\mathbb{L}}(\diamond^n, (q, 0), \langle\,\rangle) & \text{if } q \geq 0 \\ \eta_{\mathbb{L}}(\diamond^n, (0, -q), \langle\,\rangle) & \text{otherwise,} \end{cases} \tag{8.2.21}$$

where $\diamond \in \mathsf{REM}$ is the identity resource effect (Defn. 8.2.2), and $\diamond^n \in \mathsf{REM}^n$ is an n-tuple of the identity resource effect $\diamond$. The monadic return operator $\eta_{\mathbb{L}}$ (Eq (8.2.10)) of the lifting monad takes in an element of the dcpo $(\mathsf{REM}^n \times \mathsf{RM} \times [\![A]\!])$ and returns the corresponding element in the domain $(\mathsf{REM}^n \times \mathsf{RM} \times [\![A]\!])_\perp$. For those constructs that manipulate resource components, their denotations are

$$[\![\mathsf{mark}_i]\!](\gamma) \coloneqq \eta_{\mathbb{L}}(\diamond^n[i \mapsto \wr], (0, 0), \langle\,\rangle) \tag{8.2.22}$$

$$[\![\mathsf{unmark}_i]\!](\gamma) \coloneqq \eta_{\mathbb{L}}(\diamond^n[i \mapsto \wr], (0, 0), \langle\,\rangle) \tag{8.2.23}$$

$$[\![\mathsf{reset}_i]\!](\gamma) \coloneqq \eta_{\mathbb{L}}(\diamond^n[i \mapsto \circlearrowleft], (0, 0), \langle\,\rangle). \tag{8.2.24}$$

$(\!|\cdot|\!)_n : \mathsf{Ty} \to \mathsf{Ty}$ Translation of types

$$
\begin{aligned}
(\!|\mathtt{unit}|\!)_n &\coloneqq \mathtt{unit} & (\!|\mathtt{int}|\!)_n &\coloneqq \mathtt{int} \\
(\!|A_1 + A_2|\!)_n &\coloneqq (\!|A_1|\!)_n + (\!|A_2|\!)_n & (\!|A_1 \times A_2|\!)_n &\coloneqq (\!|A_1|\!)_n \times (\!|A_2|\!)_n \\
(\!|L(A)|\!)_n &\coloneqq L((\!|A|\!)_n) & (\!|A_1 \to A_2|\!)_n &\coloneqq (\!|A_1|\!)_n \to (\!|A_2|\!)_n \\
(\!|\mathsf{F}A|\!)_n &\coloneqq (\mathtt{nat}^2)^n \to \mathsf{F}((\mathtt{nat}^2)^n \times (\!|A|\!)_n)
\end{aligned}
$$

Lst. 8.3: Transformation of types in RPCF_n.

The notation $\diamond^n[i \mapsto \wr]$ means the i^{th} component of the tuple $\diamond^n$ is replaced with the resource effect $\wr$. The resource effects $\wr$, $\wr$, and $\circlearrowleft$ are defined in Defn. 8.2.3. Given an expression $e : \mathsf{F}A$, if $[\![e]\!] = \eta_{\mathbb{L}}(\sigma, c, a)$, I say that e evaluates to a with a tuple σ of resource effects and cost c.

8.3 Program Transformation

This section formalizes the transformation of resource-decomposed programs written in the language RPCF_n ($n \in \mathbb{N}$) to resource-guarded ones in RPCF_0. Given a resource-decomposed program $P_{\text{rd}}(x)$ in RPCF_n, which is annotated to define user-specified resource components (i.e., mark, unmark, and reset), the transformation eliminates the resource-effect annotations, turning them into resource guards $\mathbf{r} = (r_1, \ldots, r_n)$, which are numeric non-negative input variables. As shown in Fig. 2.2, the resulting resource-guarded program $P_{\text{rg}}(x, \mathbf{r})$ is then analyzed by one of the constituent resource-analysis techniques, yielding an overall cost bound $f(x, r_1, \ldots, r_n)$ parametric in both the original input x and resource guards r_i ($i = 1, \ldots, n$).

The program transformation, denoted by $(\!|\cdot|\!)_n$, transforms both types A (§8.3.1) and expressions e (§8.3.2).

8.3.1 Types

Listing 8.3 defines the transformation $(\!|\cdot|\!)_n : \mathsf{Ty} \to \mathsf{Ty}$, where Ty is the set of types in the language RPCF_n. Value types are transformed inductively. The only non-trivial transformation is for the computation type $\mathsf{F}A$, which captures computational effects of non-termination, computational cost, and resource components.

The computation type $\mathsf{F}A$ is translated to a function type

$$(\!|\mathsf{F}A|\!)_n \coloneqq (\mathtt{nat}^2)^n \to \mathsf{F}((\mathtt{nat}^2)^n \times (\!|A|\!)_n), \tag{8.3.1}$$

where $\mathtt{nat}$ is the type of natural numbers, and $\mathtt{nat}^2$ is an abbreviation of the product type $\mathtt{nat} \times \mathtt{nat}$. Although the language RPCF_n does not have a built-in type for natural numbers, the type $\mathtt{nat}$ can be encoded using the list type $L(\mathtt{unit})$ (i.e., unary encoding of natural numbers). This encoding is adopted in the instantiations of resource decomposition that use Conventional AARA (§8.5 and §8.6).

$(\!|\cdot|\!)_n : \mathsf{Tm}(\Gamma, A) \to \mathsf{Tm}((\!|\Gamma|\!)_n, (\!|A|\!)_n)$ Translation of expressions

$$
\begin{aligned}
(\!|\langle\,\rangle|\!)_n &:= \langle\,\rangle & (\!|z|\!)_n &:= z \\
(\!|x|\!)_n &:= x & (\!|\mathsf{left} \cdot e|\!)_n &:= \mathsf{left} \cdot (\!|e|\!)_n \\
(\!|\mathsf{right} \cdot e|\!)_n &:= \mathsf{right} \cdot (\!|e|\!)_n & (\!|\langle e_1, e_2\rangle|\!)_n &:= \langle (\!|e_1|\!)_n, (\!|e_2|\!)_n\rangle \\
(\!|[\,]|\!)_n &:= [\,] & (\!|e_1 :: e_2|\!)_n &:= (\!|e_1|\!)_n :: (\!|e_2|\!)_n \\
(\!|\mathsf{fun}\ f\ x = e|\!)_n &:= \mathsf{fun}\ f\ x = (\!|e|\!)_n & (\!|\mathsf{return}\ e|\!)_n &:= \mathsf{fun}\ \_\ s = \mathsf{return}\ \langle s, (\!|e|\!)_n\rangle
\end{aligned}
$$

Lst. 8.4: Transformation of constructors in $\textsc{Rpcf}_n$.

$(\!|\cdot|\!)_n : \mathsf{Tm}(\Gamma, A) \to \mathsf{Tm}((\!|\Gamma|\!)_n, (\!|A|\!)_n)$ Translation of expressions

$$
\begin{aligned}
(\!|\mathsf{case}\ e\ \{\mathsf{left} \cdot x_1 \hookrightarrow e_1 \mid \mathsf{right} \cdot x_2 \hookrightarrow e_2\}|\!)_n &:= \mathsf{case}\ (\!|e|\!)_n\ \{\mathsf{left} \cdot x_1 \hookrightarrow (\!|e_1|\!)_n \mid \mathsf{right} \cdot x_2 \hookrightarrow (\!|e_2|\!)_n\} \\
(\!|\mathsf{case}\ e\ \{\langle x_1, x_2\rangle \hookrightarrow e_1\}|\!)_n &:= \mathsf{case}\ (\!|e|\!)_n\ \{\langle x_1, x_2\rangle \hookrightarrow (\!|e_1|\!)_n\} \\
(\!|\mathsf{case}\ e\ \{[\,] \hookrightarrow e_1 \mid (x_1 :: x_2) \hookrightarrow e_2\}|\!)_n &:= \mathsf{case}\ (\!|e|\!)_n\ \{[\,] \hookrightarrow (\!|e_1|\!)_n \mid (x_1 :: x_2) \hookrightarrow (\!|e_2|\!)_n\} \\
(\!|e_1\ e_2|\!)_n &:= (\!|e_1|\!)_n\ (\!|e_2|\!)_n \\
(\!|\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2|\!)_n &:= \mathsf{fun}\ \_\ s = (\mathsf{let}\ \langle s_1, x\rangle = (\!|e_1|\!)_n\ s\ \mathsf{in}\ (\!|e_2|\!)_n\ s_1)
\end{aligned}
$$

Lst. 8.5: Transformation of destructors in $\textsc{Rpcf}_n$.

A function of type (8.3.1) takes as input an n-tuple of resource guards, one for each of the n resource components. Each resource guard is a pair $\langle h, r\rangle : \mathsf{nat}^2$, where h is the initial value of the resource guard and r is the remaining resources to pay for the construct mark. The resource guard is initialized to $\langle h, h\rangle : \mathsf{nat}^2$, where h is the high-water-mark measurement of the resource component. It is necessary to retain the initial value h of the resource guard since the translation of the construct reset_i resets the resource guard to h.

8.3.2 Expressions

Listings 8.4–8.6 define the transformation $(\!|\cdot|\!)_n : \mathsf{Tm}(\Gamma, A) \to \mathsf{Tm}((\!|\Gamma|\!)_n, (\!|A|\!)_n)$ of expressions in the language $\textsc{Rpcf}_n$. Here, $\mathsf{Tm}(\Gamma, A)$ denotes the set of expressions e such that $\Gamma \vdash e : A$, and $(\!|\Gamma|\!)_n$ is the pointwise extension of $(\!|\cdot|\!)_n : \mathsf{Ty} \to \mathsf{Ty}$ to a typing context Γ. Throughout the transformation's rules, given a tuple $s = (s_1, \ldots, s_n)$, the notation $s(i)$ refers to the i^{th} component p_i of the tuple. Also, $s[i \mapsto v]$ denotes a modified tuple $(s_1, \ldots, s_{i-1}, v, s_{i+1}, \ldots, s_n)$, where the i^{th} component of the original tuple s is replaced with v. The underscore (e.g., $\mathsf{fun}\ \_\ s = e$) means it does not matter what symbol goes in there.

Listing 8.4 and Listing 8.5 list the transformation rules for constructors and destructors, respectively, and they cover all the standard constructs inherited from the vanilla Pcf. In many of them, the program transformation proceeds inductively.

Listing 8.6 lists the transformation rules for effectful expressions unique to $\textsc{Rpcf}_n$. For the

$(\!|\cdot|\!)_n : \mathsf{Tm}(\Gamma, A) \to \mathsf{Tm}((\!|\Gamma|\!)_n, (\!|A|\!)_n)$ Translation of expressions

$$
\begin{aligned}
(\!|\mathsf{tick}\ q|\!)_n &:= \mathsf{fun}\ \_\ s = (\mathsf{let}\ x = \mathsf{tick}\ q\ \mathsf{in}\ \mathsf{return}\ \langle s, x\rangle)\\
(\!|\mathsf{mark}_i|\!)_n &:= \mathsf{fun}\ \_\ s = (\mathsf{let}\ \langle h, r\rangle = s(i)\ \mathsf{in}\ \mathsf{if}\ r = 0\ \mathsf{then}\ \mathsf{excp}\ \mathsf{else}\ \mathsf{return}\ \langle s[i \mapsto (h, r-1)], \langle\,\rangle\rangle)\\
(\!|\mathsf{unmark}_i|\!)_n &:= \mathsf{fun}\ \_\ s = \mathsf{let}\ \langle h, r\rangle = s(i)\ \mathsf{in}\ \mathsf{return}\ \langle s[i \mapsto (h, r+1)], \langle\,\rangle\rangle\\
(\!|\mathsf{reset}_i|\!)_n &:= \mathsf{fun}\ \_\ s = \mathsf{let}\ \langle h, \_\rangle = s(i)\ \mathsf{in}\ \mathsf{return}\ \langle s[i \mapsto (h, h)], \langle\,\rangle\rangle
\end{aligned}
$$

Lst. 8.6: Transformation of effectful expressions in $\textsc{Rpcf}_n$.

expression $\mathsf{mark}_i : \mathtt{Funit}$, it is transformed to

$$
\begin{aligned}
(\!|\mathsf{mark}_i|\!)_n := \mathsf{fun}\ \_\ s = (&\mathsf{let}\ \langle h, r\rangle = s(i)\ \mathsf{in}\\
&\mathsf{if}\ r = 0\ \mathsf{then}\ \mathsf{excp}\ \mathsf{else}\ \mathsf{return}\ \langle s[i \mapsto (h, r-1)], \langle\,\rangle\rangle),
\end{aligned}
\tag{8.3.2}
$$

which indeed has type $(\!|\mathtt{Funit}|\!)_n = (\mathsf{nat}^2)^n \to \mathsf{F}((\mathsf{nat}^2)^n \times \mathsf{unit})$. The function (8.3.2) takes as input p, which is an n-tuple of resource guards. The function first examines the i^{th} resource guard $s(i) = \langle h, r\rangle$, where h is the initial value and r is the current value of the resource guard. If we have $r = 0$, then the program raises an exception (denoted by excp), suggesting that the initial value h of the i^{th} resource guard is not high enough. Otherwise, if $r > 0$, the program decrements the current value by one and returns the unit element $\langle\,\rangle$. In §8.4, I show that exceptions never occur when the resource guards are initialized to sufficiently high values, particularly the high-water-mark measurements of the resource components.

Remark 8.3.1 (Exceptions)**.** *The purpose of the exception* $\mathsf{excp} : \mathtt{Funit}$ *raised in* $(\!|\mathsf{mark}_i|\!)_n$ *is to indicate that the initial value of the* i^{th} *resource guard is not sufficiently high (i.e., it is an unsound bound for the* i^{th} *resource component). Thus, exceptions are meant to indicate errors and terminate the program immediately, rather than to change the control flow via exception handling (which is another use case of exceptions [105, §29]).*

In fact, as long as excp *is well-typed, it does not matter what its implementation is. The soundness theorem of resource decomposition (Thm. 8.4.1) requires the initial values of resource guards to be sufficiently high (i.e., they are equal to or higher than high-water-mark measurements of the corresponding resource components). Under this requirement, a resource-guarded expression* $(\!|e|\!)_n$ *never raises an exception. Hence,* $\mathsf{excp} : \mathtt{Funit}$ *is allowed to be any expression of type* $\mathtt{Funit}$*. For example,* $\mathsf{excp} : \mathtt{Funit}$ *can be implemented as*

$$
\mathsf{excp} := \mathsf{return}\ \langle\,\rangle. \tag{8.3.3}
$$

This implementation does not tell us whether an input resource guard is sufficiently high when a resource-guarded expression $(\!|e|\!)_n$ *is executed. Nonetheless, it does not break Thm. 8.4.1.* «

Dual to the construct mark_i's translation is the construct unmark_i's translation: it increments the current value r of the i^{th} resource guard. The translation of reset_i takes in a tuple p of resource guards. If the i^{th} resource guard is $s(i) = \langle h, r\rangle$, the program resets the current value r to the corresponding initial value h.

Resource guards in the target program evolve in the opposite direction to resource components in the source program. For example, the construct mark_i increments a resource component r_i's counter by one, while its translation decrements the current value of the i^{th} resource guard by one. Also, the construct reset_i resets the counter of the resource component r_i to zero. Meanwhile, its translation resets the corresponding resource guard to its initial value h, which will be initialized to a high-water-mark measurement of the resource component.

The expression tick q ($q \in \mathbb{Q}$) is left unmodified since it represents the distinguished resource metric that the resource-guarded program should preserve.

8.4 Soundness

This section formulates the soundness of the program transformation $(\!|\cdot|\!)_n$ and proves it by a binary-logical-relation argument between the source and target programs. First, §8.4.1 formulates the soundness of the program transformation. Next, §8.4.2 defines the logical relation used in the soundness proof. Finally, §8.4.3 proves several key inductive cases in the proof.

8.4.1 Theorem Statement

Soundness of the program transformation $(\!|\cdot|\!)_n$ of a resource-decomposed program to a resource-guarded program means:

1. The transformation preserves the cost of the source program; and
2. The target program runs successfully (i.e., it does not raise an exception), provided that the resource guards are initialized with the high-water-mark measurements of the resource components.

Thm. 8.4.1 formally states the soundness of the program transformation.

Theorem 8.4.1 (Soundness of the program transformation)**.** *Let e : $\mathtt{Funit}$ be a closed expression of $\textsc{Rpcf}_n$. Suppose the expression e terminates with a tuple $\sigma \in \mathsf{REM}^n$ of resource effect and cost $c \in \mathsf{RM}$, where*

$$((h_1, r_1,), \ldots, (h_n, r_n)) \coloneqq \sigma((0,0), \ldots, (0,0)). \tag{8.4.1}$$

Let $k_i \in \mathbb{N}$ $(i = 1, \ldots, n)$ be arbitrary constants. Then

$$[\![(\!|e|\!)_n]\!]((h_1 + k_1, h_1 + k_1), \ldots, (h_n + k_n, h_n + k_n)) \tag{8.4.2}$$

terminates with the cost c.

In Thm. 8.4.1, $h_1, \ldots, h_n$ are the high-water-mark measurements of resource components obtained by running the source program e : $\mathtt{Funit}$. If the resource guards in the target program $(\!|e|\!)_n$ are initialized with these high-water-mark measurements (or higher), the target program runs successfully. Otherwise, if the resource guards' initial values are too low, during the execution of the resource-guarded program, the translation of the construct mark_i (Listing 8.6) raises an exception.

The soundness theorem makes sense because resource guards in the target program evolve in the opposite direction to the corresponding resource components in the source program. Whenever resource components are incremented by the construct mark_i (resp., decremented

by the construct unmark_i) in a resource-decomposed program, the resource guards in the target program are decremented (resp., incremented). Consequently, as long as the resource guards are initialized with the high-water marks $h_1, \ldots, h_n$ of the resource components $r_1, \ldots, r_n$, the resource guards never become negative while executing the resource-guarded program.

Thm. 8.4.1 suggests the soundness of the resource-decomposition framework: composing sound resource analyses via resource decomposition yields a sound cost bound. In particular, a sound cost bound of the original program $e : \texttt{Funit}$ can be obtained by composing the following bounds by substitution: (i) sound (but not necessarily tight) bounds g_i of high-water marks h_i of resource components r_i $(i = 1, \ldots, n)$; and (ii) a sound bound $f(r_1, \ldots, r_n)$ of the resource-guarded program $(\!|e|\!)_n$. Since the sound bounds f_i $(i = 1, \ldots, n)$ of resource components are not necessarily tight, when we plug them into the sound overall cost bound $f(r_1, \ldots, r_n)$, we must ensure that $(\!|e|\!)_n$ runs successfully even if the resource guards are strictly higher than the high-water marks h_i $(i = 1, \ldots, n)$. This is why Thm. 8.4.1 allows the resource guards to be initialized with $((h_i + k_i, h_i + k_i))_{i=1}^n$ for arbitrary constants $k_i \in \mathbb{N}$ $(i = 1, \ldots, n)$, rather than just the high-water marks $((h_i, h_i))_{i=1}^n$.

8.4.2 Logical Relation

As $\textsc{Rpcf}_n$ is a higher-order language, a logical relation is employed to prove Thm. 8.4.1. The logical relation is a commonly employed proof technique for proving properties (e.g., termination, program equality, and parametricity) of a programming language.

This section introduces a logical relation, dubbed the approximation relation. It is carefully defined such that its fundamental theorem (Thm. 8.4.2) implies the soundness theorem Thm. 8.4.1 of the program transformation $(\!|\cdot|\!)_n$.

Terminology and notation To set the stage for a logical-relation-based proof of the soundness, I introduce terminology and notation.

Definition 8.4.1 (Resource state)**.** *In the context of resource components and resource guards, an element $s \in (\mathbb{N}^2)^n$ is called a resource state.*

Definition 8.4.2 (Offset)**.** *An offset is a resource state $k = (k_1, \ldots, k_n) \in (\mathbb{N}^2)^n$, where each pair $k_i \in \mathbb{N}^2$ has equal components (i.e., $k_i.h = k_i.r$) for $i = 1, \ldots, n$. The set of offsets is denoted by $(\mathbb{N}^2)^n_{\text{offset}}$.*

Notation (Offset of high-water marks)**.** *Given a resource state $s = ((h_1, r_1), \ldots, (h_n, r_n))$, the notation $\overline{s.h}$ refers to the offset*

$$\overline{s.h} \coloneqq ((h_1, h_1), \ldots, (h_n, h_n)). \tag{8.4.3}$$

Definition 8.4.3 (Initial resource-guard values)**.** *Consider two resource states $s_1, s_2 \in (\mathbb{N}^2)^n$, where*

$$s_1 = (s_{1,1}, \ldots, s_{1,n}) \qquad s_2 = (s_{2,1}, \ldots, s_{2,n}). \tag{8.4.4}$$

A resource state $\mathsf{initrg}(s_1, s_2) \in (\mathbb{N}^2)^n$, which stands for the initial values of resource guards, is defined as

$$\mathsf{initrg}(s_1, s_2) \coloneqq ((s_{2,i}.h, s_{2,i}.h - (s_{1,i}.h - s_{1,i}.r)))_{i=1}^n. \tag{8.4.5}$$

The resource state $\mathsf{initrg}(s_1, s_2)$ in Defn. 8.4.3 denotes the values to which resource guards should be initialized such that a resource-guarded program runs successfully. To justify Defn. 8.4.3, given a resource-decomposed expression $e : \mathsf{F}A$, assume

$$[\![e]\!] = \eta_{\mathbb{L}}(\sigma, \cdot, \cdot), \tag{8.4.6}$$

where $\sigma \in \mathsf{REM}^n$ is a tuple of resource effects. Viewing the tuple σ as a function, let s_1 and s_2 be the initial and final resource states, respectively:

$$s_1 = ((h_{1,i}, r_{1,i}))_{i=1}^n \qquad s_2 \coloneqq \sigma(s_1) = ((h_{2,i}, r_{2,i}))_{i=1}^n. \tag{8.4.7}$$

The quantities $h_{1,i}$ and $h_{1,i} - r_{1,i}$ $(i = 1, \ldots, n)$ are the high-water-mark cost and net cost, respectively, of any computation (say, e_{prcd}) that precedes the expression e's execution. On the other hand, the quantities $h_{2,i}$ and $h_{2,i} - r_{2,i}$ $(i = 1, \ldots, n)$ are the *combined* high-water-mark cost and net cost, respectively, of both (i) the computation e_{prcd} preceding the expression e; and (ii) the computation of e itself. This suggests that, before executing the sequential composition of the translated expressions $(\!|e_{\mathrm{prcd}}|\!)_n$ and $(\!|e|\!)_n$, we should initialize resource guards to the high-water marks $h_{2,i}$ $(i = 1, \ldots, n)$. As long as the resource guards are initialized this way, they never become negative throughout the execution of $(\!|e_{\mathrm{prcd}}|\!)_n$ followed by $(\!|e|\!)_n$.

Furthermore, if the resource guards are initialized to $h_{2,i}$ $(i = 1, \ldots, n)$, by the end of the execution of $(\!|e_{\mathrm{prcd}}|\!)_n$ (i.e., right before running e), the resource guards have a resource state

$$\mathsf{initrg}(s_1, \sigma(s_1)) = ((h_{2,i}, h_{2,i} - (h_{1,i} - r_{1,i})))_{i=1}^n, \tag{8.4.8}$$

where the net costs $h_{1,i} - r_{1,i}$ of running the computation e_{prcd} are subtracted from the initial values $h_{2,i}$ of the resource guards $(i = 1, \ldots, n)$. Defn. 8.4.3 has been introduced to refer easily to the resource state in Eq (8.4.8). After executing the expression e, the final resource state of the resource guards is

$$s_2 = \sigma(s_1) = ((h_{2,i}, r_{2,i}))_{i=1}^n = ((h_{2,i}, h_{2,i} - (h_{2,i} - r_{2,i})))_{i=1}^n. \tag{8.4.9}$$

Definition 8.4.4 (Running a resource-guarded expression)**.** *Given a type A, consider a denotation $d \in [\![(\!|\mathsf{F}A|\!)_n]\!]$. Given a tuple $\sigma \in \mathsf{REM}^n$ of resource effects, computational cost $c \in \mathsf{RM}$, and a denotation $a \in [\![A]\!]$, the predicate $\mathsf{runrg}(d, \sigma, c, a)$ is defined as*

$$\begin{aligned} \mathsf{runrg}(d, \sigma, c, a) \iff \forall [s \in (\mathbb{N}^2)^n, k \in (\mathbb{N}^2)^n_{\mathrm{offset}}]. \\ d\,(\mathsf{initrg}(s, \sigma(s)) + k) = \eta_{\mathbb{L}}(c, \langle \sigma(s) + k, a \rangle). \end{aligned} \tag{8.4.10}$$

In the predicate, the resource guards of the denotation $d \in [\![(\!|\mathsf{F}A|\!)_n]\!]$ are initialized to $\mathsf{initrg}(s, \sigma(s)) + k$ for an arbitrary resource state $s \in (\mathbb{N}^2)^n$ and an arbitrary offset $k : (\mathbb{N}^2)^n_{\mathrm{offset}}$. The predicate states that the denotation d evaluates with cost $c \in \mathsf{RM}$ and returns the pair $\langle \sigma(s), a \rangle$, where the first component $\sigma(s)$ is the final resource state of the resource guards. The predicate $\mathsf{runrg}(\cdot)$ stands for running a resource-guarded expression.

$$
\begin{aligned}
\langle\,\rangle \lhd_{\mathtt{unit}} \langle\,\rangle &\iff \top \\
z \lhd_{\mathtt{int}} z &\iff \top \\
\mathsf{left} \cdot a_1 \lhd_{A_1+A_2} \mathsf{left} \cdot a_2 &\iff a_1 \lhd_{A_1} a_2 \\
\mathsf{right} \cdot a_1 \lhd_{A_1+A_2} \mathsf{right} \cdot a_2 &\iff a_1 \lhd_{A_2} a_2 \\
\langle a_1, a_2\rangle \lhd_{A_1\times A_2} \langle a_3, a_4\rangle &\iff (a_1 \lhd_{A_1} a_3) \wedge (a_2 \lhd_{A_2} a_4) \\
[\,] \lhd_{L(A)} [\,] &\iff \top \\
(a_1 :: a_2) \lhd_{L(A)} (a_3 :: a_4) &\iff (a_1 \lhd_A a_3) \wedge (a_2 \lhd_{L(A)} a_4) \\
a_1 \lhd_{A_1\to A_2} a_2 &\iff \forall[a_3 \lhd_{A_1} a_4].(a_1\ a_3) \lhd_{A_2} (a_2\ a_4) \\
a_1 \lhd_{\mathsf{F}A} a_2 &\iff \forall[\sigma \in \mathsf{REM}^n, c \in \mathsf{RM}, b_1 \in [\![A]\!]]. \\
&\qquad a_1 = \eta_{\mathbb{L}}(\sigma, c, b_1) \to (\exists[b_2 \in [\![(\!|A|\!)_n]\!]].b_1 \lhd_A b_2 \wedge \mathsf{runrg}(a_2, \sigma, c, b_2))
\end{aligned}
$$

Lst. 8.7: Approximation relation $\lhd_A \subseteq [\![A]\!] \times [\![(\!|A|\!)_n]\!]$.

Approximation relation I define the following logical relation, dubbed *approximation relation*:

$$\lhd_A \subseteq [\![A]\!] \times [\![(\!|A|\!)_n]\!]. \tag{8.4.11}$$

It is a binary relation between source and target denotations of the transformation $(\!|\cdot|\!)_n$ and is indexed by type A. The approximation relation (8.4.11) is defined such that the relation at $A = \mathsf{Funit}$ implies Thm. 8.4.1.

Listing 8.7 defines the approximation relation $\lhd_A$. Its definition is straightforwardly inductive for all value types (e.g., sum, product, list, and arrow types).

The only non-trivial case is $a_1 \lhd_{\mathsf{F}A} a_2$, which is defined as

$$
\begin{aligned}
&\forall[\sigma \in \mathsf{REM}^n, c \in \mathsf{RM}, b_1 \in [\![A]\!]]. \\
&\qquad a_1 = \eta_{\mathbb{L}}(\sigma, c, b_1) \to (\exists[b_2 \in [\![(\!|A|\!)_n]\!]].b_1 \lhd_A b_2 \wedge \mathsf{runrg}(a_2, \sigma, c, b_2)).
\end{aligned} \tag{8.4.12}
$$

It states that, if the resource-decomposed computation $a_1 \in [\![\mathsf{F}A]\!]$ has the form $\eta_{\mathbb{L}}(\sigma, c, b_1)$, then the resource-guarded computation $a_2 \in [\![(\!|\mathsf{F}A|\!)_n]\!]$ satisfies $\mathsf{runrg}(a_2, \sigma, c, b_2)$ for some $b_2 \in [\![(\!|A|\!)_n]\!]$ such that $b_1 \lhd_A b_2$. Here, $b_1 \in [\![A]\!]$ is the value (i.e., computational output) stored inside the computation a_1. It additionally stores a tuple $\sigma \in \mathsf{REM}^n$ of resource effect and computational cost $c \in \mathsf{RM}$. Likewise, $b_2 \in [\![(\!|A|\!)_n]\!]$ is the computational output stored inside the resource-guarded computation a_2. If we have $a_1 = \bot$ instead of $a_1 = \eta_{\mathbb{L}}(\cdot,\cdot,\cdot)$, then the approximation relation $a_1 \lhd_{\mathsf{F}A} a_2$ vacuously holds (Lem. 8.4.4).

Universal quantification over offsets The predicate $\mathsf{runrg}(a_2, \sigma, c, b_2)$ (Defn. 8.4.4) states that, for any resource state $s \in (\mathbb{N}^2)^n$ and offset $k \in (\mathbb{N}^2)^n_{\mathrm{offset}}$, if the resource-guarded computation a_2 is fed with the initial resource guards of $\mathsf{initrg}(s, \sigma(s)) + k$, then the computation a_2 successfully runs. It has the same computational cost c as the resource-decomposed computation a_1, and returns a pair $\langle \sigma(s) + k, b_2\rangle$.

It is crucial that the predicate $\mathsf{runrg}(\cdot,\cdot,\cdot,\cdot)$ universally quantifies over offsets $k \in (\mathbb{N}^2)^n_{\text{offset}}$. Without this universal quantification, an inductive proof of the fundamental theorem Thm. 8.4.2 of the logical relation $\lhd_A$ does not go through: it fails in the inductive case of a let-binding. The universal quantification over offsets comes into play in Lemmas 8.4.1 and 8.4.2, which are used to prove Prop. 8.4.2 for a let-binding.

To illustrate the need for the universal quantification over offsets, consider a let-binding:

$$\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \qquad \text{where } e_1 : \mathsf{F}A \text{ and } x : A \vdash e_2 : \mathsf{F}B. \tag{8.4.13}$$

In an inductive proof of the fundamental theorem, the inductive hypotheses are

$$[\![e_1]\!] \lhd_A (\!|e_1|\!)_n \qquad [\![e_2]\!] \lhd_{A\to \mathsf{F}B} [\![(\!|e_2|\!)_n]\!], \tag{8.4.14}$$

which in turn yields

$$[\![e_2]\!]\,[\![e_1]\!] \lhd_{\mathsf{F}B} [\![(\!|e_2|\!)_n]\!]\,[\![(\!|e_1|\!)_n]\!]. \tag{8.4.15}$$

To derive the proof goal $[\![\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2]\!] \lhd_{\mathsf{F}B} [\![(\!|\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2|\!)_n]\!]$, the resource guards must be initialized to the *combined* high-water-mark measurements of the resource components of computations e_1 and e_2. To this end, in Eq (8.4.15), the predicate $\mathsf{runrg}(\cdot,\cdot,\cdot,\cdot)$ needs to be able to initialize the resource guards to the high-water marks of $\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$. This necessitates the universal quantification over offsets $k \in (\mathbb{N}^2)^n_{\text{offset}}$ in the predicate $\mathsf{runrg}(\cdot,\cdot,\cdot,\cdot)$.

Fundamental theorem of the logical relation The fundamental theorem of a logical relation states that the logical relation, which is a type-indexed relation, holds for all types. In a logical-relation-based proof, the logical relation is defined in such a way that a target theorem follows from the fundamental theorem of the logical relation.

Definition 8.4.5 (Approximation relation on environments)**.** *The approximation relation $\lhd_A$ lifts to environments (i.e., mapping from variables to denotations). Given environments $\gamma_1 \in [\![\Gamma]\!]$ and $\gamma_2 \in [\![(\!|\Gamma|\!)_n]\!]$, I define*

$$\gamma_1 \lhd_\Gamma \gamma_2 \iff \forall[(x : A) \in \Gamma].\gamma_1(x) \lhd_A \gamma_2(x). \tag{8.4.16}$$

Definition 8.4.6 (Approximation relation on typing judgments)**.** *The approximation relation $\lhd_A$ lifts to typing judgments. Given a well-typed expression $\Gamma \vdash e : A$, I define*

$$\Gamma \vdash [\![e]\!] \lhd_A [\![(\!|e|\!)_n]\!] \iff \forall[\gamma_1 \lhd_\Gamma \gamma_2].[\![e]\!](\gamma_1) \lhd_A [\![(\!|e|\!)_n]\!](\gamma_2). \tag{8.4.17}$$

Theorem 8.4.2 (Fundamental theorem of the logical relation)**.** *For all $\Gamma \vdash e : A$, we have*

$$\Gamma \vdash [\![e]\!] \lhd_A [\![(\!|e|\!)_n]\!]. \tag{8.4.18}$$

The soundness of resource decomposition (Thm. 8.4.1) follows from the instantiation of Thm. 8.4.2 at type `Funit`. Given a closed resource-decomposed expression e : `Funit`, Thm. 8.4.2 yields

$$[\![e]\!] \lhd_{\mathtt{Funit}} [\![(\!|e|\!)_n]\!]. \tag{8.4.19}$$

Assume that the computation e runs successfully:

$$[\![e]\!] = \eta_{\mathbb{L}}(\sigma, c, \langle\,\rangle) \qquad \text{for some } \sigma \in \mathsf{REM}^n, c \in \mathsf{RM}. \tag{8.4.20}$$

Unfolding Defn. 8.4.4 of the predicate $\mathrm{runrg}(\cdot,\cdot,\cdot,\cdot)$ gives

$$\forall[s \in (\mathbb{N}^2)^n, k \in (\mathbb{N}^2)^n_{\mathrm{offset}}].[\![(\!|e|\!)_n]\!]\,(\mathrm{initrg}(s, \sigma(s)) + k) = \eta_{\mathbb{L}}(c, \langle \sigma(s) + k, \langle\,\rangle\rangle). \tag{8.4.21}$$

Fixing $s \coloneqq \diamond^n = ((0,0), \ldots, (0,0))$, we obtain

$$\begin{aligned}
[\![(\!|e|\!)_n]\!]\,(\sigma(\diamond^n) + k) &= [\![(\!|e|\!)_n]\!]\,(\mathrm{initrg}(\diamond^n, \sigma(\diamond^n)) + k) && \text{by Defn. 8.4.3} && (8.4.22)\\
&= \eta_{\mathbb{L}}(c, \langle \sigma(\diamond^n) + k, \langle\,\rangle\rangle) && \text{by Eq (8.4.21).} && (8.4.23)
\end{aligned}$$

The last line establishes Thm. 8.4.1.

8.4.3 Soundness Proof

The proof of Thm. 8.4.2 proceeds by structural induction on expressions e. This section highlights three illustrative cases in the inductive proof:

1. Prop. 8.4.1 concerns mark_i;
2. Prop. 8.4.2 concerns $\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$;
3. Prop. 8.4.3 concerns fixed-point expressions for recursive functions.

Proposition 8.4.1 (Approximation relation holds for mark_i)**.** *We have* $[\![\mathsf{mark}_i]\!] \lhd_{\mathtt{Funit}} [\![(\!|\mathsf{mark}_i|\!)_n]\!]$.

Proof. The translation $(\!|\mathsf{mark}_i|\!)_n$ is defined as (Listing 8.6):

$$\mathsf{fun}\ \_\ s = (\mathsf{let}\ \langle h, r\rangle = s(i)\ \mathsf{in}\ \mathsf{if}\ r = 0\ \mathsf{then}\ \mathsf{excp}\ \mathsf{else}\ \mathsf{return}\ \langle s[i \mapsto (h, r-1)], \langle\,\rangle\rangle). \tag{8.4.24}$$

The denotation $[\![\mathsf{mark}_i]\!]$ is defined in Eq (8.2.22):

$$\eta_{\mathbb{L}}(\sigma, (0,0), \langle\,\rangle) \quad \text{where } \sigma \coloneqq \diamond^n[i \mapsto \wr]. \tag{8.4.25}$$

To prove the claim that $[\![\mathsf{mark}_i]\!] \lhd_{\mathtt{Funit}} [\![(\!|\mathsf{mark}_i|\!)_n]\!]$, according to the definition of the approximation relation $\lhd_{\mathtt{Funit}}$ (Listing 8.7), it suffices to establish

$$\mathrm{runrg}([\![(\!|\mathsf{mark}_i|\!)_n]\!], \sigma, (0,0), \langle\,\rangle). \tag{8.4.26}$$

According to Defn. 8.4.4 of the predicate $\mathrm{runrg}(\cdot,\cdot,\cdot,\cdot)$, the proof goal expands to

$$\forall[s \in (\mathbb{N}^2)^n, k \in (\mathbb{N}^2)^n_{\mathrm{offset}}].[\![(\!|\mathsf{mark}_i|\!)_n]\!]\,(\mathrm{initrg}(s, \sigma(s)) + k) = \eta_{\mathbb{L}}((0,0), \langle \sigma(s) + k, \langle\,\rangle\rangle). \tag{8.4.27}$$

To this end, fix an arbitrary resource state $s \in (\mathbb{N}^2)^n$ and an arbitrary offset $k \in (\mathbb{N}^2)^n_{\mathrm{offset}}$:

$$s = ((h_i, r_i))^n_{i=1} \qquad k = ((k_i, k_i))^n_{i=1}. \tag{8.4.28}$$

I conduct case analysis on the value of r_i: (i) $r_i \geq 1$ and (ii) $r_i = 0$.

Consider the first case where $r_i \geq 1$. From Defn. 8.2.3 of the resource effect $\wr \in \mathsf{REM}$ appearing in the definition of the tuple $\sigma \in \mathsf{REM}^n$ (Eq (8.4.25)), it follows that

$$\sigma(s) = s[i \mapsto (h_i, r_i) \oplus (1, 0)] = s[i \mapsto (h_i, r_i - 1)]. \tag{8.4.29}$$

From Defn. 8.4.3 of the resource state $\mathsf{initrg}(\cdot, \cdot)$, we obtain

$$\mathsf{initrg}(s, \sigma(s)) = s[i \mapsto (h_i, h_i - (h_i - r_i))] = s[i \mapsto (h_i, r_i)] = s. \tag{8.4.30}$$

Suppose we run the resource-guarded expression $(\!|\mathsf{mark}_i|\!)_n$ (Eq (8.4.24)) with the resource guards initialized to the resource state $\mathsf{initrg}(s, \sigma(s)) + k$. The resource-guarded expression $(\!|\mathsf{mark}_i|\!)_n$ takes the second branch because we have

$$(\mathsf{initrg}(s, \sigma(s)) + k)(i) = r_i + k_i \geq 1, \tag{8.4.31}$$

which follows from the assumption $r_i \geq 1$. Furthermore, after running the expression $(\!|\mathsf{mark}_i|\!)_n$, the final resource state of the resource guards is

$$\begin{aligned}
&(\mathsf{initrg}(s, \sigma(s)) + k)[i \mapsto && (8.4.32)\\
&\quad ((\mathsf{initrg}(s, \sigma(s)) + k)(i).h, (\mathsf{initrg}(s, \sigma(s)) + k)(i).r - 1)] && (8.4.33)\\
&= \mathsf{initrg}(s, \sigma(s))[i \mapsto (\mathsf{initrg}(s, \sigma(s))(i).h, \mathsf{initrg}(s, \sigma(s))(i).r - 1)] + k && (8.4.34)\\
&= s[i \mapsto (h_i, r_i - 1)] + k && \text{by Eq (8.4.30)} \quad (8.4.35)\\
&= \sigma(s) + k && \text{by Eq (8.4.29).} \quad (8.4.36)
\end{aligned}$$

Thus, the output of running the expression $(\!|\mathsf{mark}_i|\!)_n$ is $\langle \sigma(s) + k, \langle\,\rangle \rangle$. It establishes the proof goal (8.4.27) for the case where $r_i \geq 1$.

Conversely, consider the second case where $r_i = 0$. We have

$$\begin{aligned}
\sigma(s) &= s[i \mapsto (h_i, 0) \oplus (1, 0)] && (8.4.37)\\
&= s[i \mapsto (h_i + 1, 0)] && (8.4.38)\\
\mathsf{initrg}(s, \sigma(s)) &= s[i \mapsto (h_i + 1, h_i + 1 - (h_i - r_i))] && (8.4.39)\\
&= s[i \mapsto (h_i + 1, 1)]. && (8.4.40)
\end{aligned}$$

If we run the resource-guarded expression $(\!|\mathsf{mark}_i|\!)_n$ (Eq (8.4.24)) with the resource guards initialized to $\mathsf{initrg}(s, \sigma(s)) + k$, the expression takes the second branch, and returns the final resource guards of

$$\begin{aligned}
&(\mathsf{initrg}(s, \sigma(s)) + k)[i \mapsto && (8.4.41)\\
&\quad ((\mathsf{initrg}(s, \sigma(s)) + k)(i).h, (\mathsf{initrg}(s, \sigma(s)) + k)(i).r - 1)] && (8.4.42)\\
&= \mathsf{initrg}(s, \sigma(s))[i \mapsto (\mathsf{initrg}(s, \sigma(s))(i).h, \mathsf{initrg}(s, \sigma(s))(i).r - 1)] + k && (8.4.43)\\
&= s[i \mapsto (h_i + 1, 0)] + k && \text{by Eq (8.4.40)} \quad (8.4.44)\\
&= \sigma(s) + k && \text{by Eq (8.4.38).} \quad (8.4.45)
\end{aligned}$$

Thus, again, the output of running the expression $(\!|\mathsf{mark}_i|\!)_n$ is $\langle \sigma(s) + k, \langle\,\rangle\rangle$. This establishes the proof goal (8.4.27) for the case where $r_i = 0$. □

Lemma 8.4.1 (Rewrite $\mathsf{initrg}(\cdot,\cdot)$)**.** *Given two tuples of $\sigma_1, \sigma_2 \in \mathsf{REM}^n$ of resource effects, consider a resource state $s \in (\mathbb{N}^2)^n$. We then have*

$$\mathsf{initrg}(s, (\sigma_2 \circ \sigma_1)(s)) = \mathsf{initrg}(s, \sigma_1(s)) + \overline{(\sigma_2 \circ \sigma_1)(s).h} - \overline{\sigma_1(s).h}, \tag{8.4.46}$$

where $\overline{\sigma_2(\sigma_1(s)).h} - \overline{\sigma_1(s).h}$ is an offset (Defn. 8.4.2).

Proof. Fix arbitrary tuples $\sigma_1, \sigma_2 \in \mathsf{REM}^n$ of resource effects and an arbitrary resource state $s \in (\mathbb{N}^2)^n$. Let

$$s = ((h_{0,i}, r_{0,i}))_{i=1}^n \qquad \sigma_1(s) = ((h_{1,i}, r_{1,i}))_{i=1}^n \qquad \sigma_2(\sigma_1(s)) = ((h_{2,i}, r_{2,i}))_{i=1}^n. \tag{8.4.47}$$

We have

$$\begin{aligned}
\mathsf{initrg}(s, (\sigma_2 \circ \sigma_1)(s)) &= (h_{2,i}, h_{2,i} - (h_{0,i} - r_{0,i}))_{i=1}^n && (8.4.48)\\
&= (h_{1,i}, h_{1,i} - (h_{0,i} - r_{0,i}))_{i=1}^n + ((h_{2,i} - h_{1,i}, h_{2,i} - h_{1,i}))_{i=1}^n && (8.4.49)\\
&= \mathsf{initrg}(s, \sigma_1(s)) + (\overline{(\sigma_2 \circ \sigma_1)(s).h} - \overline{\sigma_1(s).h}), && (8.4.50)
\end{aligned}$$

which establishes the claim (8.4.46). □

Lemma 8.4.2 (Rewrite $\mathsf{initrg}(\cdot,\cdot)$)**.** *Given two tuples of $\sigma_1, \sigma_2 \in \mathsf{REM}^n$ of resource effects, consider a resource state $s \in (\mathbb{N}^2)^n$. We then have*

$$\mathsf{initrg}(\sigma_1(s), (\sigma_2 \circ \sigma_1)(s)) = \sigma_1(s) + \overline{(\sigma_2 \circ \sigma_1)(s).h} - \overline{\sigma_1(s).h}. \tag{8.4.51}$$

Proof. Fix arbitrary tuples $\sigma_1, \sigma_2 \in \mathsf{REM}^n$ of resource effects and an arbitrary resource state $s \in (\mathbb{N}^2)^n$. Let

$$s = ((h_{0,i}, r_{0,i}))_{i=1}^n \qquad \sigma_1(s) = ((h_{1,i}, r_{1,i}))_{i=1}^n \qquad \sigma_2(\sigma_1(s)) = ((h_{2,i}, r_{2,i}))_{i=1}^n. \tag{8.4.52}$$

We have

$$\begin{aligned}
\mathsf{initrg}(\sigma_1(s), (\sigma_2 \circ \sigma_1)(s)) &= (h_{2,i}, h_{2,i} - (h_{1,i} - r_{1,i}))_{i=1}^n && (8.4.53)\\
&= (h_{1,i}, r_{1,i})_{i=1}^n + ((h_{2,i} - h_{1,i}, h_{2,i} - h_{1,i}))_{i=1}^n && (8.4.54)\\
&= \sigma_1(s) + \overline{(\sigma_2 \circ \sigma_1)(s).h} - \overline{\sigma_1(s).h}, && (8.4.55)
\end{aligned}$$

which establishes the claim (8.4.51). □

Proposition 8.4.2 (Approximation relation holds for a let-binding)**.** *Given $d_1 \triangleleft_{\mathsf{F}A} d_2$ and $f_1 \triangleleft_{A \to \mathsf{F}B} f_2$, we have*

$$\underbrace{(x \longleftarrow_{\mathbb{T}} d_1; f_1(x))}_{e_{\text{left}}} \triangleleft_{\mathsf{F}B} \underbrace{(\lambda s.\langle s_1, x\rangle \longleftarrow_{\mathbb{T}} (d_2(s)); f_2(x)(s_1))}_{e_{\text{right}}}. \tag{8.4.56}$$

Proof. Given $d_1 \in [\![\mathsf{F}A]\!]$, $d_2 \in [\![(\!| \mathsf{F}A |\!)_n]\!]$, $f_1 \in [\![A \to \mathsf{F}B]\!]$, and $f_2 \in [\![(\!| A \to \mathsf{F}B |\!)_n]\!]$, assume

$$d_1 \lhd_{\mathsf{F}A} d_2 \qquad f_1 \lhd_{A \to \mathsf{F}B} f_2. \tag{8.4.57}$$

There are two cases for the left-hand side e_{left} in the proof goal (8.4.56): (i) $e_{\text{left}} = \bot$ and (ii) $e_{\text{left}} = \eta_{\mathbb{L}}(\cdot, \cdot, \cdot)$. The first case immediately implies the claim (8.4.56) due to Lem. 8.4.4.

For the second case where $e_{\text{left}} = \eta_{\mathbb{L}}(\cdot, \cdot, \cdot)$, suppose

$$e_{\text{left}} = \eta_{\mathbb{L}}(\sigma, c, b_1) \qquad \text{for some } \sigma \in \mathsf{REM}^n, c \in \mathsf{RM}, b_1 \in [\![B]\!]. \tag{8.4.58}$$

From Defn. 8.2.19 of the bind operator $\longleftarrow_{\mathbb{T}}$, it follows that

$$d_1 = \eta_{\mathbb{L}}(\sigma_1, c_1, a_1) \qquad f_1(a_1) = \eta_{\mathbb{L}}(\sigma_2, c_2, b_1) \qquad \text{for some } \sigma_1, \sigma_2 \in \mathsf{REM}^n, c_1, c_2 \in \mathsf{RM}, a_1 \in [\![A]\!] \tag{8.4.59}$$

such that

$$\sigma = \sigma_2 \circ \sigma_1 \qquad c = c_1 \oplus c_2. \tag{8.4.60}$$

Unfolding the approximation relations $\lhd_{\mathsf{F}A}$ and $\lhd_{A \to \mathsf{F}B}$ (Listing 8.7) appearing in Eq (8.4.57), we obtain

$$a_1 \lhd_A a_2 \qquad \mathsf{runrg}(d_2, \sigma_1, c_1, a_2) \qquad \text{for some } a_2 \in [\![(\!| A |\!)_n]\!] \tag{8.4.61}$$

$$b_1 \lhd_B b_2 \qquad \mathsf{runrg}(f_2(a_2), \sigma_2, c_2, b_2) \qquad \text{for some } b_2 \in [\![(\!| B |\!)_n]\!]. \tag{8.4.62}$$

Unfolding $\mathsf{runrg}(d_2, \sigma_1, c_1, a_2)$ and $\mathsf{runrg}(f_2(a_2), \sigma_2, c_2, b_2)$ (Defn. 8.4.4) yields

$$\forall [s_1 \in (\mathbb{N}^2)^n, k_1 \in (\mathbb{N}^2)^n_{\text{offset}}]. d_2\ (\mathsf{initrg}(s_1, \sigma_1(s_1)) + k_1) = \eta_{\mathbb{L}}(c_1, \langle \sigma_1(s_1) + k_1, a_2 \rangle) \tag{8.4.63}$$

$$\forall [s_2 \in (\mathbb{N}^2)^n, k_2 \in (\mathbb{N}^2)^n_{\text{offset}}]. f_2(a_2)\ (\mathsf{initrg}(s_2, \sigma_2(s_2)) + k_2) = \eta_{\mathbb{L}}(c_2, \langle \sigma_2(s_2) + k_2, b_2 \rangle). \tag{8.4.64}$$

To prove the approximation relation $\lhd_{\mathsf{F}B}$ in the claim (8.4.56), we need to show

$$\begin{aligned} &\mathsf{runrg}(e_{\text{right}}, \sigma, c, b_2) \\ &\iff \forall [s \in (\mathbb{N}^2)^n, k \in (\mathbb{N}^2)^n_{\text{offset}}]. e_{\text{right}}\ (\mathsf{initrg}(s, \sigma(s)) + k) = \eta_{\mathbb{L}}(c, \langle \sigma(s) + k, b_2 \rangle). \end{aligned} \tag{8.4.65}$$

To this end, fix an arbitrary resource state $s \in (\mathbb{N}^2)^n$ and an arbitrary offset $k \in (\mathbb{N}^2)^n_{\text{offset}}$. Suppose we run the computation $e_{\text{right}} \in [\![(\!| \mathsf{F}B |\!)_n]\!]$ with the resource guards being initialized to $\mathsf{initrg}(s, \sigma(s)) + k$. The computation e_{right} first runs $d_2 \in [\![(\!| \mathsf{F}A |\!)_n]\!]$ with the initial resource guards of $\mathsf{initrg}(s, \sigma(s)) + k$. We can rewrite

$$\begin{aligned} \mathsf{initrg}(s, \sigma(s)) + k &= \mathsf{initrg}(s, (\sigma_2 \circ \sigma_1)(s)) + k && \text{by Lem. 8.4.1} && (8.4.66) \\ &= \mathsf{initrg}(s, \sigma_1(s)) + \overline{(\sigma_2 \circ \sigma_1)(s).h} - \overline{\sigma_1(s).h} + k && && (8.4.67) \\ &= \mathsf{initrg}(s, \sigma_1(s)) + k_1, && && (8.4.68) \end{aligned}$$

where we define $k_1 \coloneqq \overline{(\sigma_2 \circ \sigma_1)(s).h} - \overline{\sigma_1(s).h} + k$. It follows from Eq (8.4.63) that the output of running $d_2\ (\mathsf{initrg}(s, \sigma_1(s)) + k_1)$ is the pair $\langle \sigma_1(s) + k_1, a_2 \rangle$, where the first component $\sigma_1(s) + k_1$ is the resource guards' final values.

Next, after running d_2, the computation e_{right} runs $f_2(a_2)(\sigma_1(s)+k_1)$, where $\sigma_1(s)+k_1$ is the resource state produced by d_2. We have

$$\sigma_1(s) + k_1 = \sigma_1(s) + \overline{(\sigma_2 \circ \sigma_1)(s).h} - \overline{\sigma_1(s).h} + k \tag{8.4.69}$$

$$= \mathsf{initrg}(\sigma_1(s), (\sigma_2 \circ \sigma_1)(s)) + k \qquad \text{by Lem. 8.4.2.} \tag{8.4.70}$$

Setting $s_2 \coloneqq \sigma_1(s)$ and $k_2 \coloneqq k$, we appeal to Eq (8.4.64). It gives that the output of running $f_2(a_2)(\sigma_1(s)+k_1)$ is a pair $\langle \sigma_2(\sigma_1(s)) + k, b_2 \rangle$. Therefore, we have

$$e_{\text{right}}\,(\mathsf{initrg}(s, (\sigma_2 \circ \sigma_1)(s)) + k) = \eta_{\mathbb{L}}(c_1 \oplus c_2, \langle (\sigma_2 \circ \sigma_1)(s) + k, b_2 \rangle), \tag{8.4.71}$$

establishing the proof goal (8.4.65). □

Lemma 8.4.3 (Approximation relation is closed under suprema of ω-chains)**.** *Given a type A and an element $d_2 \in [\![(\!| A |\!)_n]\!]$, the predicate (i.e., subset)*

$$P \coloneqq \{d_1 \in [\![A]\!] \mid d_1 \lhd_A d_2\} \tag{8.4.72}$$

is closed under suprema of ω-chains.

Proof. The proof goes by structural induction on the type A. Because the approximation relation $\lhd_A$ is defined compositionally for most cases (Listing 8.3), most cases immediately follow from the inductive hypothesis. The only non-trivial case is $A = \mathsf{F}(\cdot)$, which I focus on in the proof.

Suppose $A = \mathsf{F}A_{\text{inner}}$. Consider an ω-chain $(d_i)_{i\in\mathbb{N}}$ such that $\forall i \in \mathbb{N}.d_i \in P$; i.e.,

$$\forall i \in \mathbb{N}.d_i \lhd_{\mathsf{F}A_{\text{inner}}} d_2. \tag{8.4.73}$$

The goal is to show that $\bigvee d_i \in P$:

$$\bigvee d_i \lhd_{\mathsf{F}A_{\text{inner}}} d_2. \tag{8.4.74}$$

There are two cases for $\bigvee d_i$: (i) $\bigvee d_i = \bot$ and (ii) $\bigvee d_i = \eta_{\mathbb{L}}(\cdot,\cdot,\cdot)$. The first case is trivial. Since $\bigvee d_i = \bot$, it is not of the form $\eta_{\mathbb{L}}(\cdot,\cdot,\cdot)$. Consequently, the definition of the approximation relation $\lhd_{\mathsf{F}A_{\text{inner}}}$ (Listing 8.7) (vacuously) holds, establishing the proof goal (8.4.74).

Consider the second case:

$$\bigvee d_i = \eta_{\mathbb{L}}(\sigma, c, a) \tag{8.4.75}$$

for some tuple $\sigma \in \mathsf{REM}^n$, computational cost $c \in \mathsf{RM}$, and denotation $a \in [\![A_{\text{inner}}]\!]$. It means that, for some $k \in \mathbb{N}$, we have

$$\forall j \geq k.d_j = \eta_{\mathbb{L}}(\sigma, c, a_j) \qquad \bigvee_{j\geq k} a_j = a. \tag{8.4.76}$$

Expanding the definition of the approximation relation $\lhd_{\mathsf{F}A_{\text{inner}}}$ (Listing 8.7) appearing in the assumption (8.4.73), we obtain that there exists $a_2 \in [\![(\!| A_{\text{inner}} |\!)_n]\!]$ such that

$$\forall j \geq k.a_j \lhd_{A_{\text{inner}}} a_2 \qquad \mathsf{runrg}(d_2, \sigma, c, a_2). \tag{8.4.77}$$

Applying the inductive hypothesis to $\forall j \geq k. a_j \lhd_{A_{\text{inner}}} a_2$ in Eq (8.4.77) yields

$$\bigvee_{j \geq k} a_j \lhd_{A_{\text{inner}}} a_2. \tag{8.4.78}$$

Finally, combining Eq (8.4.77) and Eq (8.4.78) yields

$$\bigvee d_i = \eta_{\mathbb{L}}(\sigma, c, a) = \eta_{\mathbb{L}}\left(\sigma, c, \bigvee_{j \geq k} a_j\right) \lhd_{\mathsf{F}A_{\text{inner}}} d_2. \tag{8.4.79}$$

This establishes the proof goal (8.4.74). □

Lemma 8.4.4. *Given a type $A \coloneqq A_1 \to A_2 \to \cdots \to \mathsf{F}A_k$ ($k \in \mathbb{N}$), we have $\bot_{[\![A]\!]} \lhd_A a$ for all $a \in [\![(\!|A|\!)_n]\!]$.*

Proof. First of all, the bottom element $\bot_{[\![A]\!]}$ exists. It is given by the function

$$\bot_{[\![A]\!]} : (a_1, \ldots, a_{k-1}) \in [\![A_1 \times \cdots \times A_{k-1}]\!] \mapsto \bot_{[\![\mathsf{F}A_k]\!]}. \tag{8.4.80}$$

Here, the bottom element $\bot_{[\![\mathsf{F}A_k]\!]}$ exists because $[\![\mathsf{F}A_k]\!]$ is a lift as defined in Eq (8.2.15).

Furthermore, for any $a_k \in [\![(\!|\mathsf{F}A_k|\!)_n]\!]$, we have $\bot_{[\![\mathsf{F}A_k]\!]} \lhd_{\mathsf{F}A_k} a_k$ because it (vacuously) holds according to the definition of the approximation relation $\lhd_{\mathsf{F}A_k}$ (Listing 8.7). Therefore, $\bot_{[\![A]\!]} \lhd_A a$ also holds for any $a \in [\![(\!|A|\!)_n]\!]$. □

Proposition 8.4.3. *Given a type $A \coloneqq A_1 \to \mathsf{F}A_2$, if $f_1 \lhd_{A \to A} f_2$, then $\mathsf{fix}(f_1) \lhd_A \mathsf{fix}(f_2)$.*

Proof. The claim is proved by fixed-point induction (Thm. 8.2.2). Given two denotations $f_1 \in [\![A \to A]\!]$ and $f_2 \in [\![(\!|A \to A|\!)_n]\!]$, assume

$$f_1 \lhd_{A \to A} f_2. \tag{8.4.81}$$

Define a predicate P as

$$P \coloneqq \{a \in [\![A]\!] \mid a \lhd_A \mathsf{fix}(f_2)\}. \tag{8.4.82}$$

The predicate P is closed under suprema of ω-chains (Lem. 8.4.3), and contains the bottom element $\bot_{[\![A]\!]}$ (Lem. 8.4.4). Thus, the predicate P is admissible (Defn. 8.2.15). Consequently, due to fixed-point induction (Thm. 8.2.2), it suffices to prove

$$\forall a \in [\![A]\!]. a \in P \implies f_1(a) \in P. \tag{8.4.83}$$

To this end, fix an arbitrary element $a \in [\![A]\!]$ such that $a \in P$; i.e.,

$$a \lhd_A \mathsf{fix}(f_2). \tag{8.4.84}$$

It follows from the assumption (8.4.81) that

$$f_1(a) \lhd_A f_2(\mathsf{fix}(f_2)), \tag{8.4.85}$$

where the right-hand side is equal to $\mathsf{fix}(f_2)$ as it is a fixed point. Hence, we obtain $f_1(a) \in P$, establishing the proof goal (8.4.83). □

8.5 Integrating Static Analysis and Bayesian Data-Driven Analysis

This section describes the first instantiation of resource decomposition that combines Conventional AARA (§4) and Bayesian data-driven analysis. In this instantiation, Conventional AARA statically infers a cost bound of a resource-guarded program, and Bayesian data-driven analysis statistically infers symbolic bounds of resource components from their high-water-mark measurements. This instantiation rests on a novel Bayesian inference method for learning a function that relates the input size of a program to the high-water mark of a given resource component. The Bayesian inference method is designed specifically to infer symbolic bounds of recursion depths.

8.5.1 Code Annotations and Data Collection

This section discusses code annotations for specifying resource components and a data-collection procedure for high-water-mark measurements of the resource components.

Code annotations Given a target program $P(x)$, let L be a finite set of labels used to identify resource components[5]. Let r_ℓ ($\ell \in L$) denote the resource components that the user wishes to employ in resource decomposition. To specify resource components, the original program $P(x)$ is annotated with two annotations: $\mathsf{mark}_\ell\, q$ and reset_ℓ ($\ell \in L, q \in \mathbb{Z}$). The annotation $\mathsf{mark}_\ell\, q$ is semantically equivalent[6]to a sequence of $|q|$ many annotations mark_ℓ (if $q \geq 0$) or unmark_ℓ (if $q < 0$) introduced in $\textsc{Rpcf}_n$ (§8.1). Let $P_{\mathrm{rd}}(x)$ be the resulting annotated program, which is called a resource-decomposed program.

Data collection Given program inputs $v_1, \ldots, v_N$ ($N \geq 1$), we run the resource-guarded program $P_{\mathrm{rd}}(v_i)$ ($i = 1, \ldots, N$), recording high-water-mark measurements of the resource components to collect observed data $\mathcal{D}$ for Bayesian analysis.

During the program execution, each resource component r_ℓ ($\ell \in L$) maintains a counter with three components: (i) a current value $c_\ell \in \mathbb{Z}$; (ii) a current high-water mark $h'_\ell \in \mathbb{N}$; and (iii) a global high-water mark $h_\ell \in \mathbb{N}$. They are all initialized to zero. The annotations $\mathsf{mark}_\ell\, q$ and reset_ℓ ($\ell \in L, q \in \mathbb{Z}$) manipulate the counter as follows:

1. $\mathsf{mark}_\ell\, q$ increments c_ℓ by $q \in \mathbb{Z}$ units and sets $h'_\ell \leftarrow \max(h'_\ell, c_\ell)$;
2. reset_i updates $h_\ell \leftarrow \max(h_\ell, h'_\ell)$ and resets $c_\ell, h_\ell \leftarrow 0$.

[5]In the formalization of resource decomposition (§8.2–8.4), I use the set $L \coloneqq \{1, \ldots, n\}$ ($n \in \mathbb{N}$) of labels such that the resource components r_ℓ ($\ell \in L$) can be ordered. The ordering simplifies the notation when we define, for example, an ordered tuple $\sigma = (f_1, \ldots, f_n) \in \mathsf{REM}^n$ of resource effects, one for each resource component. However, in practice, it is more user-friendly to use arbitrary labels for identifying resource components than restricting the labels to the set $\{1, \ldots, n\}$.

[6]In practice, it is more user-friendly to offer a single construct $\mathsf{mark}_\ell\, q$ than two distinct constructs mark_ℓ and unmark_ℓ. However, in the formalization of resource decomposition (§8.2–8.4), it is more convenient to restrict $q \in \mathbb{Z}$ in $\mathsf{mark}_\ell\, q$ to the set $\{1, -1\}$.

Each run of $P_{\text{rd}}(v_i)$ $(i = 1, \ldots, N)$ yields high-water-mark measurements $h_{\ell,i}$ $(\ell \in L, i = 1, \ldots, N)$ for the resource components. Reorganizing these measurements, we obtain datasets

$$\mathcal{D}_\ell \coloneqq \{(\ell, v_i, h_{\ell,i}) \mid i = 1, \ldots, N\} \qquad \mathcal{D} \coloneqq \bigcup_{\ell \in L} \mathcal{D}_\ell. \tag{8.5.1}$$

The original program $P(x)$ should be annotated such that the high-water mark h_ℓ is equal to the quantity intended for the resource component r_ℓ $(\ell \in L)$.

8.5.2 Bayesian Data-Driven Resource Analysis

This section describes Bayesian data-driven analysis of resource components. The goal of Bayesian inference is to use observed data $\mathcal{D}_\ell$ $(\ell \in L)$ to learn a symbolic bound $f_\ell(v)$ that relates a program input v with its maximum high-water mark $h_{\ell,v}$ while executing a resource-guarded program $P_{\text{rd}}(v)$.

My collaborators and I have developed a Bayesian inference method tailored for resource components that track the recursion depths of functions. It is motivated by the fact that static resource analysis (e.g., Conventional AARA) often fails on recursive programs, but can still succeed in finding the cost of a *single* recursive step. This is because the body of a (non-nested) recursive function is sequential code, which is straightforward to analyze statically. Decomposing the analysis into per-recursion cost (using Conventional AARA) and the recursion depth (using Bayesian inference) solves the problem.

Language of symbolic bounds Each program input $v : \tau$ is associated with a numeric value $m_\tau(v)$ that denotes its "size". The following domain-specific language (DSL) describes a family of size measures m_τ and corresponding cost bounds p_τ that admit linear and logarithmic expressions:

$$m_{\texttt{unit}} = m_{\texttt{int}} ::= \lambda v.1 \tag{8.5.2}$$

$$m_{L(\tau)} ::= \lambda[v_1, \ldots, v_k].k \mid \lambda[v_1, \ldots, v_k].\max\{m_\tau(v_1), \ldots, m_\tau(v_k)\} \tag{8.5.3}$$

$$m_{\tau_1 \times \tau_2} ::= \lambda\langle v_1, v_2\rangle.m_{\tau_1}(v_1) \mid \lambda\langle v_1, v_2\rangle.m_{\tau_2}(v_2) \tag{8.5.4}$$

$$p_\tau(v) ::= c_0 + c_1 m_\tau(v) \mid c_0 + c_1 \log(1 + c_2 + c_3 m_\tau(v)); \qquad c_0, c_1, c_2, c_3 \in \mathbb{R}_{\geq 0}. \tag{8.5.5}$$

In Eq (8.5.2), the base types have a trivial size measure of 1. In Eqs. (8.5.3) and (8.5.4), the composite types are associated with multiple size measures. For example, given a nested list type $\tau \coloneqq L(L(\texttt{int}))$, one size measure m_τ is the outer list length and another size measure is the maximum inner list length. For a graph algorithm where the input is an adjacency list (which is encoded as a nested list), the first size measure corresponds to the number of vertices, and the second corresponds to the maximum degree.

A recursion-depth bound $p_\tau(v)$ in Eq (8.5.5) can only mention one size measure $m_\tau(v)$. This restriction simplifies the Bayesian probabilistic model (8.5.6)–(8.5.9) while still allowing the DSL to capture recursion-depth bounds of many real-world programs (e.g., UnbalancedBST and Dijkstra in §8.5.3). The DSL can be extended to admit more expressive symbolic bounds.

Bayesian inference for symbolic bounds I next describe how to synthesize functions p_τ (Eq (8.5.5)) for resource component r_ℓ that relate an input $v : \tau$ to the high-water mark $h_{\ell,v}$ of resource component r_ℓ. Let τ be the input data type of a functional program under analysis and assume momentarily that we have already selected a size measure m_τ from the DSL (8.5.2)–(8.5.4), with $|v| := m_\tau(v)$. Let $\mathbf{v} := (v_1, \ldots, v_N)$ be a vector of input values and $\mathbf{h}_\ell := (h_{\ell,1}, \ldots, h_{\ell,N})$ the corresponding high-water marks of the resource component r_ℓ. Rather than pre-specify either a linear or logarithmic symbolic form of the cost bound p_τ from the DSL (Eq (8.5.5)), I leverage Bayesian model averaging [110] to infer the appropriate symbolic expression from the data. That is, I use a probabilistic model $\pi_{\mathbf{v}}(\theta, \mathbf{h}_\ell)$ over a set of latent parameters θ and observable high-water marks $\mathbf{h}_\ell$ as follows:

$$z \sim \text{Bernoulli}(0.5) \tag{8.5.6}$$

$$c_0^{\text{lin}} \sim \chi^2(\gamma_0), c_1^{\text{lin}} \sim \text{LogNormal}(0, \gamma_1) \qquad p_\tau^{\text{lin}} := \lambda v. c_0^{\text{lin}} + c_1^{\text{lin}} |v| \tag{8.5.7}$$

$$c_0^{\log}, c_1^{\log}, c_2^{\log}, c_3^{\log} \sim \chi^2(\gamma_2) \qquad p_\tau^{\log} := \lambda v. c_0^{\log} + c_1^{\log} \log(1 + c_2^{\log} + c_3^{\log} |v|) \tag{8.5.8}$$

$$h_{\ell,i} \sim \begin{cases} \text{Uniform}(0, p_\tau^{\text{lin}}(v_i; \theta^{\text{lin}})) & \text{if } z = 0 \\ \text{Uniform}(0, p_\tau^{\log}(v_i; \theta^{\log})) & \text{if } z = 1 \text{ and } p_\tau^{\log}(v_i; \theta^{\log}) \leq 2 \log_2(|v_i|) \\ \text{Normal}_{[0, p_\tau^{\log}(v_i; \theta^{\log})]}(p_\tau^{\log}(v_i; \theta^{\log}), 1) & \text{otherwise.} \end{cases} \tag{8.5.9}$$

The indicator random variable z in Eq (8.5.6) selects one of the two models for generating the high-water marks. The parameters $\theta^{\text{lin}} := \{c_i^{\text{lin}}\}_{i=0}^1$ for p_τ^{lin} (Eq (8.5.7)) and $\theta^{\log} := \{c_i^{\log}\}_{i=0}^3$ for $p_\tau^{\log}$ (Eq (8.5.8)) are drawn from broad prior distributions, where $\{\gamma_0, \gamma_1, \gamma_2\}$ are hyperparameters. In the linear model, the degree-one coefficient $c_{\text{lin},1}$ is drawn from a log-normal distribution that concentrates around 1, reflecting the domain knowledge that the constants of recursion depth rarely exceed this value.

In Eq (8.5.9), the observed high-water mark $h_{\ell,i}$ is generated according to both z and the value $p_\tau^{\log}(v_i)$ of the logarithmic bound. If $z = 0$, or $z = 1$ and $p_\tau^{\log}(v_i; \theta^{\log}) \leq 2 \log_2(|v_i|)$, then the observed high-water mark is drawn from a uniform distribution between 0 and the cost bound. The cutoff $2 \log_2(|v_i|)$ stems from the domain knowledge that few logarithmic bounds have a constant that exceeds 2. This assumption is particularly useful for inferring recursion-depth bounds of programs (e.g., QuickSort and lookups in (possibly unbalanced) binary search trees), where the true worst-case recursion-depth is linear, but the observed high-water marks $h_{\ell,i}$ exhibit a logarithmic trend when the inputs v_i are generated uniformly at random. Finally, if $z = 1$ and $p_\tau^{\log}(|v_i|; \theta^{\log}) > 2 \log_2(|v_i|)$, then the observed high-water mark $h_{\ell,i}$ is drawn from a truncated normal distribution with a small variance (i.e., 1) to handle outliers and prevent discontinuities in the model.

Ideally, I should set the density of this branch to zero. However, in such a case, if we ran a sampling-based probabilistic inference algorithm on a dataset $\mathcal{D}_\ell$ where some observed high-water mark $h_{\ell,i}$ ($i = 1, \ldots, |\mathcal{D}_\ell|$) exceeds $2 \log_2(|v_i|)$, we would obtain the density of zero whenever $z = 1$, causing trouble to the inference algorithm.

The posterior distribution of the latent parameters $\theta = \{c_{0:1}^{\text{lin}}, c_{0:3}^{\log}, z\}$ conditioned on the

observed data $\mathbf{h}_\ell$ is given by Bayes' rule:

$$\pi_{\mathbf{v}}(\theta \mid \mathbf{h}_\ell) = \frac{\pi_{\mathbf{v}}(\theta, \mathbf{h}_\ell)}{\int_\theta \pi_{\mathbf{v}}(\theta, \mathbf{h}_\ell)\,\mathrm{d}\theta}. \tag{8.5.10}$$

I run a sampling-based probabilistic inference algorithm NUTS [111] in the probabilistic programming language Stan [44] to repeatedly draw approximate samples of θ from the posterior. Using an ensemble $\{(\theta_j^{\mathrm{lin}}; \theta_j^{\mathrm{log}})\}_{j=1}^M$ of M approximate posterior samples of the numeric parameters, the posterior distribution of the model indicator $z \in \{0, 1\}$ is estimated as

$$A_j \coloneqq \prod_{i=1}^{N} 1/p_\tau^{\mathrm{lin}}(v_i; \theta_j^{\mathrm{lin}}); \qquad \Pr(z = 0 \mid \mathcal{D}_\ell) \approx \frac{1}{M}\sum_{j=1}^{M}\left(\frac{A_j}{A_j + B_j}\right); \tag{8.5.11}$$

$$\begin{aligned} B_j \coloneqq\ & 1/p_\tau^{\mathrm{log}}(v_i; \theta_j^{\mathrm{log}}) \cdot \mathbf{1}[p_\tau^{\mathrm{log}}(v_i; \theta_j^{\mathrm{log}}) \le 2\log_2(|v_i|)] \\ & + N(v_i; p_\tau^{\mathrm{log}}(v_i; \theta_j^{\mathrm{log}}), 1) \cdot \mathbf{1}\left[\left(2\log_2(|v_i|) < p_\tau^{\mathrm{log}}(v_i; \theta_j^{\mathrm{log}})\right) \wedge \left(v_i \in [0, p_\tau^{\mathrm{log}}(v_i; \theta_j^{\mathrm{log}})]\right)\right]. \end{aligned} \tag{8.5.12}$$

Here, the function $N(\cdot; \cdot, \cdot)$ is the probability density function of a normal distribution.

Selecting the size measure via mutual information Recall from Eqs. (8.5.3) and (8.5.4) that a composite type τ may be associated with many size measures m_τ. To select m_τ for the symbolic bound p_τ in the probabilistic model (8.5.6)–(8.5.9), we select the one with the highest statistical dependence with the observed high-water marks $\mathbf{h}_\ell$. The quantitative measure of dependence we use is mutual information, which characterizes all types of possible dependencies (e.g., linear, nonlinear, etc.) between a pair of random variables. The method of Kraskov et al. [148] is used to estimate mutual information from finitely many samples $\{(h_{\ell,i}, m_\tau(v_i)) \mid 1 \le i \le N\}$.

Remark 8.5.1 (Statistical Soundness of Resource Decomposition)**.** *The soundness of resource decomposition described in Thm. 8.4.1 remains applicable in the presence of statistical uncertainty from data-driven resource analysis (e.g., Bayesian resource analysis).*

For illustration, suppose we integrate two analysis methods by resource decomposition: (i) Analysis A (e.g., AARA) to infer an overall cost bound of a resource-guarded program; and (ii) Analysis B (e.g., Bayesian inference) to infer symbolic bounds of a resource component. Let $p_A, p_B \in [0, 1]$ be the probabilities that Analysis A and B are sound, respectively. Thm. 8.4.1 states that an overall cost bound of an original program is sound if it is composed of sound bounds from Analyses A and B. Assuming these analyses are conducted independently, the probability of the overall cost bound being sound is therefore at least $p_A \cdot p_B$. In the integration of Conventional AARA and Bayesian analysis, since Conventional AARA guarantees soundness, we have $p_A = 1$. If two constituent analysis methods are both sound (§8.6), we have $p_A = p_B = 1$, resulting in a sound overall bound with probability 1.

More generally, if a data-driven method has a statistical guarantee (e.g., Thms. 7.4.1 and 7.4.2 for Hybrid AARA), then that guarantee is inherited in a relatively simple way when the method is used within resource decomposition. «

8.5.3 Numerical Evaluation

This section describes a prototype implementation and evaluation of resource decomposition instantiated with Conventional AARA and Bayesian data-driven analysis (§8.5.2). The evaluation aims to answer the following questions:

Q1: Can the resource-decomposition technique infer asymptotically tight and sound cost bounds for benchmark programs with challenging constructs, such as functions whose recursion depth scales logarithmically with the input size, or functions where the input size does not decrease at each recursive step?

Q2: How do the accuracy and expressiveness of resource decomposition compare to those of AARA [112] (which uses static analysis) and Hybrid AARA [188] (which uses static analysis and Bayesian inference)?

I do not intend to argue that this instantiation of resource decomposition is strictly superior to Hybrid AARA, which also integrates Conventional AARA and Bayesian inference. Indeed, Hybrid AARA has its own advantage over resource decomposition (§8.8.3). Instead, I aim to demonstrate that, by using resource decomposition to integrate Conventional AARA and Bayesian inference, we can infer more expressive and accurate cost bounds than Conventional AARA or Bayesian inference alone, particularly in those benchmark programs with non-polynomial bounds (e.g., $n \log n$) that Hybrid AARA cannot express. To obtain a resource analysis that is strictly superior to Hybrid AARA, we can simply use resource decomposition to integrate the new data-driven analysis for recursion depth (§8.5.2) with *Hybrid* AARA instead of Conventional AARA.

Benchmarks My collaborators and I have curated a benchmark suite consisting of 13 challenging functional programs in OCaml. §B.1.1 describes these benchmark programs in detail, and §B.2 displays their source code. The resource metric of interest is the total number of function calls (including all recursive calls and helper functions), which serves as a first approximation of the execution time.

To collect a dataset $\mathcal{D}$ for data-driven analysis, I generate program inputs as follows. For input lists x and input graphs $G = (V, E)$, the list lengths $|x|$ and the number of vertices $|V|$ grow exponentially. To fill the content of a list, I sample integers from the interval $[0, 2^{32} - 1]$ uniformly at random (with or without replacement, depending on the benchmarks). The set of edges in a graph is generated randomly by the Erdős-Rényi model of random graphs [75].

Resource components For each benchmark, I manually annotate the source code to obtain two programs: a resource-decomposed and a resource-guarded program. The resource-decomposed code is used for collecting measurements of resource components, and the resource-guarded code is instrumented with resource guards and is used by Conventional AARA to infer overall cost bounds. Listings 2.3a and 2.3b display the two versions of annotated code for MergeSort.

All resource components in the 13 benchmark programs track recursion depths.

- MergeSort, QuickSort, BubbleSort, BellmanFord: One resource component for recursion depth.

- HeapSort and HuffmanCode: Two resource components: the recursion depth of `heapify` ([61, §6.2]) during a heap insertion and the recursion depth of extracting the minimum element.
- UnbalancedBST, RedBlackTree, AVLTree, and SplayTree: Two resource components: the recursion depths of tree insertions and lookups.
- BalancedBST: Two resource components: the recursion depths of merge sort and tree lookups.
- Prim and Dijkstra: Three resource components. The first tracks the recursion depth of the function `heapify` when extracting the minimum element from a heap. The second tracks the recursion depth of the decrease-key operation of a heap. The third tracks the recursion depth of a traversal over each inner list of the adjacency list.

To infer symbolic bounds of resource components, the Bayesian model-selection approach (§8.5.2) is implemented in the Stan [44] probabilistic programming language. It runs sampling-based posterior inference via NUTS [111]. Because Stan does not support discrete random variables, I marginalize out the indicator random variable z (Eq (8.5.6)) from the probabilistic model.

To infer overall cost bounds of programs, I use RaML [117, 118], an implementation of Conventional AARA for OCaml programs. It runs the CLP [82] linear-program solver to infer numeric values for the coefficients in polynomial potential functions.

For each benchmark program $P(x)$ with n resource components, its ground-truth resource-component bound $g_i(x)$ $(i = 1, \ldots, n)$ is manually derived. Let $f(x, \mathbf{r})$ be an overall cost bound of the corresponding resource-guarded program $P_{\text{rg}}(x, \mathbf{r})$ inferred by RaML. A ground-truth overall cost bound of the original program $P(x)$ is then given by $f(x, g_1(x), \ldots, g_n(x))$.

Soundness proportions Tab 8.1 shows the proportions of inferred symbolic bounds of resource components that are sound and the analysis time of benchmark programs. The second column shows the number of lines of code (LOC). The third column shows the ground-truth asymptotic bounds. In sorting algorithms, n is the length of the input list. In tree algorithms, n_1 is the length of the first input list for tree constructions, and n_2 is the length of the second input list for tree lookups. In graph algorithms, $|V|$ is the number of vertices, and d is the maximum degree.

The fourth and fifth columns show inference results of Conventional and Hybrid AARA:

1. *Correct* means the inferred bound is asymptotically tight;
2. *Wrong Asymptotics* means the inferred symbolic bound is asymptotically loose (e.g., a quadratic bound is inferred for MergeSort, while the ground-truth bound is $cn \log n$);
3. *Untypable* means AARA fails to infer a polynomial cost bound.

In the sixth column, i means the i^{th} resource component within a benchmark. The seventh and eighth columns show the proportions of symbolic bounds drawn from posterior distributions that are sound. The seventh column concerns the asymptotic soundness: whether the inferred bounds belong to the correct (linear or logarithm) family of symbolic bounds. The eighth column concerns the soundness of the inferred concrete coefficients (i.e., the coefficients $c_{0:1}^{\text{lin}}$ and $c_{0:3}^{\log}$), which is a stricter notion of soundness than the asymptotic soundness in the seventh

Table 8.1: Effectiveness of resource decomposition as compared to two AARA baselines: the basic method [112] (which uses only static analysis) and the hybrid method [188] (which integrates static and data-driven analysis). The AARA baselines often give incorrect results due to Wrong Asymptotics (A) or Untypable Programs (T). Percentages of sound posterior bounds from methods that use Bayesian inference are shown in parentheses.

			AARA Baselines		**Resource Decomposition (AARA + Data-Driven; §8.5.2)**				
						Sound Bounds		**Analysis Time**	
Benchmark	**LOC**	**Ground Truth**	**Basic** [112]	**Hybrid** [188]	**Guard**	Asymptotics	Coefficients	Data-Driven	Static
MergeSort	29	$n\log(n)$	✗ (A)	✗ (A)	1	✓ (100%)	✓ (59.5%)	6.0 s	0.4 s
QuickSort	22	n^2	✓	✓ (100%)	1	✓ (100%)	✓ (10.5%)	6.3 s	0.3 s
BubbleSort	16	n^2	✗ (T)	✓ (40.1%)	1	✓ (100%)	✓ (73.0%)	9.9 s	0.2 s
HeapSort	87	$n\log(n)$	✗ (T)	✗ (A)	1	✓ (100%)	✓ (31.6%)	3.9 s	3.2 s
					2	✓ (100%)	✓ (43.1%)	4.3 s	
HuffmanCode	121	$n\log(n)$	✗ (T)	✗ (A)	1	✓ (100%)	✓ (31.6%)	0.5 s	6.7 s
					2	✓ (100%)	✓ (39.6%)	4.2 s	
BalancedBST	96	$(n_1+n_2)\log(n_1)$	✗ (A)	✗ (A)	1	✓ (100%)	✓ (27.3%)	14.8 s	2.3 s
					2	✓ (100%)	✓ (7.8%)	10.0 s	
UnbalancedBST	47	$(n_1+n_2)n_1$	✓	✓ (100%)	1	✓ (100%)	✗ (0%)	11.3 s	1.6 s
					2	✓ (100%)	✗ (0%)	14.5 s	
RedBlackTree	65	$(n_1+n_2)\log(n_1)$	✗ (A)	✗ (A)	1	✓ (100%)	✗ (0%)	7.5 s	1023.4 s
					2	✓ (100%)	✗ (0%)	7.9 s	
AVLTree	124	$(n_1+n_2)\log(n_1)$	✗ (T)	✗ (A)	1	✓ (100%)	✗ (0%)	6.5 s	6.4 s
					2	✓ (100%)	✗ (0%)	7.9 s	
SplayTree	103	$(n_1+n_2)n_1$	✓	✓ (100%)	1	✓ (100%)	✗ (0%)	61.5 s	11.1 s
					2	✓ (100%)	✗ (0%)	95.1 s	
Prim	192	$\lvert V\rvert d\log(\lvert V\rvert)$	✗ (T)	✗ (A)	1	✓ (100%)	✓ (69.4%)	16.2 s	2524.4 s
					2	✓ (100%)	✓ (59.8%)	21.0 s	
					3	✓ (100%)	✓ (94.2%)	29.0 s	
Dijkstra	203	$\lvert V\rvert d\log(\lvert V\rvert)$	✗ (T)	✗ (A)	1	✓ (100%)	✓ (69.4%)	15.6 s	3179.3 s
					2	✓ (100%)	✓ (73.6%)	15.2 s	
					3	✓ (100%)	✓ (94.2%)	31.6 s	
BellmanFord	93	$\lvert V\rvert^2 d$	✗ (T)	✗ (0%)	1	✓ (100%)	✓ (56.1%)	31.7 s	6.6 s

column.

For instance, for MergeSort, the ground-truth recursion-depth bound is $1+\log_2(n)$. Given an inferred recursion-depth bound of the form

$$c_0 + c_1 \ln(1 + c_2 + c_3 n), \tag{8.5.13}$$

where $c_0, \ldots, c_3 \in \mathbb{R}_{\geq 0}$, it can be rewritten as

$$c_0 + c_1 \ln(1 + c_2 + c_3 n) = c_0 + c_1 \ln(c_3) + c_1 \ln(2) \log_2\left(\frac{1}{c_3} + \frac{c_2}{c_3} + n\right). \tag{8.5.14}$$

Hence, to check the soundness of coefficients for MergeSort, I check

$$1 \leq c_0 + c_1 \ln(c_3) \wedge 1 \leq c_1 \ln(2). \tag{8.5.15}$$

In RedBlackTree, I use the ground-truth recursion depth of $2\log_2(n)$ for tree insertions and lookups, where n is the number of nodes in the tree. This is because $2\log_2(n)$ is the best theoretical bound of the red-black tree's height I know of. However, this bound is not necessarily the tightest recursion-depth bound.

In SplayTree, I use the ground-truth recursion-depth bound of n for tree insertions and lookups because it is the worst-case height of a splay tree at any point. However, if we consider a sequence of splay-tree operations, as I do here, their logarithmic *amortized* costs come into play. In fact, if we successively insert n_1 elements into the empty splay tree and then perform n_2 lookups, the total cost is $O((n_2+n_1)\log(n_1))$ [208]. On the other hand, resource decomposition infers bounds of the form $O((n_2+n_1)n_1)$ as the worst-case asymptotic complexity of single tree insertions and lookups is $O(n_1)$.

By Remark 8.5.1, the soundness proportion of overall cost bounds is lower-bounded by the product of the soundness proportions of all resource components.

All 13/13 benchmarks achieve 100% sound asymptotic bounds. Furthermore, 9/13 benchmarks have positive proportions of sound coefficients. This demonstrates the effectiveness of the new hybrid-resource-analysis framework in practice. When Conventional AARA fails to infer asymptotically tight bounds (e.g., MergeSort) or any polynomials bounds (e.g., BubbleSort), resource decomposition can successfully infer a posterior distribution containing sound symbolic bounds.

Four benchmarks (i.e., UnbalancedBST, RedBlackTree, AVLTree, and SplayTree) have 0% sound coefficients. UnbalancedBST and SplayTree have the worst-case recursion depth of $1 \cdot n$ for tree insertions and lookups. Although the method correctly infer 100% linear recursion-depth bounds, the coefficient c_1 in the linear bound samples $c_0 + c_1 n$ is below 1 in all posterior samples. This is because the worst-case inputs rarely arise if inputs are generated randomly, and most data points lie well below the line $1 \cdot n$. RedBlackTree has the worst-case recursion depth of $2\log_2(n)$, and AVLTree has the worst-case recursion depth of $\log_\phi(1+\sqrt{5}n)$, where ϕ is the golden ratio. Again, with randomly generated inputs, the worst-case inputs rarely arise. Moreover, it is unclear what the *tight* worst-case recursion depth of RedBlackTree is.

Comparison to Hybrid AARA Because Hybrid AARA can only infer polynomial bounds, it cannot infer asymptotically tight bounds for 8/13 benchmarks whose ground-truth bounds involve logarithm (e.g., MergeSort). Since Conventional AARA is a special case of Hybrid AARA, 3/13 benchmarks that can be analyzed by Conventional AARA are also handled successfully by Hybrid AARA. The remaining 2/13 benchmarks (i.e., BubbleSort and BellmanFord) have ground-truth polynomial bounds and cannot be handled by Conventional AARA. To conduct Hybrid AARA on these two benchmarks, the best we can do is to analyze the entire source code using data-driven analysis on the total costs of the programs. Since Conventional AARA already fails to reason about the outermost functions' recursion depths in both benchmarks, we cannot perform data-driven analysis on a single loop iteration and static analysis on the outermost functions. For BubbleSort, Hybrid AARA's data-driven analysis yields 40.1% sound quadratic bounds [188]. For BellmanFord, it delivers 0.0% sound cubic bounds, even if we only examine the most significant coefficient (for the factor $|V|^2 d$) in the inferred polynomial cost bounds.

Analysis time The last two columns of Tab 8.1 show the analysis time of Bayesian inference for resource components' bounds and Conventional AARA for the overall cost bounds. The analysis time of Bayesian inference is under 2 min. Its variance is due to the variance in

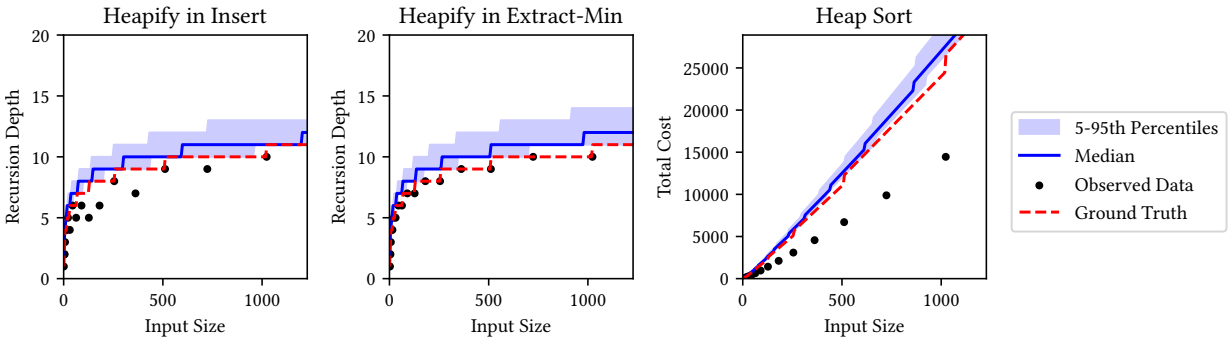


Figure 8.1: Posterior distributions of resource guards and total cost in HeapSort.

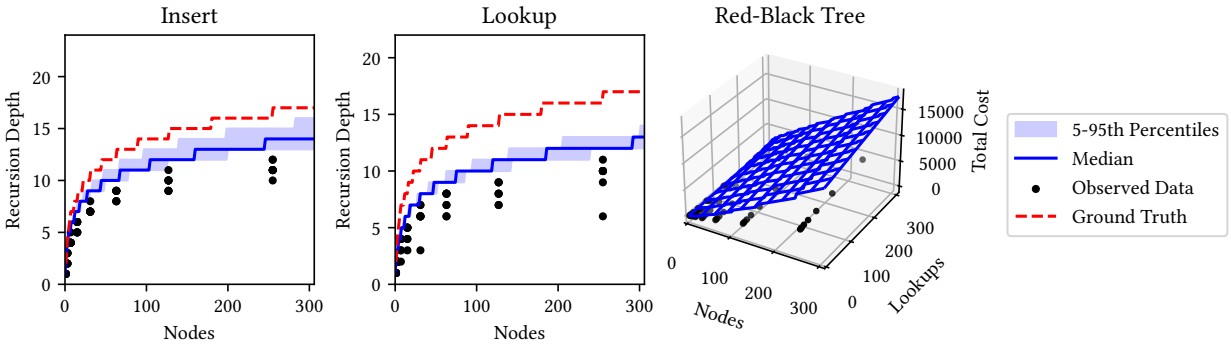


Figure 8.2: Posterior distributions of resource guards and total cost in RedBlackTree.

the dataset sizes $|\mathcal{D}_\ell|$. The analysis time of Conventional AARA varies greatly, from 0.16 s (BubbleSort) to 53 min (Dijkstra). RaML takes a long time for RedBlackTree, Prim, and Dijkstra because their resource-guarded programs yield a large number of linear constraints to solve.

Distributions of inferred bounds Figs. 8.1 and 8.2 display three posterior distributions in HeapSort and RedBlackTree, respectively: two resource components' symbolic bounds and the overall cost bounds. In the two leftmost plots of Fig. 8.1 for HeapSort's resource components, the 5–95$^{\text{th}}$ percentile ranges of the posterior distributions (light-blue shades) have visible width. This variation accounts for the uncertainty of the distance between the true worst-case bounds and the maximum observed values. Furthermore, the probabilistic model used in Bayesian inference is designed such that any sound symbolic bound, including the true bound, has a positive density in the posterior distribution.

In the two leftmost plots in Fig. 8.2 for RedBlackTree's resource components, although the posterior distributions lie above the observed data (black dots), they are still below the ground-truth bound $2\log_2(n)$ (dashed red lines). The ground-truth bound is the best (but not necessarily tight) bound that we know of for the red-black tree's height. If one seeks a more conservative bound, one can adjust the probabilistic model by adding more buffer on top of the maximum observed data.

In the rightmost plots of Figs. 8.1 and 8.2 for the overall cost bounds, the median inferred cost bounds (blue line and surface) are significantly higher than the observed costs (black dots), even though they are very close to each other in the two leftmost plots for individual resource

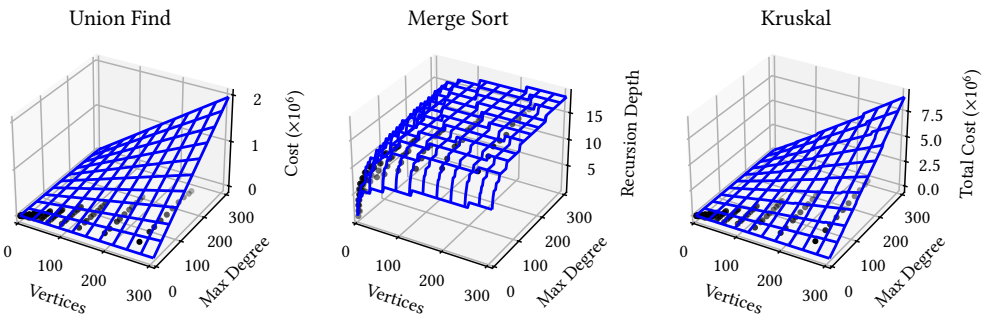


Figure 8.3: Interactively derived bounds of two resource components and the total cost in Kruskal. Blue wireframes are the bounds inferred by AARA+interactive resource analysis, and black dots indicate observed data.

components. This is because the overall cost bounds $f(\mathbf{x}, \mathbf{r})$ of the resource-guarded programs of HeapSort and RedBlackTree, which are statically inferred by Conventional AARA, are sound but not tight with respect to the observed cost measurements.

8.6 Integrating Static Analysis and Interactive Theorem Proving

This section presents an integration of static resource analysis, specifically AARA [112], and *interactive resource analysis*, where cost bounds are derived via interactive theorem provers (e.g., Coq [29] and Agda [182]). While AARA and resource analysis via interactive theorem proving both guarantee soundness of inferred bounds, they differ in expressiveness and automation. AARA is fully automatic, but can only express polynomial cost bounds. Meanwhile, interactive methods let the user manually prove any symbolic cost bounds that is expressible in the underlying program logic. By combining these complementary techniques, we can derive non-trivial cost bounds while reducing the manual work of interactive analysis. The manual work of deriving a cost bound of a code fragment P via interactive theorem proving can be amortized if P is a helper function in many programs that the user would like to analyze.

To showcase this instantiation, I analyze Kruskal's algorithm for minimum spanning trees implemented in OCaml, where the resource metric of interest is the number of function calls. §B.2.14 displays OCaml code of Kruskal. Given a weighted graph, Kruskal first runs MergeSort to sort all edges in the ascending order of their weights. The algorithm next assigns a singleton set (i.e., singleton spanning tree) to each vertex. The algorithm then iteratively merges two sets in the ascending order of edge weights, where sets are tracked by a union-find data structure. Every operation in the union-find data structure has an amortized cost of $O(\alpha(n))$, where $\alpha(\cdot)$ is the inverse Ackermann function and n is the number of elements in the data structure [14, 138, 214, 215].

To apply resource decomposition to Kruskal, I use two resource components: (i) recursion depth of merge sort for sorting all edges; and (ii) total cost of all calls to the union-find data structure. Their symbolic bounds, g_1 and g_2, can be derived by interactive resource analysis [52,

180, 222]:

$$g_1(d, |V|) \coloneqq 1 + \lceil \log_2(d \cdot |V|) \rceil \qquad g_2(d, |V|) \coloneqq c_{\texttt{make}} \cdot |V| + c_{\texttt{eq}} \cdot d \cdot |V| + c_{\texttt{union}} \cdot |V|, \quad (8.6.1)$$

where d is the maximum degree of vertices, $|V|$ is the number of vertices, and c_i is an amortized cost of a union-find operation $i \in \{\texttt{make}, \texttt{eq}, \texttt{union}\}$. Amortized cost bounds of union-find operations are formally derived by Charguéraud and Pottier [52] using Iris with time credits [168]:

$$c_{\texttt{make}} \coloneqq 3 \qquad c_{\texttt{eq}} \coloneqq 4\alpha(|V|) + 9 \qquad c_{\texttt{union}} \coloneqq 4\alpha(|V|) + 12. \quad (8.6.2)$$

Meanwhile, the original program of Kruskal is extended with two resource guards r_i $(i = 1, 2)$ that track the two resource components. Analyzing the resulting resource-guarded program, AARA infers the following overall cost bound for Kruskal parametric in the resource guards:

$$f(d, |V|, r_1, r_2) \coloneqq 7 + 3.5|V| + d \cdot |V| + 3.5d \cdot |V| \cdot r_1 + 1.5r_2. \quad (8.6.3)$$

Finally, substituting resource-component bounds g_i (Eq (8.6.1)) for r_i $(i = 1, 2)$ in Eq (8.6.3) gives an overall cost bound for the original Kruskal:

$$\begin{aligned} f(d, |V|, g_1(d, |V|), g_2(d, |V|)) = 7 + 26|V| + 18d \cdot |V| + 3.5d \cdot |V| \cdot \lceil \log_2(d \cdot |V|) \rceil \\ + 6\alpha(|V|) \cdot d \cdot |V| + 6\alpha(|V|) \cdot |V|. \end{aligned} \quad (8.6.4)$$

This bound is sound because, by Thm. 8.4.1, composing the resource components' sound bounds (8.6.1) and the resource-guarded program's sound bound (8.6.3) yields a sound bound of the original Kruskal. The bound (8.6.4) is $O(d \cdot |V| \log(d \cdot |V|))$, which is dominated by the cost of merge sort.

Fig. 8.3 displays inferred bounds (blue wireframes) for the two resource components and total cost of Kruskal. The bounds are sound with respect to the observed cost measurements (black dots).

8.7 Integrating SMT-Based Semi-Automatic Analysis with Bayesian Data-Driven Analysis

This section demonstrates a third instantiation of resource decomposition that integrates SMT-based semi-automatic resource analysis and Bayesian data-driven resource analysis. For illustration, I analyze quicksort on lists of natural numbers, where the resource metric of interest is the number of function calls. To compare natural numbers, the comparison function converts them to binary encodings and traverses them. Hence, an overall cost bound of quicksort is $O(n^2 \log m)$, where n is the input list length and m is the maximum number appearing in the list. §B.2.15 displays the source code.

TiML [222] is an SMT-based semi-automatic resource-analysis method for Standard ML programs. In TiML, the user first annotates a program to define sizes of data structures and specify candidate symbolic cost bounds. Walking through the annotated code, a type checker generates verifications conditions (VCs), which are then automatically checked by the SMT

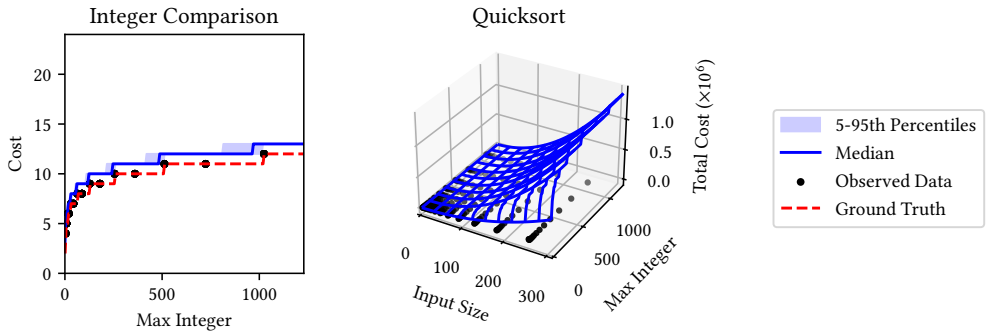


Figure 8.4: Posterior distributions of a resource guard and total cost in QuickSortTiML.

solver Z3 [72]. TiML is sound: if Z3 successfully verifies the VCs, the user-supplied candidate bounds must be worst-case bounds.

To analyze quicksort described above (dubbed QuickSortTiML), I set a resource component to be the logarithmic cost of the comparison function. TiML struggles to verify *concrete* bounds with logarithm because SMT solvers do not handle logarithm well. TiML can only infer *asymptotic* bounds with logarithm (e.g., MergeSort) by pattern-matching VCs with the Master Theorem [222]. To complement TiML, I use Bayesian data-driven analysis to statistically infer a logarithmic symbolic bound of the resource component. I run QuickSortTiML on many inputs, recording the cost of the comparison function and two input sizes n (input list length) and m (maximum number in the list). For Bayesian inference of the comparison function's cost, I reuse the same probabilistic model (8.5.6)–(8.5.9) from the first instantiation of resource decomposition (§8.5.2), which can infer logarithmic and linear symbolic bounds.

Meanwhile, I use TiML to verify a candidate cost bound of the resource-guarded code of QuickSortTiML, which extends the original code with a resource guard to track the comparison function's cost. TiML successfully verifies a user-supplied cost bound

$$f(n, m, r) \coloneqq 2n + n(n+1)(r+2), \tag{8.7.1}$$

where n and m are the two original input sizes and r is the resource guard. Finally, I substitute a statistically inferred resource-component bound, say $g(n, m) = \lceil 1.1 + 1.2 \log_2(m+1) \rceil$, for r in Eq (8.7.1). It yields an overall cost bound for the original QuickSortTiML:

$$f(n, m, g(n, m)) = 2n + n(n+1)(\lceil 1.1 + 1.2 \log_2(m+1) \rceil + 2). \tag{8.7.2}$$

Fig. 8.4 displays posterior distributions of the resource guard and total cost of QuickSortTiML. 8410/12000 (70.1%) posterior samples of the resource-component bounds are sound with respect to the ground-truth bound $1 + \lceil \log_2(m+1) \rceil$ of the comparison function. Hence, due to Remark 8.5.1 and the soundness guarantee of TiML, the posterior distribution of overall cost bounds, which are given by composing TiML's and Bayesian analysis's inferred bounds, also have the soundness proportion of 70.1%.

Even though TiML struggles to verify symbolic bounds involving logarithm, it is still beneficial to retain it, instead of resorting to Bayesian analysis exclusively. Fig. 8.4 (right plot) exhibits a large gap between observed total costs (black dots) and the ground-truth total cost bound (which is close to the blue wireframe). If we conducted *purely* data-driven analysis without TiML, the inferred overall bounds could be unsound with a high probability.

8.8 Discussion

This chapter has introduced the second hybrid resource analysis: resource decomposition. In this section, I first reflect on the design of resource decomposition, discussing its generalization (§8.8.1). Next, I discuss how to identify suitable resource components in practice (§8.8.2). Finally, I compare resource decomposition with Hybrid AARA, describing their upsides and downsides (§8.8.3).

8.8.1 Interface Design of Resource Decomposition

This section discusses the resource-guard-mediated interface between constituent analyses in resource decomposition. It also discusses possible generalization of resource guards.

User-adjustable interface Similarly to Hybrid AARA, resource decomposition has a user-adjustable interface: the user can freely specify what quantities should be analyzed by one of the constituent analysis methods. Consequently, resource decomposition covers a spectrum ranging from one analysis to another analysis, as does Hybrid AARA. If the user specifies no resource components, a whole program is analyzed by one analysis method. On the other hand, if a single resource component represents the cost of a whole program, then the program is analyzed by the other analysis method.

Variable-based interface In resource decomposition, the interface between constituent analyses is a resource guard, which is a numeric non-negative variable. Let $P_{\mathrm{rd}}(x)$ be a resource-decomposed program and $P_{\mathrm{rg}}(x, \mathbf{r})$ be a resource-guarded program with resource guards $\mathbf{r}$. One constituent analysis analyzes the program $P_{\mathrm{rg}}(x, \mathbf{r})$ to infer an overall cost bound $f(x, r_1, \ldots, r_{|\mathbf{r}|})$, while the other constituent analysis examines the program $P_{\mathrm{rd}}(x)$ to infer symbolic bounds $g_i(x)$ $(i = 1, \ldots, |\mathbf{r}|)$ of high-water marks of the resource components. The inferred symbolic bounds $f(x, r_1, \ldots, r_{|\mathbf{r}|})$ and $g_i(x)$ $(i = 1, \ldots, |\mathbf{r}|)$ are composed by substituting the latter for the resource guards r_i in the former. Thus, resource guards act as an interface between the inference results of the constituent analyses.

In the literature of program analysis, numeric-variable-based interfaces are not used outside resource analysis. This is because many program-analysis tasks concern logical properties of programs, while numeric variables are only useful for quantitative properties (e.g., cost bounds). Within the literature of resource analysis, as numeric-variable-based interfaces, SPEED [102] uses *counters*, and **calf** [180] uses *clocks*. §5.2.1 and §5.2.3 describe similarities and differences between resource guards and counters (in SPEED) and clocks (in **calf**), respectively.

Comparison with ghost variables Tab 8.2 compares resource components, resource guards, and ghost variables. Resource components and ghost variables have similar functionalities: they are both placed in original programs and do not modify their operational semantics. Hence, resource components and ghost variables are both *ghosts*. Nonetheless, they have different use cases and serve different roles. Resource components are used in hybrid resource analysis: they specify what quantities are analyzed by constituent analysis methods of the resource-decomposition framework. Meanwhile, ghost variables are used in program verification and

Table 8.2: Comparison of resource components, resource guards, and ghost variables.

	Resource Component	Resource Guard	Ghost Variable
Placement	Original program	Modified program	Original program
Control Flow	Unmodified	Modified	Unmodified
Use Case	Hybrid resource analysis	Hybrid resource analysis	Program verification and contracts
Role	Specify a resource metric to be analyzed by one of the constituent analyses	Create a resource-guarded program whose cost bound is parametric in resource components	Store old values of mutable program variables or represent extra-functional properties [125]

contracts: ghost variables retain old values of mutable program variables or represent extra-functional properties (e.g., resource usage) [125].

More importantly, the chief novelty of the resource-decomposition framework is not resource components, but the automatic insertion of resource guards according to user-specified resource components. Program verification via ghost variables has no counterparts to this automatic insertion of resource guards. Resource components only serve as user annotations for specifying what quantities are tracked by resource guards.

Generalization of resource components and guards Resource guards can be generalized to more complex data types (e.g., lists and trees) to store more information. In the current formulation of resource decomposition, resource guards are numeric variables tracking user-specified resource components. Thus, a single resource guard only captures one number, that is, one quantitative aspect of an inference result. Examples include the cost of a code fragment, the recursion depth of a function, and the number of recursive calls of a function.

If a resource guard is extended from a numeric type (e.g., $\mathbb{N}$ and $\mathbb{Q}_{\geq 0}$) to a more complex data type, it can capture richer information about an inference result. If a resource guard has a sum type $\tau_1 + \tau_2$ (e.g., Booleans), it can capture which branch of an if-else expression is taken. Furthermore, if a resource guard has a list type $L(\tau_1 + \tau_2)$, it can represent a chain of branches taken in a sequence of if-else expressions. Lastly, with a tree-typed resource guard, it can represent a recursion tree (i.e., tree-shaped visualization of recursive calls) of a function.

A technical challenge of this idea is that we would need to develop an analysis method to infer symbolic bound returning non-numerical objects (e.g., lists and trees). Also, upper bounds only make sense when some ordering (e.g., partial orders and total orders) exists, but it does not necessarily exist for resource guards of non-trivial types.

8.8.2 Using Resource Decomposition in Practice

This section discusses how to identify suitable resource components.

Choice of resource components As with program annotations in other program-analysis settings (e.g., loop invariants and ghost variables), identifying suitable resource components is an essential part of developing an effective instantiation of resource decomposition. These

components must be carefully selected by the user such that the resource guards in the resulting resource-guarded program can each be analyzed by existing resource-analysis methods. So the resource components must be developed in tandem with the chosen resource analyses, and this requires knowledge of both the program being analyzed and the strengths and limitations of the analysis methods.

Typically, resource components capture quantities that cannot be analyzed by the analysis technique that analyzes a resource-guarded program. This chapter has demonstrated three examples of resource components: recursion depths (§8.5), the total cost of union-find operations (§8.6), and the individual cost of a comparison function (§8.7). In §8.5 and §8.6, Conventional AARA infers an overall cost bound $f(x, \mathbf{r})$ of a resource-guarded program. Just like any other static resource analysis, Conventional AARA can fail to statically infer recursion-depth bounds (e.g., $O(n)$ recursion depth of BubbleSort). Hence, in §8.5, resource components are set to recursion depths, and their symbolic bounds are inferred by a different analysis technique, particularly Bayesian analysis. Similarly, Conventional AARA is unable to express non-polynomial symbolic bounds (e.g., logarithmic recursion-depth bounds of MergeSort and the inverse Ackermann function appearing in the time complexity of a union-find data structure). So resource components are set to such quantities in Kruskal (§8.6), and their bounds are derived by interactive resource analysis.

Automatic selection of resource components It is possible to automate the selection of resource components. Given a target program $P(x)$, suppose we analyze it by the integration of static and data-driven analyses as in §8.5. Our goal is to infer a cost bound of the program P with maximal reliance on static analysis. If static analysis fails to analyze the whole program P, we incrementally expand the scope of data-driven analysis until we derive a bound. We examine functions in the program P in the reversed topological order in the call graph. For each function f, we test if the function f can be analyzed by static analysis. If not, we narrow the scope of static analysis by trying out candidate resource components:

1. The recursion depth of the function f;
2. The number of recursive calls to the function f;
3. The maximum cost of a single function call to the function f (including its recursive calls to itself); and
4. The total cost of all function calls to the function f made in the program P.

If a resource component is set to the cost of the program P's main function (which is the last function to examine in the reversed topological order in the call graph), resource decomposition boils down to fully data-driven analysis.

This automatic procedure for selecting resource components stems from an observation that static analysis commonly fails for the following reasons: (i) the analysis is unable to bound the number of recursive calls (or loop iterations); or (ii) the analysis is unable to bound the size of an argument to a function in the program P. Given a recursive function f, if static analysis does not fail for the second reason, but the first reason, we can analyze (i) the recursion body by static analysis; and (ii) the number of recursive calls by data-driven analysis. However, if static analysis fails for the second reason, we have a resource component track the maximum cost of

Table 8.3: Comparison of Hybrid AARA and resource decomposition.

	Hybrid AARA (§7)	Resource Decomposition (§8)
Interface	Resource-annotated typing judgment	Resource guard
Outer Inference Result	Resource-annotated typing tree	Symbolic bound $f(x, r)$ of a resource-guarded program
Inner Inference Result	Resource-annotated typing judgment	Symbolic bound $g(x)$ of a resource component
Information Exchange	Linear constraints are propagated from Conventional AARA to data-driven analysis	No exchange of information between constituent analyses
Inferred Bounds	Polynomial bounds	Any symbolic bounds
Source Code	Constituent analyses only need black-box access to each other's code fragments	Constituent analyses may need white-box access to each other's code fragments
Analysis Methods	Modified to incorporate linear constraints	Unmodified
Combined Analyses	Conventional AARA + data-driven analysis	Any

a single function call to f (including its recursive calls to itself). Lastly, if the total number of calls to the function f made in the program P cannot be statically analyzed, we have a resource component track the total cost of all calls to the function f.

8.8.3 Comparison between Hybrid AARA and Resource Decomposition

This section compares the two hybrid resource analyses introduced in this thesis: Hybrid AARA and resource decomposition. Tab 8.3 summarizes the comparison.

Interface design Hybrid AARA and resource decomposition differ in two aspects of their interface designs:

1. Encodings of inference results; and
2. Exchange of information between constituent analyses.

For an interface between inference results, Hybrid AARA uses types (namely resource-annotated types), while resource decomposition uses numeric variables (namely resource guards). In Hybrid AARA, data-driven analysis infers a resource-annotated typing judgment J_{anno}, and Conventional AARA infers a resource-annotated typing tree T_{anno}. To compose the two inference results, the typing judgment J_{anno} is inserted into the typing tree T_{anno}. In resource decomposition, one analysis infers a resource-component bound $g(x)$ and another analysis infers an overall cost bound $f(x, r)$ parametric in both an original input x and a resource guard r. The two inference results are composed by substitution, resulting in $f(x, g(x))$. This difference is the most fundamental difference between the two hybrid resource analyses. It is responsible for the other differences as discussed below.

Hybrid AARA and resource decomposition differ in whether they exchange information between constituent analyses while they are conducted. Hybrid AARA propagates linear con-

straints from Conventional AARA to data-driven analysis *before* deriving an overall cost bound (by either optimization or sampling). Specifically, Hybrid Opt and Hybrid BayesWC merge linear constraints from Conventional AARA with linear constraints from the respective data-driven analyses, solving the resulting linear program for an overall cost bound. Hybrid BayesPC uses linear constraints from the static part to restrict the state space of a sampling algorithm in Bayesian inference. By contrast, resource decomposition does not exchange constraints between constituent analyses. They are performed *independently* to obtain their respective symbolic bounds, which are then composed by substitution.

In Hybrid AARA, the need to exchange information (i.e., linear constraints) between constituent analyses arises from the use of resource-annotated types in the encoding of inference results. Given an annotated expression $\mathrm{stat}_\ell\ e$ ($\ell \in \mathcal{L}$), the data-driven part of Hybrid AARA infers a resource-annotated typing judgment $\Gamma; p_0 \vdash_{\mathcal{D}} \mathrm{stat}_\ell\ e : \langle a, q_0 \rangle$, which assigns potential functions to the input and output of the expression e. Given a fixed net-cost bound of the expression e, there exist (infinitely) many ways to assign input and output potential functions. An appropriate resource-annotated typing judgment is determined only when we know how much output potential must remain after the expression e is executed. Therefore, before inferring a resource-annotated typing judgment, the data-driven part of Hybrid AARA must receive linear constraints from the static part of Hybrid AARA.

Expressible symbolic bounds Hybrid AARA and resource decomposition have a trade-off in their expressible symbolic bounds. Thus, neither of them is more expressive than the other.

Thanks to the compositionality of types, symbolic bounds inferred by constituent analyses of Hybrid AARA are *local*: they are parametric in the inputs (and outputs) of local code fragments. For illustration, consider an annotated code fragment $\mathrm{stat}_\ell\ e$ ($\ell \in \mathcal{L}$). Data-driven analysis inside Hybrid AARA infers a resource-annotated typing judgment of the expression e. It is a symbolic bound parametric in the input and output of e, which is the scope of data-driven analysis, rather than the input and output of the whole program. However, Hybrid AARA can only express polynomial bounds due to the fact that resource annotated types only capture polynomial potential functions.

On the other hand, for resource decomposition, resource-component bounds are *global*: they are parametric in the inputs of entire programs. Let $P_{\mathrm{main}}(x)$ be a target program for resource analysis and $P_{\mathrm{helper}}(y)$ be a (helper) function defined inside the program $P_{\mathrm{main}}(x)$. Suppose we set a resource component to the recursion depth of the function $P_{\mathrm{helper}}(y)$. Resource-component bounds can have arbitrary shapes, including non-polynomial bounds. However, resource-component bounds must be parametric in the *global input* x (and also the output if the user wishes) of the target program $P_{\mathrm{main}}(x)$, as opposed to the *local input* y of the helper function $P_{\mathrm{helper}}(y)$. This is unsatisfactory since it would be more sensible to derive a recursion-depth bound of $P_{\mathrm{helper}}(y)$ in terms of its own input y.

If we instead derive a recursion-depth bound $f(|y|)$ parametric in the input size of $P_{\mathrm{helper}}(y)$, we additionally need to obtain a bound $|y| \leq g(|x|)$ relating the local input size $|y|$ and the global input size $|x|$. The bound $|y| \leq g(|x|)$ is used to create a recursion-depth bound $f(g(|x|))$ parametric in the global input x. Otherwise, if we simply substitute the resource-component bound $f(|y|)$ for a resource guard, the resulting overall cost bound of $P_{\mathrm{main}}(x)$ mentions the

local input y. This is undesirable as, in resource analysis, cost bounds should only be parametric in the global input x.

Black-box access to the source code Hybrid AARA is more suitable than resource decomposition if some code fragments are black boxes (i.e., their code is unavailable publicly). In Hybrid AARA, given an annotated code fragment $\mathrm{stat}_\ell\ e$ ($\ell \in \mathcal{L}$), the static part needs no access to the expression e's internal code. This is because resource-annotated typing judgments, which are used as an interface between constituent analyses in Hybrid AARA, store information about not only the costs of code fragments but also how their inputs sizes change. On the other hand, in resource decomposition, a single resource component only captures one number. For example, if a resource component is set to the high-water-mark cost of a code fragment e, the resource component cannot capture the net cost or size-change information of e. As a result, when a resource-guarded program is analyzed to infer an overall cost bound, we cannot omit the code fragment e, since the analysis may require to know how the expression e changes the input size.

To fix this drawback of resource decomposition, we may use another resource component to represent the output size of the expression e. This works if e has a simple data type (e.g., non-nested lists), and its size is expressible as a high-water mark of annotations mark and unmark. But if it has a complex data type (e.g., nested lists) with multiple size measures (e.g., the outer list length and the combined inner list length), resource composition may not work effectively.

Black-box access to analysis methods Resource decomposition is more suitable than Hybrid AARA if constituent analyses to be integrated are black boxes (i.e., their implementations cannot be easily modified). In resource decomposition, constituent analyses methods are performed individually, and their inferred symbolic bounds are composed by substitution. By contrast, Hybrid AARA, particularly Hybrid BayesPC, propagates linear constraints from the static part to the data-driven part to restrict the latter's search space. Hence, the data-driven analysis method must be modified to incorporate linear constraints. For instance, the prototype implementation of Hybrid BayesPC (§7.5.3) uses the sampling algorithm Reflective Hamiltonian Monte Carlo (ReHMC) [47, 49, 50, 171], which adapts the conventional HMC be able to incorporate linear constraints. If a different data-driven method is to be used (e.g., large language models (LLMs) [51]), its implementation needs to be modified to be used in Hybrid BayesPC.

Range of composable analysis techniques Resource decomposition is more versatile than Hybrid AARA in the integration of analysis techniques: the former imposes fewer restrictions on the kind of resource analyses it can integrate. In resource decomposition, one constituent analysis infers a resource-component bound from a resource-decomposed program, and the other analysis infers an overall cost bound of a resource-guarded program. For both of the constituent analyses, their inference results are symbolic bounds parametric in the program inputs. Therefore, any analysis techniques can be integrated by resource decomposition as long as their inference results are symbolic bounds. Hybrid AARA, on the other hand, is only applicable to the integration of Conventional AARA and another analysis method (e.g., data-driven analysis) that can infer resource-annotated typing judgments. This is due to the use of

resource-annotated types in the interface design of Hybrid AARA.

Conceptually, Hybrid AARA is not specific to Conventional AARA—any technique can be used in place of Conventional AARA as long as it returns a symbolic bound with a placeholder for a resource-annotated typing judgment. However, in practice, only Conventional AARA, which returns resource-annotated typing trees, meets this criterion.

Chapter 9

Optimization of Probabilistic Program-Input Generators

This chapter presents optimization of program-input generators. Given a target program $P(x)$ of data-driven resource analysis, a program-input generator generates inputs $v_1, \ldots, v_N$ to the program $P(x)$. The inputs v_i $(i = 1, \ldots, N)$ are used to collect cost measurements of $P(x)$. Typically, data-driven analysis has no control over the data-collection procedure: fixed default generators are used to generate program inputs in data-driven analysis.

In this chapter, I present a new data-driven-analysis methodology where statistical analysis infers not only worst-case cost bounds but also worst-case generators for data collection. §9.1 motivates the optimization of generators. §9.2 presents a domain-specific language (DSL) of generators where (i) all values of a user-specified target size have positive probabilities of being generated; and (ii) the DSL admits generators of any algebraic data types. §9.3 presents a genetic-algorithm-based optimization procedure for generators. Finally, §9.4 evaluates the optimization algorithm of generators. The empirical evaluation shows that optimized generators trigger higher costs than baseline generators that draw integers uniformly at random from a broad interval. However, optimized generators sometimes fail to trigger the worst-case costs.

9.1 Introduction

Random input generation Data-driven resource analysis in the literature [94, 131, 234] typically generates program inputs *randomly* (e.g., sample integers uniformly at random from some interval), running an input program on these inputs to record their cost measurements. Randomly generated inputs do not always exhibit worst-case costs. As a result, Opt (§7.3.2), which optimizes a symbolic cost bound without adding an extra buffer on top of maximum observed costs, may return unsound bounds (see Fig. 7.2a). Bayesian data-driven analysis methods BayesWC (§7.3.3) and BayesPC (§7.3.4) partially mitigate the unsoundness of Opt by (i) allowing the user to specify how conservative inferred cost bounds should be in a probabilistic model; and (ii) returning a robust posterior distribution of cost bounds where any bound has a positive probability density (Eq (7.3.9)).

Nonetheless, even with the ability to adjust the gaps between observed costs and predicted

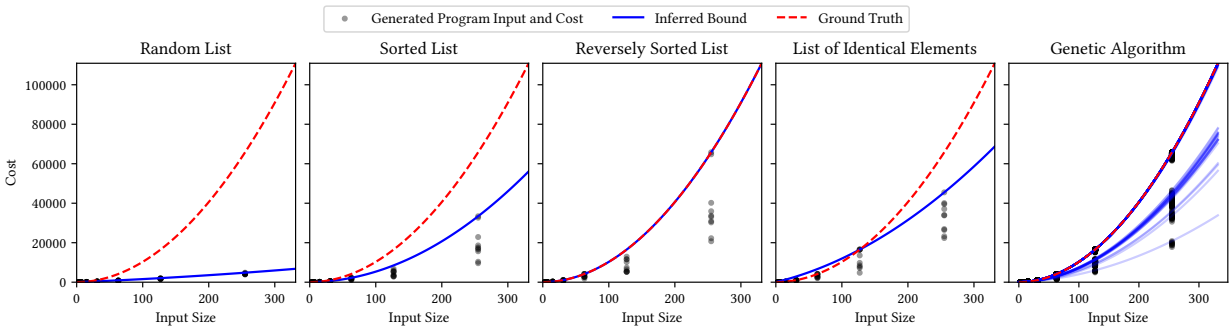


Figure 9.1: Runtime-cost data generated by probabilistic program-input generators for QuickSort. For each input size, 10 integer lists are generated. Inferred cost bounds (blue lines) are derived by Opt (§7.3.2), which optimizes a cost bound subject to the constraint that the bound lies above all cost measurements. The rightmost plot displays the runtime-cost data of the probabilistic generators obtained by a genetic algorithm.

cost bounds, Bayesian data-driven analysis can still struggle to infer sound cost bounds from cost measurements of randomly generated inputs. This happens when the average-case complexity of a program is significantly lower than its worst-case complexity. For example, in QuickSort, the worst-case time complexity is $O(n^2)$, while the average-case complexity over lists of uniformly distributed random integers is $O(n \log n)$. Furthermore, the time complexity of QuickSort concentrates tightly around $O(n \log n)$ [162, 163, 164, 221]. Unless this knowledge is incorporated into a probabilistic model, it is difficult for data-driven analysis to correctly infer an $O(n^2)$ worst-case cost bound.

In Fig. 9.1, the leftmost plot shows the costs of QuickSort on integer lists whose elements are drawn uniformly at random from an interval $[-500, 500]$. Here, the resource metric of interest is the number of function calls (including helper functions). For each input size, 10 integer lists are generated. In the plot, the black dots indicate the costs of program inputs, the blue line is a quadratic cost bound inferred by conducting Opt on the generated runtime-cost data, and the red dashed line is the worst-case cost bound of QuickSort. Indeed, for the input size of $n = 256$ in the plot, the worst-case cost (red dashed line) is significantly higher than the maximum observed cost of 10 randomly generated integer lists (black dots).

Hybrid resource analysis To further mitigate the unsoundness of data-driven analysis, hybrid resource analysis developed in this thesis combines data-driven analysis it with sound resource analysis. For example, Hybrid AARA (§7) splits the source code into two disjoint code regions, which are respectively analyzed by Bayesian data-driven analysis and Conventional AARA. Also, in resource decomposition, the first instantiation (§8.5) performs Bayesian inference on recursion depths of functions and Conventional AARA on the costs of individual recursive calls. However, hybrid resource analysis only works when the source code of an input program is available.

Generator optimization Orthogonal to hybrid resource analysis, another approach to mitigating the unsoundness of data-driven analysis is to search a diverse set of *program-input gener-*

ators for an optimal generator for a target program. The suggested approach works as follows. Given a program $P : \tau_1 \to \tau_2$, let $\mathcal{G} = \{G_1, G_2, \ldots\}$ be a (possibly infinite) set of program-input generators of type τ_1. Generators are programs that, when executed, generate values of type τ_1 of a specified input size. We run an optimization algorithm (e.g., genetic algorithm) in the space $\mathcal{G}$ to identify an optimal (or nearly optimal) generator G_{opt} that triggers high computational cost of the target program P. The generator G_{opt} is then executed to generate program inputs $v_1, \ldots, v_N$ of varying input sizes. Finally, the program P is executed on inputs $v_1, \ldots, v_N$ to construct runtime-cost data $\mathcal{D}$ to be used in statistical analysis.

The optimization of program-input generators is allowed to be *adaptive*: it runs an input program $P : \tau_1 \to \tau_2$ on some program input $v : \tau_1$ and uses its cost measurement to adapt the next search space to focus on. By contrast, in existing data-driven analysis, the data-collection procedure uses a *fixed* set of program inputs to record cost measurements, regardless of an input program P.

A generator is a domain-specific program that has the same inductive structure as the target data type τ_1 of values to generate. A generator takes as input the size of a value to generate. A genetic algorithm is employed to search for an optimal (or near-optimal) generator. The genetic algorithm maintains a finite pool, called a population, of generators. In each iteration, a new population is created by performing genetic operations (e.g., mutations and crossovers) on the abstract syntax trees (ASTs) of generators in the current population[1].

An alternative to the optimization of generators is to optimize costs of individual input sizes. We first perform fuzzing to identify worst-case (or nearly worst-case) inputs for each input size [42, 152, 155, 160, 181, 185, 219, 226]. The worst-case costs discovered by fuzzing for different input sizes are then aggregated into a dataset $\mathcal{D}$ of cost measurements.

Compared to this alternative approach, the optimization of generators is more scalable: while the search space of program inputs grows exponentially in the input size, while the search space of generators is independent of the input size. Wei et al. [224] demonstrate that Singularity, a fuzzer of generators they have developed, is more effective than a program-input fuzzer when the input size is large. Furthermore, generator fuzzing has an advantage that, once a desirable generator is identified, it can generate values of any input sizes subsequently, without the need to running a generator-optimization algorithm again. On the other hand, in program-input fuzzing, every time the user wants a value of a new input size, a fuzzer must be invoked, which can be time-consuming.

Fig. 9.1 shows runtime-cost data generated by probabilistic generators besides a random one: (strictly) sorted lists[2], (strictly) reversely sorted lists, and lists of identical elements. Unlike Singularity [224], these generators are all probabilistic: they can generate different values every time they are executed. Furthermore, for a fixed list length, any content of a generated list has a positive probability of being generated, subject to the constraint that integers come from the interval $[-2^{10}, 2^{10} - 1]$. The rightmost plot in Fig. 9.1 shows the runtime-cost data of generators obtained by a genetic algorithm. The best generator from the genetic algorithm successfully generates runtime-cost data close to the ground-truth $O(n \log n)$ cost bound (red dashed line).

[1]The use of genetic algorithms to search for an optimal program is known as genetic programming [146, 147].

[2]A worst-case input of QuickSort in Fig. 9.1 is a non-strictly reversely sorted list, which can be a list where all elements are identical. The resource metric of interest is the number of function calls, including helper functions. A

9.2 Language of Probabilistic Program-Input Generators

This section introduces a domain-specific language (DSL) of probabilistic program-input generators. A generator is a program that probabilistically generates values of a fixed algebraic data type and a fixed target size. The DSL of generators has the following characteristics that Singularity [224], an existing work on generators, does not have:

1. All values of a user-specified target size have positive probabilities of being generated;
2. The DSL admits generators of any user-defined algebraic data types.

§9.2.1 defines the syntax of generators and sizes of values. §9.2.2 then describes an operational semantics of generators. Finally, §9.2.3 presents a type system of values and generators.

9.2.1 Syntax

Algebraic data types Fix a set $\mathcal{T}$ of type names and a set C of data constructors. Generators support algebraic data types τ_{sum} defined in the following grammar:

$$\begin{aligned}
\tau_{\text{atom}} &::= \texttt{int} \mid t \in \mathcal{T} && \text{atomic types} \\
\tau_{\text{prod}} &::= \tau_{\text{atom},1} \times \cdots \times \tau_{\text{atom},k} && \text{product type; } k \in \mathbb{N} \\
\tau_{\text{sum}} &::= c_1\, \tau_{\text{prod},1} + \cdots + c_k\, \tau_{\text{prod},k} && \text{sum type; } k \in \mathbb{N}, c_1, \ldots, c_k \in C.
\end{aligned}$$

This grammar is analogous to OCaml's grammar for type definitions, where product types τ_{prod} are prepended with data constructors $c \in C$. In the following, when data constructors do not matter, I simply write $\tau_{\text{prod},1} + \cdots + \tau_{\text{prod},k}$, omitting data constructors.

The unit type `unit` can be defined as the unit of the product-type constructor, and the Boolean type can be defined as the sum type `unit + unit`.

The user provides a finite set T of type definitions of the form

$$t \coloneqq \tau_{\text{sum},t}, \tag{9.2.1}$$

where $t \in \mathcal{T}$ is a type name being defined and $\tau_{\text{sum},t}$ is the type definition of t. The type definition $\tau_{\text{sum},t}$ is allowed to mention other type names in the set T, including t itself. If a type name $t \in \mathcal{T}$ is defined in a finite set T of type definitions, I write $t \in \text{dom}(T)$.

Values Values v are formed by the following grammar:

$$v ::= z \in \mathbb{Z} \mid \text{fold}_t(c\, \langle v_1, \ldots, v_k \rangle) \qquad (k \in \mathbb{N}; c \in C; t \in \text{dom}(T)).$$

Only values of atomic types (i.e., `int` and type names $t \in \mathcal{T}$) are considered as standalone values. Sub-values such as tuples $\langle v_1, \ldots, v_k \rangle$ and tagged values $c\, v$ are never generated as standalone values.

For illustration, consider a recursive type definition for integer lists:

$$L_{\texttt{int}} \coloneqq (\texttt{nil unit}) + (\texttt{cons int} \times L_{\texttt{int}}), \tag{9.2.2}$$

strictly sorted list is not a worst-case input, as it makes fewer calls to the recursively implemented append function than a reversely sorted list when recursive results are combined at the end of each recursive call of QuickSort.

where $\texttt{nil}, \texttt{cons} \in C$ are data constructors for the empty list and non-empty lists, respectively. The empty list is encoded as

$$[\,] \coloneqq \mathsf{fold}_{L_{\mathsf{int}}}(\texttt{nil}\ \langle\rangle). \tag{9.2.3}$$

Sizes of values To run a generator, the user specifies the size of a value to generate. In contrast to the content of a value (e.g., lists), its size is fixed by the user, instead of being probabilistically generated. Otherwise, if the size as well as the content of a value were probabilistically generated by a generator, we would not be able to compare the quality of two generators fairly.

Given a value v and a type name $t \in \mathcal{T}$, the size of the value v according to the type name t is a finite multiset of natural numbers denoted by $\mathsf{size}_t(v) \subset \mathbb{N}_{\geq 1}$. The elements of $\mathsf{size}_t(v)$ each indicate the numbers of constructs $\mathsf{fold}_t(\cdot)$ that appear along different branches of the value v's AST. The size $\mathsf{size}_t(v)$ is defined as

$$\mathsf{size}_t(z \in \mathbb{Z}) \coloneqq \emptyset \qquad \mathsf{size}_t(c\ \langle v_1, \ldots, v_k\rangle) \coloneqq \biguplus_{i=1}^{k} \mathsf{size}_t(v_i) \tag{9.2.4}$$

$$\mathsf{size}_t(\mathsf{fold}_t(v)) \coloneqq \left\{1 + \sum_{s \in \mathsf{size}_t(v)} s\right\} \qquad \mathsf{size}_{t_1}(\mathsf{fold}_{t_2}(v)) \coloneqq \mathsf{size}_{t_1}(v), \tag{9.2.5}$$

where the operator $\biguplus$ in Eq (9.2.4) denotes a (disjoint) union of multisets, and Eq (9.2.5) assumes $t_1 \neq t_2$.

In the left equation of Eq (9.2.4), the size of an integer with respect to any type name $t \in \mathcal{T}$ is defined as the empty set $\emptyset$. This is because integers contain no occurrences of the construct $\mathsf{fold}_t(\cdot)$. In the right equation of Eq (9.2.4), given a tagged tuple $v \coloneqq c\ \langle v_1, \ldots, v_k\rangle$, its size with respect to a type name $t \in \mathcal{T}$ is given by the union of $\mathsf{size}_t(v_i)$ $(i = 1, \ldots, k)$. That is, if the AST of the tagged tuple v has multiple branches, the size is given by the union of the constituent branches' sizes.

Eq. (9.2.5) has two cases for a value $\mathsf{fold}_t(v)$. The first case is where the outermost construct $\mathsf{fold}_t(\cdot)$ has the same type t as the type in the size operator $\mathsf{size}_t(\cdot)$. Given a value $\mathsf{fold}_t(v)$, where $v = c\ \langle v_1, \ldots, v_k\rangle$ $(c \in C)$, its size with respect to the type name t is a singleton set whose sole element is one plus the sum of the sizes $\mathsf{size}_t(v_i)$ $(i = 1, \ldots, n)$ of constituent components in the tuple. Conversely, if we seek $\mathsf{size}_{t_t}(\mathsf{fold}_{t_2}(v))$ where ${}_1 \neq t_2$, then it is given by $\mathsf{size}_{t_1}(v)$.

For illustration, consider a nested integer list given by the type

$$L_{\mathsf{outer}} \coloneqq \texttt{unit} + L_{\mathsf{inner}} \times L_{\mathsf{outer}} \qquad L_{\mathsf{inner}} \coloneqq \texttt{unit} + \texttt{int} \times L_{\mathsf{inner}}. \tag{9.2.6}$$

Consider a nested integer list

$$v \coloneqq [[z_{1,1}, \ldots, z_{1,n_1}], \ldots, [z_{k,1}, \ldots, z_{k,n_k}]] \qquad (k \in \mathbb{N}; n_1, \ldots, n_k \in \mathbb{N}), \tag{9.2.7}$$

where k is the outer list length, and n_i $(i = 1, \ldots, k)$ is the length of the i^{th} inner list. We have

$$\mathsf{size}_{L_{\mathsf{outer}}}(v) = 1 + k \qquad \mathsf{size}_{L_{\mathsf{inner}}}(v) = \{1 + n_1, \ldots, 1 + n_k\}. \tag{9.2.8}$$

The outer list size $\text{size}_{L_{\text{outer}}}(v)$ is $1+k$, instead of just k, because the empty list (Eq (9.2.3)) already has size 1. Additionally, the size of the nested list v with respect to the inner list type L_{inner} is a multiset $\{1 + n_1, \ldots, 1 + n_k\}$, whose elements are the inner lists' sizes.

To run a generator, the user specifies target sizes for all relevant types names. In the above example of nested integer lists (Eq (9.2.6)), the user specifies two target sizes $s_{\text{outer}}, s_{\text{inner}} \in \mathbb{N}$, which are natural numbers instead of multisets. Let v denote a nested list to be generated. The target size s_{outer} is the outer list size $\text{size}_{L_{\text{outer}}}(v)$, and the target size s_{inner} is the size of *all* inner lists in the nested list v.

The current implementation of generators cannot generate those values whose sizes cannot be specified by single numbers (e.g., a nested list whose inner lists have different lengths). If the user seeks greater flexibility in specifying the size and shape of a value to be generated, the implementation of generators can be modified such that target sizes are lists, instead of single numbers. For instance, for a nested list, if the user would like the inner lists to be of different sizes, they can provide a list of numbers to be used as inner lists' sizes. However, for simplicity, the user-specified target sizes are restricted to numbers, as opposed to lists.

Distributions for tagged values and tuples To generate inhabitants of algebraic data types, generators make two decisions:

1. Which data constructor $c \in C$ to choose for a tagged value $c\,v$ of a sum type;
2. How to split a target size $s \in \mathbb{N}$ of a tuple among its components.

To make these decisions probabilistically, a generator is equipped with two categorical distributions, d_{sum} and d_{prod}. They are both formed by the following grammar:

$$d ::= (p_1, \ldots, p_k) \qquad (p_1, \ldots, p_k \in [0, 1]; \textstyle\sum_i p_i = 1).$$

For the first decision, given a categorical distribution d_{sum}, a data constructor $c_i \in C$ $(i = 1, \ldots, n)$ is selected with probability $p_i \in [0, 1]$ to create a tagged value $c_i\,v$. For the second decision, for each type name $t \in \text{dom}(T)$, its target size $s_t \in \mathbb{N}$ is split among a tuple's components according to a multinomial distribution parametrized by $s_t \in \mathbb{N}$ and a categorical distribution d_{prod}. Multinomial distributions generalize binomial distributions and have two parameters: $s \in \mathbb{N}$ (i.e., the target size) and $(p_1, \ldots, p_k)$ (i.e., a categorical distribution d_{prod}). Given $s \in \mathbb{N}$ many objects, each of them is randomly assigned the i^{th} class (i.e., the i^{th} component of a tuple) with probability $p_i \in [0, 1]$. The size assigned to the i^{th} component is the number of objects assigned to the i^{th} component.

Expressions Fix a countable set $\mathcal{X}$ of program variables. Expressions e are formed by the grammar

$$\begin{array}{lll}
e ::= x \in \mathcal{X} \mid z \in \mathbb{Z} \mid \text{succ}(e) \mid \text{pred}(e) & & \text{variable and arithmetic expressions} \\
\quad \mid \text{fold}_t(c\,\langle e_1, \ldots, e_k\rangle) & & \text{fold}; c \in C; t \in \mathcal{T} \\
\quad \mid \text{proj}_\tau^k(e) & & \text{projection}; \tau \in \{\texttt{int}\} \cup \mathcal{T}; k \in \mathbb{N}_{\geq 1} \\
\quad \mid \text{size}_t(e) & & \text{size}; t \in \mathcal{T}.
\end{array}$$

The projection operator $\mathrm{proj}_{\tau}^{k}(e)$ ($t \in \{\texttt{int}\} \cup \mathcal{T}$ and $k \in \mathbb{N}_{\geq 1}$) first evaluates the expression e. If its value has the form $\mathrm{fold}_t(\langle v_1, \ldots, v_i \rangle)$ with $i \geq k$ and the k^{th} component v_k has the type τ, the projection operator returns v_k. Otherwise, if the evaluation result of the expression e does not have enough components or its k^{th} component has a type different from $\tau \in \{\texttt{int}\} \cup \mathcal{T}$, a random value of type τ is generated and returned.

The size operator $\mathrm{size}_t(e)$ evaluates the expression e and returns a size $s \in \mathbb{N}$, where $\{s\} = \mathrm{size}_t(v)$ and v is the expression e's value. When expressions e are generated during an optimization of generators, the size operator $\mathrm{size}_t(\cdot)$ is only applied to an expression e that has the type t. Consequently, when the expression e is evaluated to a value v, the size $\mathrm{size}_t(v)$ is guaranteed to be a singleton multiset.

Generators for integers A generator g generating integers is defined as

$$\mathrm{gene}\, g\, \langle x_1, \ldots, x_m \rangle = (e_{\text{lower}}, e_{\text{upper}}, e_{\ell}), \tag{9.2.9}$$

where $x_1, \ldots, x_m$ are input variables and $e_{\text{lower}}, e_{\text{lower}}, e_{\ell}$ are expressions. The expressions e_{lower} and e_{lower} encode lower and upper bounds, respectively, of a uniform distribution to sample from. The expression e_{ℓ} encodes a list of integers[3]that the generator should not sample. The expressions e_{lower} and e_{upper} can be standard expressions e (defined above) that mention the parameters $x_1, \ldots, x_m$ of the generator. Alternatively, the expressions e_{lower} and e_{upper} can be a fixed minium bound $z_{\min} \in \mathbb{Z}$ and a fixed maximum bound $z_{\max} \in \mathbb{Z}$, respectively. The two constants $z_{\min}, z_{\max} \in \mathbb{Z}$ are user-tunable hyperparameters, and a prototype implementation sets $z_{\min} := -500$ and $z_{\max} := 500$.

The generator g works as follows. It first samples a Boolean variable $b \in \{0, 1\}$ from a Bernoulli distribution:

$$b \sim \mathrm{Bernoulli}(p), \tag{9.2.10}$$

where $p \in (0, 1)$ is a user-tunable hyperparameter and is set to 0.985 in a prototype implementation. If $b = 1$ (with probability p), the generator g goes on to draw an integer uniformly at random from a set:

$$z \sim \mathrm{Uniform}([e_{\text{lower}}, e_{\text{upper}}] \setminus e_{\ell}), \tag{9.2.11}$$

where $[e_{\text{lower}}, e_{\text{upper}}] \setminus e_{\ell}$ denotes the set of all integers in the interval $[e_{\text{lower}}, e_{\text{upper}}]$, excluding those in the list e_{ℓ}. The sampling of integers from the set (9.2.11) is only well-defined if the set is non-empty. Otherwise, if $b = 0$ (with probability $1-p$) or $[e_{\text{lower}}, e_{\text{upper}}] \setminus x = \emptyset$, the generator g resorts to a default generator, drawing an integer uniformly at random from a broad interval (e.g., $[-2^{10}, 2^{10} - 1]$ in a prototype implementation).

Generators for type names Given a type name $t \in \mathcal{T}$ of a value to be generated, suppose the definition of the type t has the form

$$t := c_1\, \tau_1 + \cdots + c_k\, \tau_k \qquad \tau_i := \tau_{i,1} \times \cdots \times \tau_{i,n_i} \qquad (i = 1, \ldots, k). \tag{9.2.12}$$

[3]Since the component e_{ℓ} inside the generator (9.2.9) is a list of integers, whenever integers (or compound values involving integers) are generated, the integer-list type $L_{\texttt{int}}$ (Eq (9.2.2)) should be defined. This is required even when integer lists are not generated anywhere.

A generator g for the type t has the same inductive structure as the type definition (9.2.12). The generator g has parameters $x_1, \ldots, x_m$. For a product type $\tau_i := \tau_{i,1} \times \cdots \times \tau_{i,n_i}$ $(i = 1, \ldots, k)$, the generator g evaluates an expression $e_{i,j}$ and assigns its value $v_{i,j}$ to the j^{th} component of a tuple $(j = 1, \ldots, n_i)$. The expression $e_{i,j}$ $(i = 1, \ldots, k$ and $j = 1, \ldots, n_i)$ can call other generators and mention the following variables: (i) parameters $x_1, \ldots, x_m$ of the generator g; and (ii) values $v_{i,1}, \ldots, v_{i,j-1}$ that have been generated for the preceding components of a tuple. If the i^{th} component of the sum type t is chosen, the generator g evaluates all $e_{i,j}$ for $j = 1, \ldots, n_i$, yielding a tagged value $c_i \langle v_{i,1}, \ldots, v_{i,n_i} \rangle$.

Formally, generators for type names are defined as follows. Fix a countable set $\mathcal{G}$ of generator identifiers. Code blocks h are formed by the grammar

$$h ::= \mathsf{let}\ x_1 = g_1 \langle e_{1,1}, \ldots, e_{1,n_1} \rangle\ \mathsf{in}\ \ldots\ \mathsf{let}\ x_k = g_k \langle e_{k,1}, \ldots, e_{k,n_k} \rangle\ \mathsf{in}\ \mathsf{fold}_t(c \langle x_1, \ldots, x_k \rangle), \tag{9.2.13}$$

where $g_1, \ldots, g_k \in \mathcal{G}$ are generator identifiers. The code block h generates all components of a tuple, tags it with a data constructor, and encloses it inside $\mathsf{fold}_t(\cdot)$.

A generator identifier $g \in \mathcal{G}$ for the type name t (Eq (9.2.12)) is defined as

$$\mathsf{gene}\, g \langle x_1, \ldots, x_m \rangle = (t, (d^{\text{base}}_{\text{sum}}, d^{\text{rec}}_{\text{sum}}), (d^{i,t}_{\text{prod}}; i = 1, \ldots, k, t \in \mathsf{dom}(T)), (h_i; i = 1, \ldots, k)), \tag{9.2.14}$$

where $x_1, \ldots, x_m$ are input variables of the generator g. The right-hand side of Eq (9.2.14) contains three components: (i) the target type name $t \in \mathcal{T}$; (ii) a pair of $(d^{\text{base}}_{\text{sum}}, d^{\text{rec}}_{\text{sum}})$ of categorical distributions; (iii) a tuple $(d^{i,t}_{\text{prod}}; i = 1, \ldots, k, t \in \mathsf{dom}(T))$ of categorical distributions; and (iv) a tuple $(h_i; i = 1, \ldots, k)$ of code blocks. Here, T is a finite set of type definitions.

The pair $(d^{\text{base}}_{\text{sum}}, d^{\text{rec}}_{\text{sum}})$ of categorical distributions in Eq (9.2.14) is used to determine which data constructor c_i $(i = 1, \ldots, k)$ is chosen during the execution of the generator. The distribution $d^{\text{base}}_{\text{sum}}$ is used when the target size s_t is one; otherwise, the distribution $d^{\text{base}}_{\text{sum}}$ is used. It is crucial to use different distributions for these two cases because when $s_t = 1$, the generator g should not choose a data constructor c_i $(i = 1, \ldots, k)$ that always leads to $\mathsf{fold}_t(\cdot)$. For example, for the integer-list type $L_{\mathtt{int}}$ (Eq (9.2.2)), if the target size is $s_{L_{\mathtt{int}}} = 0$, the generator should not choose the second branch of the sum type $L_{\mathtt{int}}$, which would lead to $\mathsf{fold}_{L_{\mathtt{int}}}(\cdot)$. Dually, if $s_t > 1$, the generator should not choose a data constructor that never leads to $\mathsf{fold}_t(\cdot)$. Thus, the two distributions $d^{\text{base}}_{\text{sum}}$ and $d^{\text{rec}}_{\text{sum}}$ may need to have different supports (i.e., the set of values with positive probabilities).

The tuple $(d^{i,t}_{\text{prod}}; i = 1, \ldots, k, t \in \mathsf{dom}(T))$ of categorical distributions in Eq (9.2.14) is used to split user-specified target sizes among components of tuples. Suppose that the generator decides to generate a tuple of the product type τ_i $(i = 1, \ldots, k)$. For each type name $t \in \mathsf{dom}(T)$, if the generator g has not generated $\mathsf{fold}_t(\cdot)$ yet (i.e., the value being currently generated is not wrapped inside $\mathsf{fold}_t(\cdot)$), the target size s_t is *copied* to each component of a tuple. Otherwise, if the generator has generated $\mathsf{fold}_t(\cdot)$, the target size s_t is *split* among the components $\tau_{i,1}, \ldots, \tau_{i,n_i}$ of the product type according to a multinomial distribution parametrized by s_t and $d^{i,t}_{\text{prod}}$.

For illustration, consider a nested integer list (Eq (9.2.6)). Let $s_{L_{\text{inner}}}, s_{L_{\text{inner}}} \in \mathbb{N}$ be target sizes of an outer list and inner lists, respectively. During the execution of a generator g, it recursively generates a chain of $s_{L_{\text{inner}}} \in \mathbb{N}$ many constructs $\mathsf{fold}_{L_{\text{outer}}}(\cdot)$, and this chain forms an outer list. In each recursive call, if $s_{L_{\text{outer}}} \geq 1$, this target size of the outer list is decremented by one.

$$\text{E:Var}\ \frac{V(x) = v}{V \vdash_T x \Downarrow v} \qquad \text{E:Int}\ \frac{z \in \mathbb{Z}}{V \vdash_T z \Downarrow z} \qquad \text{E:Succ}\ \frac{V \vdash_T e \Downarrow z}{V \vdash_T \mathsf{succ}(e) \Downarrow (z+1)} \qquad \text{E:Pred}\ \frac{V \vdash_T e \Downarrow z}{V \vdash_T \mathsf{pred}(e) \Downarrow (z-1)}$$

$$\text{E:Fold}\ \frac{\forall i = 1, \ldots, k. (V \vdash_T e_i \Downarrow v_i)}{V \vdash_T \mathsf{fold}_t(c\,\langle e_1, \ldots, e_k\rangle) \Downarrow \mathsf{fold}_t(c\,\langle v_1, \ldots, v_k\rangle)} \qquad \text{E:Proj}\ \frac{V \vdash_T e \Downarrow \mathsf{fold}_t(c\,\langle v_1, \ldots, v_n\rangle) \qquad n \geq k \qquad v_k :_T \tau}{V \vdash_T \mathsf{proj}^k_\tau(e) \Downarrow v_k}$$

$$\text{E:Proj:Fail:1}\ \frac{V \vdash_T e \Downarrow \mathsf{fold}_t(c\,\langle v_1, \ldots, v_n\rangle) \qquad n < k \qquad v = \mathsf{value}(\tau)}{V \vdash_T \mathsf{proj}^k_\tau(e) \Downarrow v}$$

$$\text{E:Proj:Fail:2}\ \frac{V \vdash_T e \Downarrow \mathsf{fold}_t(c\,\langle v_1, \ldots, v_n\rangle) \qquad n \geq k \qquad \neg(v_k : \tau) \qquad v = \mathsf{value}(\tau)}{V \vdash_T \mathsf{proj}^k_\tau(e) \Downarrow v}$$

$$\text{E:Size}\ \frac{V \vdash_T e \Downarrow \mathsf{fold}_t(v) \qquad \mathsf{size}_t(\mathsf{fold}_t(v)) = \{z\}}{V \vdash_T \mathsf{size}_t(e) \Downarrow z} \qquad \text{E:Size:Fail}\ \frac{V \vdash_T e \Downarrow v \qquad v \not\equiv \mathsf{fold}_t(\_) \qquad z = \mathsf{value}(\texttt{int})}{V \vdash_T \mathsf{size}_t(e) \Downarrow z}$$

Lst. 9.1: Operational semantics of expressions e.

Otherwise, if $s_{L_{\text{outer}}} = 0$ and hence cannot be decremented, the generator raises an exception. In this (outer) recursion, the generator has not generated $\mathsf{fold}_{L_{\text{inner}}}(\cdot)$ yet, so the target size $s_{L_{\text{inner}}}$ for inner lists is copied to each node of the outer list. To generate individual inner lists, the generator runs an inner recursion, recursively generating a chain of constructs $\mathsf{fold}_{L_{\text{inner}}}(\cdot)$. Each recursive call first decrements the target size $s_{L_{\text{inner}}}$ by one and then splits the remaining size between the two components of an inner list, namely the head and the tail, according to a multinomial distribution.

Generator programs A generator program G in the DSL of generators is a finite set of (possibly mutually recursive) generator definitions, each of which has either the form (9.2.9) (if the target type is `int`) or the form (9.2.14) (if the target type is a type name $t \in \mathcal{T}$). In addition, the generator program[4]defines a main generator $g_{\text{main}} \in G$ to run.

9.2.2 Operational Semantics

Expressions Operational semantics of expressions is defined by a judgment

$$V \vdash_T e \Downarrow v, \tag{9.2.15}$$

[4]In the following, whenever there is no risk of confusion between a target program P for data-driven resource analysis and a generator program G that generates program inputs for the target program P, I refer to the latter simply as a program.

E:Gen:TypeName

$$\frac{\begin{array}{c}(\text{gene } g\ \langle x_1, \ldots, x_m\rangle = (t_g, (d^{\text{base}}_{\text{sum}}, d^{\text{rec}}_{\text{sum}}), (d^{i,t}_{\text{prod}}; i = 1, \ldots, k, t \in \text{dom}(T)), (h_i; i = 1, \ldots, k))) \in G \\ j = \text{choose}((d^{\text{base}}_{\text{sum}}, d^{\text{rec}}_{\text{sum}}), S(t_g)) \qquad (S_1, \ldots, S_k) = \text{split}((d^{j,t}_{\text{prod}}; t \in \text{dom}(T)), S[t_g \mapsto S(t_g) - 1], \pi) \\ \forall i = 1, \ldots, m.(V \vdash_T e_i \Downarrow v_i) \qquad \{x_i \mapsto v_i \mid i = 1, \ldots, m\}; (S_1, \ldots, S_k); \pi \cup \{t_g\} \vdash_{G,T} h_j \Downarrow v\end{array}}{V; S; \pi \vdash_{G,T} g\ \langle e_1, \ldots, e_m\rangle \Downarrow v}$$

E:CodeBlock

$$\frac{\begin{array}{c}h \equiv \text{let } x_1 = g_1\ \langle e_{1,1}, \ldots, e_{1,n_1}\rangle \text{ in } \ldots \text{ let } x_k = g_k\ \langle e_{k,1}, \ldots, e_{k,n_k}\rangle \text{ in fold}_t(c\ \langle x_1, \ldots, x_k\rangle) \\ \forall i = 1, \ldots, k.(V \cup \{y_1 \mapsto v_1, \ldots, y_{i-1} \mapsto v_{i-1}\}; S_i; \pi \vdash_{G,T} g_i\ \langle e_{i,1} \ldots e_{i,n_i}\rangle \Downarrow v_i)\end{array}}{V; (S_1, \ldots, S_k); \pi \vdash_{G,T} h \Downarrow \text{fold}_t(c)\langle v_1, \ldots, v_k\rangle}$$

E:Gen:Int

$$\frac{\begin{array}{c}(\text{gene } g\ \langle x_1, \ldots, x_m\rangle = (e_{\text{lower}}, e_{\text{upper}}, e_\ell)) \in G \qquad b \sim \text{Bernoulli}(p) \qquad b = 1 \qquad V \vdash_T e_{\text{lower}} \Downarrow v_{\text{lower}} \\ V \vdash_T e_{\text{upper}} \Downarrow v_{\text{upper}} \qquad V \vdash_T e_\ell \Downarrow v_\ell \qquad [v_{\text{lower}}, v_{\text{upper}}] \setminus v_\ell \neq \emptyset \qquad z \sim \text{Uniform}([v_{\text{lower}}, v_{\text{upper}}] \setminus v_\ell)\end{array}}{V; S; \pi \vdash_{G,T} g\ \langle e_1, \ldots, e_m\rangle \Downarrow z}$$

E:Gen:Int:Fail:1

$$\frac{\begin{array}{c}(\text{gene } g\ \langle x_1, \ldots, x_m\rangle = (e_{\text{lower}}, e_{\text{upper}}, e_\ell)) \in G \qquad b \sim \text{Bernoulli}(p) \qquad b = 1 \\ V \vdash_T e_{\text{lower}} \Downarrow v_{\text{lower}} \qquad V \vdash_T e_{\text{upper}} \Downarrow v_{\text{upper}} \qquad V \vdash_T e_\ell \Downarrow v_\ell \qquad [v_{\text{lower}}, v_{\text{upper}}] \setminus v_\ell = \emptyset \qquad z = \text{value}(\texttt{int})\end{array}}{V; S; \pi \vdash_{G,T} g\ \langle e_1, \ldots, e_m\rangle \Downarrow z}$$

E:Gen:Int:Fail:2

$$\frac{(\text{gene } g\ \langle x_1, \ldots, x_m\rangle = (e_{\text{lower}}, e_{\text{upper}}, e_\ell)) \in G \qquad b \sim \text{Bernoulli}(p) \qquad b = 0 \qquad z = \text{value}(\texttt{int})}{V; S; \pi \vdash_{G,T} g\ \langle e_1, \ldots, e_m\rangle \Downarrow z}$$

Lst. 9.2: Operational semantics of generators g and code blocks h.

where V is an environment (i.e., a mapping from variables to values), T is a set of type definitions, e is an expression, and v is a value. The judgment (9.2.15) states that, under an environment V and a set T of type definitions, the expression e evaluates to a value v.

Listing 9.1 defines the judgment (9.2.15). The underscore in the rule E:Size:Fail means it does not matter what goes in here.

The operators $\text{proj}^k_\tau(\cdot)$ and $\text{size}_t(\cdot)$ can fail. For the projection operator $\text{proj}^k_\tau(e)$, the rule E:Proj is only applicable when the expression e's value has at least $k \in \mathbb{N}$ components (the second premise) and the k^{th} component can be typed with $\tau \in \{\texttt{int}\} \cup \mathbb{T}$ (the third premise). Otherwise, the rules E:Proj:Fail:1 and E:Proj:Fail:2 apply, where the former is used when the expression e's value does not have enough components, and the latter is used when the k^{th} component is not typable with τ. In the rule E:Proj:Fail:2, to type-check the value v with the type τ, the set T of type definitions is necessary (Listing 9.3). In both E:Proj:Fail:1 and E:Proj:Fail:2, the premise $v = \text{value}(\tau)$ generates a (random) value of type τ and returns it.

Given the size operator $\text{size}_t(e)$, the rule E:Size is only applicable when the expression e evaluates to $\text{fold}_t(v)$ for some value v. Otherwise, the rule E:Size:Fail applies, generating a (random) integer $v = \text{value}(\texttt{int})$.

Generators for type names Operational semantics of generators is given by a judgment

$$V; S; \pi \vdash_{G,T} g\,\langle e_1, \ldots, e_m \rangle \Downarrow v, \tag{9.2.16}$$

where V is an environment, S is a mapping from type names to their target sizes, π is a set of type names $t \in \mathcal{T}$, G is a program of generators (i.e., a set of generator definitions), and T is a set of type definitions, $g \in G$ is a generator, $e_1, \ldots, e_m$ are expressions, and v is a value. The set π records type names that the generator has seen so far and is initialized to the empty set $\emptyset$. The judgment (9.2.16) states that, under an environment V, mapping S, and set π, the generator $g \in G$ with arguments $e_1, \ldots, e_m$ generates a value v. The judgment (9.2.16) does not indicate the probability of generating a value v.

Listing 9.2 defines the judgment (9.2.16). In the rule E:Gen:TypeName, the last premise contains an evaluation judgment for a code block h:

$$V; (S_1, \ldots, S_k); \pi \vdash_{G,T} h \Downarrow v. \tag{9.2.17}$$

This judgment is defined by the rule E:CodeBlock (Listing 9.2).

The rule E:Gen:TypeName concerns a generator application $g\,\langle e_1, \ldots, e_m \rangle$, where g is a generator for a type name, instead of `int`. Suppose the first premise: the generator g is defined as E:Gen:TypeName:

$$\mathsf{gene}\, g\,\langle x_1, \ldots, x_m \rangle = (t_g, (d^{\mathrm{base}}_{\mathrm{sum}}, d^{\mathrm{rec}}_{\mathrm{sum}}), (d^{i,t}_{\mathrm{prod}}; i = 1, \ldots, k, t \in \mathsf{dom}(T)), (h_i; i = 1, \ldots, k)). \tag{9.2.18}$$

Assume that the type name $t_g \in \mathcal{T}$ is defined as

$$t_g \coloneqq \tau_1 + \cdots + \tau_n \qquad \tau_i \coloneqq c_{i,1} \times \cdots \times c_{i,n_i} \qquad (i = 1, \ldots, n). \tag{9.2.19}$$

The generator first probabilistically chooses a branch $j \in \{1, \ldots, n\}$ in a sum type by drawing from a categorical distribution (the second premise):

$$j = \mathsf{choose}((d^{\mathrm{base}}_{\mathrm{sum}}, d^{\mathrm{rec}}_{\mathrm{sum}}), S(t_g)). \tag{9.2.20}$$

Here, $S(t_g)$ returns a numeric size of the type name $t_g \in \mathcal{T}$, which is the target type name of the generator g. Given two categorical distributions $d^{\mathrm{base}}, d^{\mathrm{rec}}$ and a target size $s \in \mathbb{N}$, the operator $\mathsf{choose}((d^{\mathrm{base}}, d^{\mathrm{rec}}), s)$ is defined as

$$\mathsf{choose}((d^{\mathrm{base}}, d^{\mathrm{rec}}), s) \coloneqq \begin{cases} j \sim d^{\mathrm{base}} & \text{if } s = 1 \\ j \sim d^{\mathrm{rec}} & \text{if } s > 1, \end{cases} \tag{9.2.21}$$

where the notation $j \sim d$ means $j \in \mathbb{N}_{\geq 1}$ is drawn from a categorical distribution d.

Next, the generator g decides how to split the target-size mapping S among the $k \coloneqq n_j$ components of the chosen product type τ_j. The third premise of the rule E:Gen:TypeName states

$$(S_1, \ldots, S_k) = \mathsf{split}((d^{j,t}_{\mathrm{prod}}; t \in \mathsf{dom}(T)), S[t_g \mapsto S(t_g) - 1], \pi), \tag{9.2.22}$$

where the tuple $(S_1, \ldots, S_k)$ is the result of splitting the mapping S. The mapping $S[t_g \mapsto S(t_g) - 1]$ is obtained by subtracting 1 from the target size of $t_g \in \mathcal{T}$ stored in the mapping S,

provided that $S(t_g) \geq 1$. Otherwise, if $S(t_g) = 0$, the generator raises an exception because it cannot generate a value of the target size $S(t_g)$. The output of $\mathsf{split}((d^t; t \in \mathsf{dom}(T)), S, \pi)$ is defined as

$$(S_1(t), \ldots, S_k(t)) \coloneqq \begin{cases} \mathbf{I} + \mathbf{s} & \text{if } t \in \pi \\ (S(t), \ldots, S(t)) & \text{otherwise.} \end{cases} \qquad (t \in \mathsf{dom}(T)) \tag{9.2.23}$$

The tuple $\mathbf{I}$ in Eq (9.2.23) is defined as

$$\mathbf{I} \coloneqq (r_1, \ldots, r_k) \qquad r_i \coloneqq \begin{cases} 1 & \text{if } \tau_i \xrightarrow[T]{\text{always}} t \\ 0 & \text{otherwise,} \end{cases} \qquad (i = 1, \ldots, k) \tag{9.2.24}$$

where the relation $\tau_i \xrightarrow[T]{\text{always}} t$ means that the type τ_i can always reach the type name $t \in \mathcal{T}$ regardless of which path is taken. The relation is defined in Listing 9.6. The tuple $\mathbf{s}$ in Eq (9.2.23) is a random tuple of size k drawn from a multinomial distribution:

$$\mathbf{s} \sim \text{Multinomial}(S(t) - \mathbf{I}, d^t). \tag{9.2.25}$$

If $\tau_i \xrightarrow[T]{\text{always}} t$ holds, the construct $\text{fold}_t(\cdot)$ is guaranteed to appear inside the i^{th} component of the tuple. Thus, the size $S_i(t)$ should be at least one to account for the construct $\text{fold}_t(\cdot)$. To ensure $S_i(t) \geq 1$ whenever $\tau_i \xrightarrow[T]{\text{always}} t$ holds ($i = 1, \ldots, k$), we first subtract the tuple $\mathbf{I}$ from the tuple of target sizes $S(t)$ (Eq (9.2.24)), draw a sample $\mathbf{s}$ from a multinomial distribution (Eq (9.2.25)), and then add the tuple $\mathbf{I}$ back to the result (Eq (9.2.23)).

If the set π of type names contains the type name $t \in \mathcal{T}$, it means the generator has seen the type t (i.e., the value currently being generated is wrapped inside $\text{fold}_t(\cdot)$). In this case, the target size $S(t)$ is split according to a multinomial distribution parametrized by $S(t)$ and d^t. Otherwise, if $t \notin \pi$, the target size s_t is copied to all k components.

Finally, the generator g evaluates each argument e_i ($i = 1, \ldots, m$) (the fourth premise in the rule E:Gen:TypeName) and evaluates the code block h_j corresponding to the chosen branch j (the last premise). The set π of type names that the generator g has seen so far is extended with the target type t_g.

To evaluate a code block, the rule E:CodeBlock comes into play, where a generator application $g_i \langle e_{i,1} \ldots e_{i,n_i} \rangle$ ($i = 1, \ldots, k$) (the second premise) can access not only the parameters $x_1, \ldots, x_m$ of the generator g but also the variables $y_1, \ldots, y_{i-1}$. These variables are bound to the values $v_1, \ldots, v_{i-1}$ of the preceding components of a tuple that have been generated. Finally, the generated values v_i ($i = 1, \ldots, k$) are aggregated into a final output $\text{fold}_t(c \langle v_1, \ldots, v_k \rangle)$.

Generators for integers A generator g first draws a Boolean random variable $b \in \{0, 1\}$ from a Bernoulli distribution (the second premise):

$$b \sim \text{Bernoulli}(p), \tag{9.2.26}$$

$$\textsc{V:Int}\ \frac{z \in \mathbb{Z}}{z :_T \texttt{int}} \qquad \textsc{V:Fold}\ \frac{(t \coloneqq \cdots + c\,\tau + \cdots) \in T \qquad \tau \coloneqq \tau_1 \times \cdots \times \tau_k \qquad \forall i \in \{1, \ldots, k\}.(v_i :_T \tau_i)}{\mathsf{fold}_t(c\,\langle v_1, \ldots, v_k \rangle) :_T t}$$

Lst. 9.3: Well-typed values v.

$$\textsc{T:Var}\ \frac{\Gamma(x) = \tau}{\Gamma \vdash_T x : \tau} \qquad \textsc{T:Int}\ \frac{z \in \mathbb{Z}}{\Gamma \vdash_T z : \texttt{int}} \qquad \textsc{T:Succ}\ \frac{\Gamma \vdash_T e : \texttt{int}}{\Gamma \vdash_T \mathsf{succ}(e) : \texttt{int}} \qquad \textsc{T:Pred}\ \frac{\Gamma \vdash_T e : \texttt{int}}{\Gamma \vdash_T \mathsf{pred}(e) : \texttt{int}}$$

$$\textsc{T:Fold}\ \frac{(t \coloneqq \cdots + c\,\tau + \cdots) \in T \qquad \tau \coloneqq \tau_1 \times \cdots \times \tau_k \qquad \forall i \in \{1, \ldots, k\}.(\Gamma \vdash_T e_i : \tau_i)}{\Gamma \vdash_T \mathsf{fold}_t(c\,\langle e_1, \ldots, e_k \rangle) : t} \qquad \textsc{T:Proj}\ \frac{}{\Gamma \vdash_T \mathsf{proj}_\tau^k(e) : \tau}$$

$$\textsc{T:Size}\ \frac{t \in \mathsf{dom}(T) \qquad \Gamma \vdash_T e : t}{\Gamma \vdash_T \mathsf{size}_t(e) : \texttt{int}}$$

Lst. 9.4: Type system of expressions e.

where $p \in (0, 1)$ is a user-tunable hyperparameter and is set to 0.985 in a prototype implementation. If $b = 1$, the generator g computes the set $[e_{\text{lower}}, e_{\text{upper}}] \setminus e_\ell$. If this set is non-empty, the rule E:Gen:Int (Listing 9.2) applies, drawing an integer uniformly at random from the set:

$$z \sim \text{Uniform}([v_{\text{lower}}, v_{\text{upper}}] \setminus v_\ell). \tag{9.2.27}$$

Otherwise, if the set is empty (e.g., $e_{\text{lower}} > e_{\text{upper}}$), the rule E:Gen:Int:Fail:1 applies, generating a (random) integer, which is indicated by the last premise $v = \mathsf{value}(\texttt{int})$. Also, if $b = 0$, the rule E:Gen:Int:Fail:2 applies, also generating a (random) integer. Because $0 < p < 1$, the generator can generate any integer (ideally from some broad interval specified by the user) with a positive probability at least $1 - p$.

9.2.3 Type System

A typing judgment of values is

$$v :_T \tau, \tag{9.2.28}$$

where v is a value, T is a set of type definitions, and $\tau \in \{\texttt{int}\} \cup \mathcal{T}$ is a type. Listing 9.3 defines the typing judgment (9.2.28).

A typing judgment of expressions is

$$\Gamma \vdash_T e : \tau, \tag{9.2.29}$$

where Γ is a typing context (i.e., a mapping from variables to their types), T is a set of type definitions, e is an expression, and $\tau \in \{\texttt{int}\} \cup \mathcal{T}$ is a type.

T:Gen:TypeName

$$\frac{(t_g \coloneqq \tau_1 + \cdots + \tau_k) \in T \qquad \forall i \in \{1,\ldots,k\}.\tau_i \coloneqq \tau_{i,1} \times \cdots \times \tau_{i,n_i} \qquad |d^{\text{base}}_{\text{sum}}| = |d^{\text{rec}}_{\text{sum}}| = k \qquad \forall i \in \{1,\ldots,k\}, t \in \text{dom}(T).|d^{i,t}_{\text{prod}}| = n_i \qquad \forall i \in \{1,\ldots,k\}.(\{x_i : \tau_i \mid i = 1,\ldots,m\} \vdash_{\Sigma,T} h_i : t_g)}{\cdot \vdash_{\Sigma,T} (\text{gene}\, g\, \langle x_1,\ldots,x_m\rangle = (t_g, (d^{\text{base}}_{\text{sum}}, d^{\text{rec}}_{\text{sum}}), (d^{i,t}_{\text{prod}}; i = 1,\ldots,k, t \in \text{dom}(T)), (h_i; i = 1,\ldots,k))) : \tau_1 \times \cdots \times \tau_m \to t_g}$$

T:CodeBlock

$$\frac{(t \coloneqq \cdots + c\,\tau + \cdots) \in T \qquad \tau \coloneqq \tau_1 \times \cdots \times \tau_k \qquad h \equiv \text{let } x_1 = g_1\,\langle e_{1,1},\ldots,e_{1,n_1}\rangle \text{ in } \ldots \text{ let } x_k = g_k\,\langle e_{k,1},\ldots,e_{k,n_k}\rangle \text{ in } \text{fold}_t(c\,\langle x_1,\ldots,x_k\rangle) \qquad \forall i \in \{1,\ldots,k\}.\Gamma \cup \{y_1 : \tau_1,\ldots,y_{i-1} : \tau_{i-1}\} \vdash_{\Sigma,T} g_i\,\langle e_{i,1},\ldots,e_{i,n_i}\rangle : \tau_i}{\Gamma \vdash_{\Sigma,T} h : t}$$

T:Gen:App

$$\frac{\Sigma(g) = \tau_1 \times \cdots \times \tau_m \to \tau \qquad \forall i \in \{1,\ldots,m\}.(\Gamma \vdash_T e_i : \tau_i)}{\Gamma \vdash_{\Sigma,T} g\,\langle e_1,\ldots,e_m\rangle : \tau}$$

T:Gen:Int

$$\frac{\Gamma = \{x_i : \tau_i \mid i = 1,\ldots,m\} \qquad \Gamma \vdash_T e_{\text{lower}} : \texttt{int} \qquad \Gamma \vdash_T e_{\text{upper}} : \texttt{int} \qquad \Gamma \vdash_T e_\ell : L_{\texttt{int}}}{\cdot \vdash_{\Sigma,T} (\text{gene}\, g\, \langle x_1,\ldots,x_m\rangle = (e_{\text{lower}}, e_{\text{upper}}, e_\ell)) : \tau_1 \times \cdots \times \tau_m \to \texttt{int}}$$

Lst. 9.5: Type system of generators and code blocks.

Listing 9.4 defines the typing judgment (9.2.29). The rule T:Proj does not have a premise—the expression $\text{proj}^k_\tau(e)$ is always well-typed. If the expression e does not have enough components for projection or the k^{th} component of e's value is not of type $t \in \mathcal{T}$, then a random value of type t is generated by the rules E:Proj:Fail:1 and E:Proj:Fail:2 in the operational semantics (Listing 9.1).

A typing judgment of generators is

$$\Gamma \vdash_{\Sigma,T} g : \tau_1 \times \cdots \times \tau_m \to \tau, \tag{9.2.30}$$

where Γ is a typing context, Σ is a signature (i.e., a mapping from generator identifiers to their types), T is a set of type definitions, g is a generator, and $\tau_i \in \{\texttt{int}\} \cup \mathcal{T}$ $(i = 1,\ldots,m)$ are input types, and $\tau \in \{\texttt{int}\} \cup \mathcal{T}$ is an output type. In order for a program G (i.e., a finite set of generator definitions) to be well-typed, each generator definition $\text{gene}\, g\, \langle\cdots\rangle = \cdots$ must be well-typed according to the type signature $\Sigma(g)$.

Listing 9.5 defines the typing judgment (9.2.30). The rule T:Gen:TypeName concerns a generator definition

$$\text{gene}\, g\, \langle x_1,\ldots,x_m\rangle = (t_g, (d^{\text{base}}_{\text{sum}}, d^{\text{rec}}_{\text{sum}}), (d^{i,t}_{\text{prod}}; i = 1,\ldots,k, t \in \text{dom}(T)), (h_i; i = 1,\ldots,k)), \tag{9.2.31}$$

which intends to generate values of type $t_g \in \mathcal{T}$. Assume the first and second premises of the rule T:Gen:TypeName:

$$t_g \coloneqq \tau_1 + \cdots + \tau_k \qquad \tau_i \coloneqq \tau_{i,1} \times \cdots \times \tau_{i,n_i} \qquad (i = 1,\ldots,k). \tag{9.2.32}$$

The third premise requires the categorical distributions $d^{\text{base}}_{\text{sum}}$ and $d^{\text{rec}}_{\text{sum}}$ to have $k \in \mathbb{N}$ many components, where k is the number of components in the sum type t_g (Eq (9.2.32)). Additionally,

the fourth premise requires each categorical distribution $d^{i,t}_{\text{prod}}$ ($i = 1, \ldots, k$ and $t \in \text{dom}(T)$) to have $n_i \in \mathbb{N}$ many components, where n_i is the number of components in the product type τ_i. The last premise then requires that the code block h_i ($i = 1, \ldots, k$) to be well-typed.

The rule T:CodeBlock concerns a code block h of the form

$$h \equiv \text{let } x_1 = g_1 \langle e_{1,1}, \ldots, e_{1,n_1} \rangle \text{ in } \ldots \text{ let } x_k = g_k \langle e_{k,1}, \ldots, e_{k,n_k} \rangle \text{ in } \text{fold}_t(c \langle x_1, \ldots, x_k \rangle). \tag{9.2.33}$$

Each generator application $g_i \langle e_{i,1}, \ldots, e_{i,n_i} \rangle$ ($i = 1, \ldots, k$) must have type τ_i such that, when all generated components of a tuple are assembled into $\text{fold}_t(c \langle x_1, \ldots, x_k \rangle)$, it has type $\tau := \tau_1 \times \cdots \times \tau_k$. To type-check a generator function, the rule T:Gen:App is used.

Finally, the rule T:Gen:Int concerns a generator for integers:

$$\text{gene}\, g \langle x_1, \ldots, x_m \rangle = (e_{\text{lower}}, e_{\text{upper}}, e_\ell), \tag{9.2.34}$$

where the expressions e_{lower} and e_{upper} must have type `int`, and the expression e_ℓ must have type $L_{\texttt{int}}$, which is the integer-list type (Eq (9.2.2)).

Infeasible target sizes A well-typed generator does not necessarily successfully generate values. A generator raises an exception when the rule E:Gen:TypeName in the operational semantics (Listing 9.2) performs

$$(S_1, \ldots, S_k) = \text{split}((d^{j,t}_{\text{prod}}; t \in \text{dom}(T)), S[t_g \mapsto S(t_g) - 1], \pi). \tag{9.2.35}$$

It happens when the generator attempts to generate a value $\text{fold}_{t_g}(\cdot)$, whereas the target size $S(t_g)$ is zero and hence cannot be decremented.

To illustrate how infeasible target sizes can arise, consider the integer-list type:

$$L_{\texttt{int}} \coloneqq (\texttt{nil}\, \texttt{unit}) + (\texttt{cons}\, \texttt{int} \times L_{\texttt{int}}). \tag{9.2.36}$$

It has two branches in the sum type, equipped with data constructors `nil` and `cons`, respectively. The first branch leads to the empty list, which has the size 1 with respect to the type name $L_{\texttt{int}}$. Consequently, in a generator g for integer lists, the distribution $d^{\text{base}}_{\text{sum}}$ must be

$$d^{\text{base}}_{\text{sum}} = (1, 0). \tag{9.2.37}$$

That is, the generator chooses the `nil` data constructor whenever the target size is $s_{L_{\texttt{int}}} = 1$. Otherwise, if we have

$$d^{\text{base}}_{\text{sum}} = (1 - \epsilon, \epsilon) \qquad (\epsilon > 0), \tag{9.2.38}$$

then the generator attempts to create a non-empty list with probability $\epsilon > 0$ even when the target size is inadequate (i.e., $s_{L_{\texttt{int}}} = 1$).

To avoid inadequate target sizes, one idea is to first identify which data constructors in the type $L_{\texttt{int}}$ can lead to the type itself. The `nil` constructor never leads to the type $L_{\texttt{int}}$: as we unfold the product type `unit` associated with the `nil` constructor, we never reach the type $L_{\texttt{int}}$. By contrast, the `cons` data constructor always leads to the type $L_{\texttt{int}}$: we immediately reach the target type by following the second component of the product type $\texttt{int} \times L_{\texttt{int}}$. By restricting

the support of the distribution $d_{\text{sum}}^{\text{base}}$ to the `nil` constructor, which never leads to the target type, the generator never attempts to generate a non-empty list when the target size is inadequate.

Dually to the distribution $d_{\text{sum}}^{\text{base}}$, the distribution $d_{\text{sum}}^{\text{rec}}$ should ideally be

$$d_{\text{sum}}^{\text{rec}} = (0, 1). \tag{9.2.39}$$

That is, the generator should choose the `cons` data constructor whenever the target size is $s_{L_{\texttt{int}}} > 1$. Otherwise, the generator might try to create the empty list when $s_{L_{\texttt{int}}} > 1$, yielding a value with a different size from the target size. This is problematic, but it does not raise an exception during execution, unlike in Eq (9.2.38).

9.3 Optimization of Generators

This section describes an optimization algorithm to identify generators that trigger high costs of input programs. The optimization algorithm runs a genetic algorithm on the abstract syntax trees (ASTs) of generators. Each iteration of the genetic algorithm creates a new population of generators from the old population by genetic operations (e.g., mutations and crossovers).

9.3.1 Templates of Generators

To create the initial population in a genetic algorithm, generators are randomly synthesized. This is done in two steps:

1. A template of a generator program is synthesized; and
2. Generator definitions are synthesized from the template.

The template defines the following characteristics of a generator program G:

- The set of generator identifiers $g \in G$;
- The call graph of generators in G describing how they call each other;
- The number and types of parameters of generators $g \in G$; and
- The supports of distributions $d_{\text{sum}}^{\text{base}}$, $d_{\text{sum}}^{\text{rec}}$, and $(d_{\text{prod}}^{i,t}; i = 1, \ldots, k, t \in \text{dom}(T))$ in Eq (9.2.14).

Once a template is generated, a generator definition for each identifier $g \in G$ is fleshed out with expressions and distributions.

Call graphs of generators Generators can call each other. For example, assume that a generator g for a type name $t \in \mathcal{T}$ contains the following code block:

$$h \equiv \text{let } x_1 = g_1 \langle e_{1,1}, \ldots, e_{1,n_1} \rangle \text{ in } \ldots \text{ let } x_k = g_k \langle e_{k,1}, \ldots, e_{k,n_k} \rangle \text{ in } \text{fold}_t(c \langle x_1, \ldots, x_k \rangle). \tag{9.3.1}$$

The generator g calls generators $g_1, \ldots, g_k$, some of which may be identical to g itself. Hence, a call graph of generators contains edges from the generator g to generators $g_1, \ldots, g_k$.

To decide on the set of generator identifiers and their call graph, a template-synthesis algorithm takes as input a categorical distribution d_{cycle} of the cycle length of mutual recursion:

$$d_{\text{cycle}} = (p_1, \ldots, p_k) \qquad (p_1, \ldots, p_k \in [0, 1]; \textstyle\sum_i p_i = 1). \tag{9.3.2}$$

$$\frac{t \in \mathrm{dom}(T)}{t \xrightarrow[T]{\text{always}} t} \qquad \frac{(t_1 \coloneqq \tau_1) \in T \qquad \tau_1 \xrightarrow[T]{\text{always}} t_2}{t_1 \xrightarrow[T]{\text{always}} t_2} \qquad \frac{\exists i \in \{1, \ldots, k\}.\tau_i \xrightarrow[T]{\text{always}} t}{\tau_1 \times \cdots \times \tau_k \xrightarrow[T]{\text{always}} t} \qquad \frac{\forall i \in \{1, \ldots, k\}.\tau_i \xrightarrow[T]{\text{always}} t}{c_1\,\tau_1 + \cdots + c_k\,\tau_k \xrightarrow[T]{\text{always}} t}$$

Lst. 9.6: Judgment $\tau \xrightarrow[T]{\text{always}} t$ that a type τ always reaches a type $t \in \mathrm{dom}(T)$.

This distribution is used to determine how many generators of the same target type name $t \in \mathcal{T}$ are involved in mutual recursion. For instance, suppose a call graph contains a cycle wherein the generators of a target type name $t \in \mathcal{T}$ are $g_1, \ldots, g_k$ ($k \in \mathbb{N}_{\geq 1}$). The cycle may contain other generators (of target types other than t). If a generator g_i (indirectly) calls g_{i+1} ($i = 1, \ldots, k-1$) and $g_1 \equiv g_k$, then this mutual recursion among generators of target type t has cycle length k.

Let $\tau_{\text{target}} \in \{\texttt{int}\} \cup \mathcal{T}$ be the target type of values to generate. If $\tau_{\text{target}} \equiv \texttt{int}$, the template-synthesis algorithm simply creates a call graph with a sole generator g for integers, which does not call other generators.

Conversely, if $\tau_{\text{target}} \in \mathcal{T}$ is a type name, the template-synthesis algorithm walks through the type definition $\tau_{\text{target}} \coloneqq \tau$, iteratively adding edges to the call graph. The algorithm explores the type definition τ, maintaining lists $\mathbf{g}_t$ ($t \in \mathcal{T}$) of generator identifiers of target type t created so far. The algorithm adds a new generator and a new call-graph edge to this generator whenever a type name t is encountered such that $|\mathbf{g}_t| < k_t$, where k_t is the mutual-recursion cycle length for the type name t and is drawn from the distribution d_{cycle} (Eq (9.3.2)). Otherwise, if the mutual-recursion cycle length has been reached (i.e., $|\mathbf{g}_t| = k_t$), the algorithm links the current generator to the first generator in the list $\mathbf{g}_t$, thereby closing a cycle of mutual recursion.

Types of generator parameters Let $t_{\text{target}} \in \mathcal{T}$ be a target type name and T be a finite set of type definitions. For each generator in a generator program, the number of parameters is drawn from a user-specified categorical distribution. The types of these parameters are chosen from the set of types $\tau \in \{\texttt{int}\} \cup \mathcal{T}$ such that (i) τ can be reached from the target type name t_{target} as we unfold and explore its type definition in the set T; and (ii) τ does not reach t_{target}. Reachability of types is formally given by a judgment $\tau \xrightarrow[T]{\text{can}} t$ defined in Listing 9.7.

The second condition ensures that argument types are strictly simpler than the target type t_{target}. For instance, assume the target type is the integer-list type: $t_{\text{target}} \coloneqq L_{\texttt{int}}$ (Eq (9.2.2)). Argument types should not be the target type $L_{\texttt{int}}$ itself. Rather, it is more sensible for generators to generate values of a complex type t_{target} using arguments of simpler types. Thus, generators for integer lists can only have arguments of type `int`, but not $L_{\texttt{int}}$.

Reachability of types Reachability of types is given by two judgments:

$$\tau \xrightarrow[T]{\text{always}} t \qquad \tau \xrightarrow[T]{\text{can}} t, \tag{9.3.3}$$

where τ is a type (including not only atomic types $\{\texttt{int}\} \cup \mathcal{T}$ but also product types and sum types), T is a finite set of type definitions, and $t \in \mathcal{T}$ is a type name. The judgment $\tau \xrightarrow[T]{\text{always}} t$

$$\frac{t \in \mathrm{dom}(T)}{t \xrightarrow[T]{\text{can}} t} \qquad \frac{(t_1 \coloneqq \tau_1) \in T \qquad \tau_1 \xrightarrow[T]{\text{can}} t_2}{t_1 \xrightarrow[T]{\text{can}} t_2} \qquad \frac{\exists i \in \{1,\ldots,k\}.\tau_i \xrightarrow[T]{\text{can}} t}{\tau_1 \times \cdots \times \tau_k \xrightarrow[T]{\text{can}} t} \qquad \frac{\exists i \in \{1,\ldots,k\}.\tau_i \xrightarrow[T]{\text{can}} t}{c_1\,\tau_1 + \cdots + c_k\,\tau_k \xrightarrow[T]{\text{can}} t}$$

Lst. 9.7: Judgment $\tau \xrightarrow[T]{\text{can}} t$ that a type τ can reach a type name $t \in T$.

means that, as the type τ is unrolled, it eventually reaches the type name $t \in \mathrm{dom}(T)$, regardless of which paths are taken. That is, every value of the type τ contains a subexpression of type name t. Dually, the judgment $\tau \xrightarrow[T]{\text{can}} t$ means that, as the type τ is unrolled, at least one path reaches the type name t. That is, at least one value of type τ contains a subexpression of type name t.

Listings 9.6 and 9.7 define the two judgments (9.3.3). Their inductive definitions differ in how they handle sum types: the judgment $\tau \xrightarrow[T]{\text{always}} t$ must hold for all components of a sum type, whereas the judgment $\tau \xrightarrow[T]{\text{can}} t$ only needs to hold for at least one component.

The relation $\tau \xrightarrow[T]{\text{always}} t$ is used in splitting a tuple of target sizes during generator execution (Eq (9.2.23)). The relation $\tau \xrightarrow[T]{\text{always}} t$ is used to determine the supports of categorical distributions that appear inside generator definitions when synthesizing generator definitions.

Supports of distributions Generators for type names (Eq (9.2.14)) contain the following distributions: (i) categorical distributions $d^{\text{base}}_{\text{sum}}$ and $d^{\text{rec}}_{\text{sum}}$ for choosing data constructors in sum types; and (ii) a tuple $(d^{i,t}_{\text{prod}}; i = 1,\ldots,k, t \in \mathrm{dom}(T))$ of categorical distributions for splitting target sizes among components of tuples. The template-synthesis algorithm determines supports (i.e., sets of values with positive probabilities) of these distributions based on the reachability of types (Eq (9.3.3)).

Given a finite set T of type definitions, let $t_g \in \mathrm{dom}(T)$ be a target type name of a generator g. Assume the type name t is defined as

$$t_g \coloneqq \tau_1 + \cdots + \tau_k \qquad \tau_i \coloneqq \tau_{i,1} \times \cdots \times \tau_{i,n_i} \qquad (i = 1,\ldots,k). \tag{9.3.4}$$

The supports of the distributions $d^{\text{base}}_{\text{sum}}$ and $d^{\text{rec}}_{\text{sum}}$ in the generator g are

$$\mathrm{support}(d^{\text{base}}_{\text{sum}}) = \left\{ t \in \mathrm{dom}(T) \mid \neg \left((\tau_1 + \cdots + \tau_k) \xrightarrow[T]{\text{can}} t_g \right) \right\} \tag{9.3.5}$$

$$\mathrm{support}(d^{\text{rec}}_{\text{sum}}) = \left\{ t \in \mathrm{dom}(T) \mid (\tau_1 + \cdots + \tau_k) \xrightarrow[T]{\text{can}} t_g \right\}. \tag{9.3.6}$$

The supports of the distributions $d^{i,t}_{\text{prod}}$ $(i = 1,\ldots,k, t \in \mathrm{dom}(T))$ are

$$\mathrm{support}(d^{i,t}_{\text{prod}}) = \left\{ t \in \mathrm{dom}(T) \mid \tau_i \xrightarrow[T]{\text{can}} t \right\} \qquad (i = 1,\ldots,k, t \in \mathrm{dom}(T)). \tag{9.3.7}$$

Algorithm 2 Genetic algorithm for searching for an optimal generator program

Require: Population size $N_{\text{popu}} \geq 1$; tournament size $1 \leq N_{\text{tour}} \leq N_{\text{popu}}$
Require: Number $N_{\text{epochs}} \geq 1$ of epochs; number $N_{\text{iters}} \geq 1$ of iterations within each epoch
Require: Hyperparameters $\theta = (\theta_{\text{temp}}, \theta_{\text{gene}}, \theta_{\text{op}}, \theta_{\text{score}})$

1: **procedure** GeneticAlgorithm($N_{\text{popu}}, N_{\text{tour}}, N_{\text{epochs}}, N_{\text{iters}}, \theta$)
2: **for** $i = 1, \ldots, N_{\text{epochs}}$ **do**
3: $\mathbb{T} \leftarrow$ `synthesizeTemplate`(θ_{temp})
4: $\mathbb{P} \leftarrow$ `randomPopulation`$(N_{\text{popu}}, \mathbb{T}, \theta_{\text{gene}})$
5: $G_{\text{best}} \leftarrow \text{bestGene}(\mathbb{P}, \theta_{\text{score}})$
6: **for** $j = 1, \ldots, N_{\text{iters}}$ **do**
7: **for** $n = 1, \ldots, N_{\text{popu}}$ **do**
8: $G_n \leftarrow$ GeneticOperation$(\mathbb{T}, \mathbb{P}, N_{\text{tour}}, (\theta_{\text{gene}}, \theta_{\text{op}}))$
9: $\mathbb{P} \leftarrow \{G_1, \ldots, G_{N_{\text{popu}}}\}$
10: $G_{\text{best}} \leftarrow \text{bestGene}(\{G_{\text{best}}\} \cup \mathbb{P}, \theta_{\text{score}})$
11: **return** G_{best}

9.3.2 Genetic Algorithm

To search for an optimal (or nearly optimal) generator program of a given target type, I adopt a genetic algorithm[5]. Genetic algorithms are heuristic-based optimization algorithms for discrete search spaces and are widely used in fuzzing (e.g., AFL [233], SlowFuzz [185], and PerfFuzz [155]). A genetic algorithm maintains a population of generator programs, and in each iteration, a new population is created from the old one by performing genetic operations (e.g., mutations and crossovers).

Alg. 2 shows the pseudocode of the genetic algorithm for generator optimization. This pseudocode is adapted from Singularity, a generator fuzzer developed by Wei et al. [224].

In the i^{th} epoch ($i = 1, \ldots, N_{\text{epochs}}$), the algorithm synthesizes a template $\mathbb{T}$ of a generator program based on a hyperparameter θ_{temp} (line 3). The hyperparameter θ_{temp} specifies (i) a set T of type definitions; (ii) a target type name $t \in \text{dom}(T)$; (iii) a categorical distribution d_{cycle} of the mutual-recursion cycle length (Eq (9.3.2)); and (iv) a categorical distribution of the number of input variables in generators. Next, a population $\mathbb{P}$ of $N_{\text{popu}} \geq 1$ many generator programs is created by randomly synthesizing generators from the template $\mathbb{T}$ and a hyperparameter θ_{gene} (line 4). The hyperparameter θ_{gene} specifies the maximum depth of ASTs of expressions inside generator definitions. In line 5, to identify the best generator G_{best} in the initial population $\mathbb{P}$, the algorithm calculates scores of all generators in the population according to a hyperparameter θ_{score}. The algorithm then selects a generator with the highest score.

In each iteration $j = 1, \ldots, N_{\text{iters}}$ within each epoch, a new population is created from the previous one. In line 8, each generator G_n ($n = 1, \ldots, N_{\text{popu}}$) to be included in the new population is created by performing a procedure GeneticOperation (Alg. 3). Finally, a new population $\mathbb{P}$ is formed (line 9), and the best generator G_{best} so far is updated (line 10).

[5]The application of genetic algorithms to the abstract syntax trees (ASTs) of programs is known as genetic programming [146, 147].

Algorithm 3 Genetic operation used inside the genetic algorithm

Require: Generator template $\mathbb{T}$ and population $\mathbb{P}$
Require: Tournament size $N_{\text{tour}} \geq 1$
Require: Hyperparameter $\theta = (\theta_{\text{gene}}, \theta_{\text{op}})$

procedure GeneticOperation($\mathbb{T}, \mathbb{P}, N_{\text{tour}}, \theta$)
  for $g_i \in \mathbb{T}$[generators] **do**
    for $e_j \in g_i$[expressions] **do**
      $\oplus \leftarrow$ `chooseOperator`(θ_{op})
      for $k = 1, \ldots,$ arity$(\oplus)$ **do**
        $H_k \leftarrow$ `tournament`$(\mathbb{P}, N_{\text{tour}}, \theta_{\text{op}})$
        $e_k \leftarrow$ `extractExpression`(H_k, e_j)
      $e_{j,\text{new}} \leftarrow \oplus(e_1, \ldots, e_{\text{arity}(\oplus)}; \theta_{\text{gene}})$
    $g_{i,\text{new}} \leftarrow \{e_{j,\text{new}}\}_j$
  $G_{\text{new}} \leftarrow \{g_{i,\text{new}}\}_i$
  return G_{new}

Scoring To calculate a score of a given generator G, the genetic algorithm first runs the generator G to generate values $v_1, \ldots, v_N$ of varying sizes. The hyperparameter θ_{score} specifies (i) mappings $S_1, \ldots, S_m$ from type names to their target sizes; and (ii) the number of values to generate for each mapping S_i of target sizes. A target program P (specified by the hyperparameter θ_{score}) is executed on each generated values v_i ($i = 1, \ldots, N$), recording a high-water-mark cost $c_i \in \mathbb{Q}_{\geq 0}$. Let $\mathcal{D}$ be a dataset of cost measurements:

$$\mathcal{D} \coloneqq \{(v_i, c_i) \mid i = 1, \ldots, N\}. \tag{9.3.8}$$

Let $n_1, \ldots, n_k$ be size measures (e.g., the length of an outer list and the maximum inner list length) of inputs to the target program P. The genetic algorithm performs optimization-based data-driven analysis (Opt in §7.3.2) to infer a multivariate polynomial bound of a user-specified polynomial degree $d \in \mathbb{N}$

$$(n_1, \ldots, n_k) \mapsto \mathsf{sum}\left(\left\{q_{i_1,\ldots,i_k} \prod_{j=1}^{k} \binom{n_j}{i_j} \mid i_1, \ldots, i_k \in \mathbb{N}, i_1 + \cdots + i_k \leq d\right\}\right), \tag{9.3.9}$$

where polynomial coefficients $q_{i_1,\ldots,i_k} \in \mathbb{Q}_{\geq 0}$ are to be inferred. After the polynomial coefficients $q_{i_1,\ldots,i_k}$ are inferred, those coefficients of the same polynomial degree are summed up:

$$\mathbf{q} \coloneqq (q_d, \ldots, q_0) \qquad q_j \coloneqq \mathsf{sum}\left(\{q_{i_1,\ldots,i_k} \mid i_1 + \cdots + i_k = j\}\right) \qquad (j = 0, \ldots, d). \tag{9.3.10}$$

The score of a generator is given by the tuple $\mathbf{q}$ and is ordered lexicographically such that coefficients with higher polynomial degrees have higher priorities.

Genetic operators Alg. 3 displays the pseudocode of the procedure GeneticOperation, which perform genetic operations to create a new generator program. The notation $\mathbb{T}$[generators]

in line 2 refers to the finite set of generator identifiers (and their call graph) specified by the template $\mathbb{T}$. To synthesize a new generator program, the algorithm synthesizes new generator definitions $g_{i,\text{new}}$, aggregating them into a generator program $G_{\text{new}} \coloneqq \{g_{i,\text{new}}\}_i$ (line 10). Likewise, the notation $g_i[\text{expressions}]$ in line 3 refers to the set of expressions (i.e., arguments to generator calls) and categorical distributions to synthesize. Once all necessary expressions $e_{j,\text{new}}$ are synthesized, they are aggregated into a generator definition $g_{i,\text{new}} \coloneqq \{e_{j,\text{new}}\}_j$ (line 9).

For each expression/distribution e_j to synthesize, the algorithm first randomly chooses a genetic operator $\oplus$ (e.g., mutations and crossovers) according to a hyperparameter θ_{op}, which specifies a categorical distribution of genetic operators (line 4). To determine arguments to the genetic operators $\oplus$, for each $k = 1, \ldots, \text{arity}(\oplus)$, the algorithm performs a tournament (line 6). First, $N_{\text{tour}} \geq 1$ many generators are first sampled uniformly at random from the current population $\mathbb{P}$. The best generator H_k is identified according to the hyperparameter θ_{score}. From the generator H_k, an expression e_k that corresponds to the expression placeholder e_j (i.e., they both appear in the same node inside the AST) is extracted (line 7). Finally, in line 8, a new expression $e_{j,\text{new}}$ is synthesized by

$$e_{j,\text{new}} \leftarrow \oplus(e_1, \ldots, e_{\text{arity}(\oplus)}; \theta_{\text{gene}}), \tag{9.3.11}$$

which performs the genetic operator $\oplus$ on the arguments $e_1, \ldots, e_{\text{arity}(\oplus)}$ while respecting the hyperparameter θ_{gene} (e.g., the maximum AST depth of expressions).

Like in Singularity [224], three genetic operators are offered: mutation $\oplus_{\text{mutate}}$, crossover $\oplus_{\text{cross}}$, and reproduction $\oplus_{\text{repro}}$. The mutation operator $\oplus_{\text{mutate}}$ takes as input an expression e and mutates its AST while preserving its type. Specifically, the mutation operator first randomly chooses a subexpression $e_{1,\text{sub}}$ inside the input expression e, replacing the subexpression $e_{1,\text{sub}}$ with a randomly sampled expression $e_{2,\text{sub}}$ of the same type.

The crossover operator $\oplus_{\text{cross}}$ takes as input two expressions e_1 and e_2 of the same type. It then randomly chooses a pair $(e_{1,\text{sub}}, e_{2,\text{sub}})$ of subexpressions inside e_1 and e_2, respectively, where the two subexpressions have the same type. The crossover operator then swaps the subexpressions $e_{1,\text{sub}}$ and $e_{2,\text{sub}}$ in the ASTs of e_1 and e_2. Finally, the operator returns one of the two modified ASTs. Because the input expressions e_1 and e_2 are of the same type, there exists at least one pair $(e_{1,\text{sub}}, e_{2,\text{sub}})$ of subexpressions of the same type inside the ASTs of e_1 and e_2.

The reproduction operator $\oplus_{\text{repro}}$ takes in an expression simply and returns it without any modification.

9.4 Evaluation

My collaborators and I have implemented a prototype of the generator-optimization algorithm (§9.3) in Python. A user first supplies (i) a finite set T of type definitions; (ii) a Python program $P(x)$ of input type $t \in \text{dom}(T)$; and (iii) hyperparameters for the genetic algorithm (Alg. 2) (e.g., the number of iterations and the maximum AST depth of expressions). The genetic algorithm runs, returning a generator program G_{best} with the highest score among all generators that the algorithm has synthesized.

This section evaluates the prototype implementation of the generator-optimization algorithm. The evaluation aims to answer the following questions:

Q1: Is the DSL of generators (§9.2) expressive enough to capture worst-case inputs of functional programs in practice?

Q2: Can the generator-optimization algorithm find a generator that produces (nearly) worst-case program inputs?

Q3: How does the runtime-cost data of a generator returned by the genetic algorithm compare with the runtime-cost data of a random-input generator?

9.4.1 Benchmark Suite

Benchmark programs To evaluate generator optimization, my collaborators and I have curated a benchmark suite consisting of 21 functional programs (written in Python). All benchmark programs are taken from (with or without modification) type-guided worst-case input generation developed by Wang and Hoffmann [219].

- QuickSort, QuickSortRev, QuickSortStr, and QuickSortRevStr implement (deterministic) quicksort where the head element is used as a pivot in the function `partition`. The four benchmarks differ in (i) whether the function `partition` preserves the ordering of an input list's elements; and (ii) whether integer comparison uses the strong inequality (i.e., $<$) or the weak inequality (i.e., $\leq$).
- InsertionSort implements insertion sort.
- Lpairs takes an integer list x and returns a list of pairs of adjacent elements that are ordered in the list x.
- LpairsAlt is similar to Lpairs, except that LpairsAlt alternates between two modes to determine whether to include a pair in the output: the pair is ordered or out of order.
- Opairs takes an integer list x and returns a list of pairs of (not necessarily adjacent) elements that are ordered in the list x.
- LinearSearch takes two inputs, $x : L(\texttt{int})$ and $k : \texttt{int}$, and traverses the list x to search for the integer k.
- QuickSelect and QuickSelectStr take two inputs, $x : L(\texttt{int})$ and $k : \texttt{int}$, and return the k^{th} smallest element in the list x, provided that $0 \leq k < |x|$. Otherwise, if $k < 0$ or $k \geq |x|$, return some default value.
- Queue maintains a functional queue q implemented using two stacks. The program takes as input a list x containing two kinds of requests: enqueue and dequeue. The kind of requests is indicated by a Boolean value. The program traverses the list x, updating the queue q according to each request in the list x.
- QuickSortPairs and QuickSortPairsStr take in a list x of pairs of integers and run quicksort on the list x using lexicographic comparison between integer pairs.
- SplitSort and SplitSortStr take as input a list x of pairs of integers. The programs first partition the list x into a nested list by grouping together those pairs with identical first components. The programs then sort each inner list by running QuickSort and QuickSortStr, respectively.

- Compare takes as input a pair (x_1, x_2) of integer lists and performs lexicographic comparison between the lists x_1 and x_2.
- QuickSortLists and QuickSortListsStr take as input a nested list $x : L(L(\texttt{int}))$ of integers and run quicksort on the list x using lexicographic comparison.
- SortAll and SortAllStr take as input a nested list $x : L(L(\texttt{int}))$ of integers and sort each inner list in the input x.

If a benchmark's name contains str (which stands for "strong"), it means the benchmark uses the strong inequality (i.e., $<$), as opposed to the weak one (i.e., $\leq$), in integer comparison.

The benchmarks QuickSort and QuickSortStr implement the function `partition` such that it preserves the ordering of an input list's elements. For example, given an input list $x := [1, 2, 4, 5]$ and an integer pivot $p := 3$, the list x is partitioned into $([1, 2], [3, 4])$. This implementation is obtained if the `partition` is implemented recursively without using accumulators.

By contrast, the benchmarks QuickSortRev and QuickSortRevStr implement the function `partition` such that the two output lists reversely order the integers. For example, given an input list $x := [1, 2, 4, 5]$ and an integer pivot $p := 3$, the list x is partitioned into $([2, 1], [4, 3])$. This happens because accumulators a_1, a_2 are used to maintain output lists: each iteration of the function `partition` takes out the head h of an input list, prepending the head h to one of the two accumulators a_1, a_2. This implementation of the `partition` function is used in the benchmark qsort in Wang and Hoffmann [219].

Resource metrics The following resource metrics are used:

- Lpairs and LpairsAlt concern the output size.
- SplitSort concerns the number of comparisons in the function `quicksort` plus the number of function calls to the function `append`. This is identical to the tick metric used in the benchmark split_sort in Wang and Hoffmann [219].
- The remaining benchmarks concern the number of function calls.

Input types The input types of the benchmarks are listed below:

- QuickSort to Opairs: $L(\texttt{int})$.
- LinearSearch to QuickSelectStr: $L(\texttt{int}) \times \texttt{int}$.
- Queue: $L(\texttt{bool} \times \texttt{int})$, where $\texttt{bool} := \texttt{unit} + \texttt{unit}$.
- QuickSortPairs to SplitSortStr: $L(\texttt{int} \times \texttt{int})$.
- Compare: $L(\texttt{int}) \times L(\texttt{int})$.
- QuickSortLists to SortAllStr: $L(L(\texttt{int}))$.

9.4.2 Experiment Results

Setup To evaluate the effectiveness of the generator-optimization algorithm, I measure the relative errors of runtime-cost data generated by two generator programs:

1. A random-input generator program G_{random} and

2. The best generator program G_{genetic} synthesized by the genetic algorithm (Alg. 2).

The random-input generator G_{random} fills in data structures (e.g., lists and nested lists) with (i) integers drawn uniformly at random from a broad interval; and (ii) Booleans drawn from Bernoulli(0.5) (used in the benchmark Queue).

The genetic algorithm (Alg. 2) is performed with the following parameters:

$$N_{\text{popu}} = 20 \qquad N_{\text{tour}} = 5 \qquad N_{\text{epochs}} = 4 \qquad N_{\text{iters}} = 15. \tag{9.4.1}$$

All benchmarks share the same set of other hyperparameters (e.g., the maximum depth of ASTs and a distribution of the mutual-recursion cycle length), with the only exception being the benchmark LpairsAlt. This benchmark uses a categorical distribution d_{cycle} (Eq (9.3.2))

$$d_{\text{cycle}} \coloneqq (0, 0, 0.2, 0.6, 0.2), \tag{9.4.2}$$

which means that the mutual-recursion cycle length is set to 3 with probability 0.2, 4 with probability 0.6, and 5 with probability 0.2. Meanwhile, the remaining benchmarks use the categorical distribution

$$d_{\text{cycle}} \coloneqq (0.6, 0.4). \tag{9.4.3}$$

I use a different distribution d_{cycle} for LpairsAlt since its worst-case inputs have a cycle length of 4, while the cycle length of 1 suffices for worst-case inputs of other benchmarks.

First, the generator programs G_{random} and G_{genetic} are executed 100 times, each with (i) a different seed for pseudorandom number generators; and (ii) a randomly generated input expression (if the generator programs take inputs). These 100 runs yield values $v_1, \ldots, v_{100}$, and they are fed to a benchmark program P to record cost measurements $c_1, \ldots, c_{100}$. Finally, the cost measurements are used to calculate the 50^{th} and 95^{th} percentiles of relative errors with respect to ground-truth cost bounds.

To determine ground-truth cost bounds of the benchmark programs, they are re-implemented in OCaml (while preserving the semantics) and are analyzed by RaML [117, 118] to statically infer sound polynomial cost bounds. The ground-truth symbolic bounds inferred by Conventional AARA are tight for all the 21 benchmark programs (i.e., worst-case inputs achieve the bounds for infinitely many input sizes). However, it does not mean that the bounds are tight for all input sizes. For instance, in the benchmark Lpairs, Conventional AARA infers a polynomial bound $n \mapsto n$, but this is only tight when an input list contains an even number of integers.

Relative errors Tab 9.1 reports relative errors of runtime-cost data generated by random-input generators G_{random} and the best generator G_{genetic} returned by the genetic algorithm for the 21 benchmark programs. The second column shows ground-truth asymptotic cost bounds. In the benchmark Compare, the size measure n is the size of the first input list, which is also equal to the second input size. For the last four benchmarks, the size measure n is the outer list length, and the size measure m is the maximum inner list length. The third column shows the fixed input size for which relative errors are measured.

For relative errors, Tab 9.1 calculates the relative errors of the 100 costs $c_1, \ldots, c_{100}$ and shows their 50^{th} and 95^{th} percentiles. Given a ground-truth cost bound $c^{\text{max}} \in \mathbb{Q}_{\geq 0}$ and a cost $c \in \mathbb{Q}_{\geq 0}$ generated by a generator, its relative error is defined as

$$(c - c^{\text{max}})/c^{\text{max}}, \tag{9.4.4}$$

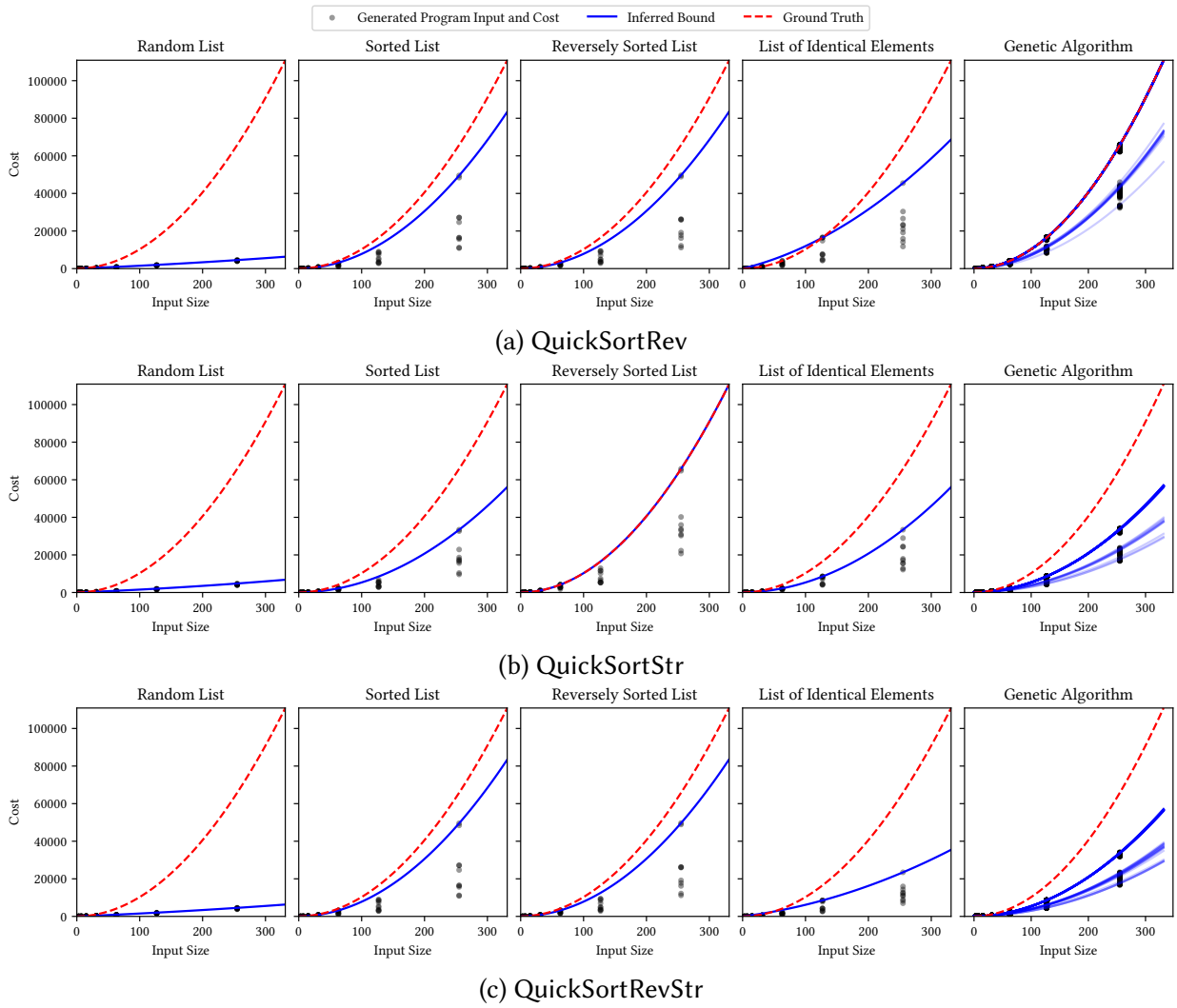
Table 9.1: Relative errors of runtime-cost data generated by a random-input generator and the genetic algorithm. For each benchmark, the 50^{th} and 95^{th} percentiles of relative errors are reported. The symbols ✓, =, and ✗ indicate that the relative errors of the genetic algorithm are better than, equal to, and worse than those of random-input generators, respectively.

			Relative Error						
			50^{th} Percentile			**95^{th} Percentile**			**Time**
Benchmark	**Ground Truth**	**Input Size**	Random	Genetic		Random	Genetic		(min)
QuickSort	n^2	300	-0.945	-0.021	✓	-0.939	-0.007	✓	39.0
QuickSortRev	n^2	300	-0.944	-0.020	✓	-0.938	-0.007	✓	40.3
QuickSortStr	n^2	300	-0.945	-0.500	✓	-0.939	-0.492	✓	38.0
QuickSortRevStr	n^2	300	-0.945	-0.500	✓	-0.939	-0.492	✓	34.5
InsertionSort	n^2	300	-0.498	-0.419	✓	-0.463	-0.363	✓	31.7
Lpairs	n	300	-0.498	-0.017	✓	-0.438	-0.003	✓	24.8
LpairsAlt	n	300	-0.505	-0.030	✓	-0.444	-0.003	✓	26.3
Opairs	n^2	300	-0.246	-0.174	✓	-0.233	-0.065	✓	147.3
LinearSearch	n	300	-0.000	-0.000	=	-0.000	-0.000	=	46.0
QuickSelect	n^2	300	-0.985	-0.705	✓	-0.984	-0.192	✓	46.4
QuickSelectStr	n^2	300	-0.985	-0.993	✗	-0.984	-0.293	✓	44.5
Queue	n	300	-0.120	-0.013	✓	-0.095	-0.004	✓	34.4
QuickSortPairs	n^2	300	-0.945	-0.042	✓	-0.938	-0.016	✓	59.5
QuickSortPairsStr	n^2	300	-0.945	-0.507	✓	-0.938	-0.495	✓	58.7
SplitSort	n^2	300	-0.996	-0.048	✓	-0.996	-0.021	✓	65.2
SplitSortStr	n^2	300	-0.996	-0.518	✓	-0.996	-0.502	✓	60.1
Compare	n	300	-0.997	-0.927	✓	-0.997	-0.705	✓	193.2
QuickSortLists	n^2m	(40, 40)	-0.980	-0.604	✓	-0.978	-0.470	✓	242.6
QuickSortListsStr	n^2m	(40, 40)	-0.980	-0.603	✓	-0.978	-0.480	✓	293.4
SortAll	nm^2	(40, 40)	-0.735	-0.017	✓	-0.729	-0.011	✓	253.7
SortAllStr	nm^2	(40, 40)	-0.736	-0.457	✓	-0.729	-0.453	✓	263.7

which is always in the range $[-1, 0]$ because $0 \leq c \leq c^{\max}$ must hold. The closer the relative error is to 0, the more desirable it is. The symbols ✓, =, and ✗ indicate that the relative errors of the genetic algorithm are better than, equal to, and worse than those of random-input generators, respectively.

In 20/21 benchmarks, the generator G_{genetic} has a better (or equal) 50^{th}-percentile relative error than the random-input generator G_{random}. The only exception is the benchmark QuickSelectStr, where the generator G_{genetic} has a slightly worse 50^{th}-percentile relative error than the generator G_{random}. Also, in the benchmark LinearSearch, both generators have the relative errors of -0.000. The generator G_{random} already performs well in this benchmark because a randomly generated integer $k : \texttt{int}$ is unlikely to exist in a randomly generated integer list $x : L(\texttt{int})$. Consequently, LinearSearch will likely traverse the entire list x to search for the integer k, yielding the worst-case cost.

In all 21/21 benchmarks, including QuickSelectStr, the generator G_{genetic} has a better (or equal) 95^{th}-percentile relative error. This answers the question **Q3**: the generators returned by the genetic algorithm yield higher costs than the random-input generators.



(a) QuickSortRev

(b) QuickSortStr

(c) QuickSortRevStr

Figure 9.2: Runtime-cost data in the benchmarks QuickSortRev, QuickSortStr and QuickSortRevStr.

Analysis time The last column of Tab 9.1 shows the running time of the genetic algorithm. The analysis time ranges from 24.8 min (in Lpairs) to 293.4 min (in QuickSortListsStr). Its variation can be partially explained by the input type of benchmark programs, which in turn determines the number of atomic-type values (e.g., integers) that need to be drawn for each run of a generator. For example, the benchmarks QuickSort to LpairsAlt, all of which have the input type $L(\texttt{int})$, have similar running time of the genetic algorithm. However, the benchmark Opairs, which also has the same input type $L(\texttt{int})$, takes significantly longer analysis time.

Expressiveness of generators To address the questions **Q1** and **Q2**, I consider the four variants of quicksort: QuickSort, QuickSortRev, QuickSortStr, and QuickSortRevStr. QuickSort and QuickSortRev use the weak inequality (i.e., $\leq$) in the function `partition`, while QuickSortStr and QuickSortRevStr use the strong inequality (i.e., $<$). Also, the function `partition` in Quick-

Sort and QuickSortStr preserve the ordering of elements in an input list, while the function `partition` in QuickSortRev and QuickSortRevStr reverses the ordering due to the use of accumulators. In all the four benchmarks, the generators G_{genetic} returned by the genetic algorithm generate lists of identical elements (with a high probability).

Two findings from Tab 9.1 are:

- In QuickSort and QuickSortRev, the generators G_{genetic} have the 50^{th}-percentile relative errors close to zero: -0.021 and -0.020, respectively.
- In QuickSortStr and QuickSortRevStr, the generator G_{genetic} has the 50^{th}-percentile relative error -0.500, which is significantly worse than those of QuickSort and QuickSortRev.

Figs. 9.1 and 9.2 show the runtime-cost data of various generators, including the random-input generator G_{random} (leftmost plot), and the best generators across iterations in the genetic algorithm (rightmost plot). The worst-case input lists of the four benchmarks are listed below:

- QuickSort: a (non-strictly reversely) sorted list (e.g., $[4, 3, 2, 1]$ where elements are strictly reversely sorted and $[0, 0, 0, 0]$ where elements are identical);
- QuickSortRev: a list where all elements are identical (e.g., $[0, 0, 0, 0]$);
- QuickSortStr: a strictly reversely sorted list (e.g., $[4, 3, 2, 1]$, but not $[0, 0, 0, 0]$); and
- QuickSortRevStr: a list that would be reversely sorted if we go back and forth between the front and rear of the list (e.g., $[5, 3, 1, 2, 4]$).

Although a strictly sorted list (e.g., $[1, 2, 3, 4]$) incurs a quadratic cost in QuickSort, it is not a worst-case input of QuickSort. This is because such a list does not incur the maximum cost of the function append when concatenating two recursive results of the function `quicksort`.

In QuickSort (Fig. 9.1) and QuickSortRev (Fig. 9.2a), the generators G_{genetic} (shown in the rightmost plots) yield inferred cost bounds (blue lines) that are very close to the ground-truth bounds (red dashed lines). By contrast, in QuickSortStr (Fig. 9.2b), the generator G_{genetic} yields a cost bound that is significantly below the ground-truth bound. Meanwhile, the third plot in Fig. 9.2b shows that the generator G_{rev} for (strictly) reversely sorted lists yields an inferred cost bound close to the ground-truth bound. This answers the question **Q2**: the genetic algorithm can fail in discovering an optimal (i.e., worst-case) generator even if it is expressible in the DSL of generators (§9.2). To fix this problem, the genetic algorithm should run with more epochs and iterations.

Moreover, in QuickSortRevStr (Fig. 9.2c), the worst-case input is not expressible in the DSL of generators, although the variant QuickSortRevStr of quicksort is a realistic implementation. A worst-case input of QuickSortRevStr is a length-n list x_n ($n \in \mathbb{N}$) defined as

$$x_0 \coloneqq [\,] \qquad x_1 \coloneqq [1] \qquad x_n \coloneqq [n] + x_{n-2} + [n-1] \qquad (n \geq 2), \tag{9.4.5}$$

where the operator + denotes list concatenation. Such lists cannot be inductively generated: list elements are not determined exclusively by the preceding elements. To generate such lists, the sampling of integers must be aware of the target size of a list. However, this is not permitted in the current design of generators: the target size cannot appear inside generator definitions, but is only used to determine when the generator's execution terminates. This answers the question **Q1**: the DSL of generators cannot capture worst-case inputs of some real-world programs.

Probabilistic execution of generators For QuickSort, in the fourth plot of Fig. 9.1, the cost bound inferred using the generator G_{ident} lies below the ground-truth bound with a noticeable margin. This is due to the probabilistic nature of generators: they always have a small but positive probability of deviating from the generator's standard behavior and drawing an integer from a broad interval (E:GEN:INT:FAIL:2 in Listing 9.2). The runtime-cost data in third plot of Fig. 9.1 is generated by a generator G_{ident}, which is described in Remark 9.4.1.

Remark 9.4.1 (Generator for lists of identical elements)**.** *A generator G_{ident} for a length-n list of identical elements works as follows. It has an input parameter x : `int` and generates the i^{th} element v_i in a list by drawing an integer from an interval $[x, x]$. The generator then recursively calls itself, where the parameter x is set to v_i in the recursive call. However, with a small yet positive probability, the generator G_{ident} can sample a random integer from a broad interval, yielding a list of the form*

$$[\underbrace{v_1, \ldots, v_1}_{k \text{ times}}, \underbrace{v_2, \ldots, v_2}_{n-k \text{ times}}] \qquad (0 < k < n; v_1, v_2 \in \mathbb{Z}; v_1 \neq v_2). \tag{9.4.6}$$

In the generation of the list (9.4.6), the generator G_{ident} initially keeps drawing v_1. But halfway through the execution (i.e., after k recursive calls of the generator), it draws a random integer $v_2 \neq v_1$ from a broad interval, drawing v_2 repeatedly afterward. Such a list triggers a noticeably lower cost than the ground-truth cost bound of QuickSort, particularly when k is close to $n/2$.

The above generator G_{ident} is not the only way to generate lists of identical elements (with a high probability). An alternative generator H_{ident} takes as input an integer parameter x : `int` and generates each element in a list by drawing an integer from the interval $[x, x]$. If the generator H_{ident} deviates from the standard behavior once, its output list has the form

$$[\underbrace{v_1, \ldots, v_1}_{k \text{ times}}, v_2, \underbrace{v_1, \ldots, v_1}_{n-k-1 \text{ times}}] \qquad (0 < k < n; v_1, v_2 \in \mathbb{Z}; v_1 \neq v_2). \tag{9.4.7}$$

The list (9.4.7) usually incurs a higher cost than the list (9.4.6) since the result of the function `partition` *is more uneven in the former than in the latter.* «

Likewise, for QuickSortRev, in the fourth plot of Fig. 9.2a, the inferred cost bound (blue line) is far from the ground-truth bound (red line), although lists of identical elements are worst-case inputs of QuickSortRev. This is, again, due to the probabilistic nature of generators. Nonetheless, the genetic algorithm successfully triggers worst-case costs, as shown in the rightmost plot of Fig. 9.2a. This is because some generators produce lists of (almost identical) elements with higher costs than other generators. For instance, output lists of the generator H_{ident} (Remark 9.4.1) are likely to incur higher costs than output lists of the generator G_{ident}, although both generators produce lists of (almost) identical elements.

9.4.3 Random Enumeration of Generators

The generator-optimization algorithm (§9.3) adopts a genetic algorithm to search for a high-cost generator. A benefit of the genetic algorithm is that, in theory, it can be adaptive: the algorithm can choose to focus on different regions of a search space, depending on the performance of the generators that the algorithm has examined. Consequently, for each target program, the

Table 9.2: Relative errors of runtime-cost data generated by random enumeration and the genetic algorithm. For each benchmark, the 50^{th} and 95^{th} percentiles of relative errors are reported. The symbols ✓, =, and ✗ indicate that the relative errors of the genetic algorithm are better than, equal to, and worse than those of the random enumeration, respectively.

			Relative Error					
			50^{th} Percentile			95^{th} Percentile		
Benchmark	**Ground Truth**	**Input Size**	Enumerate	Genetic		Enumerate	Genetic	
QuickSort	n^2	300	-0.021	-0.021	=	-0.007	-0.007	=
QuickSortRev	n^2	300	-0.020	-0.020	=	-0.007	-0.007	=
QuickSortStr	n^2	300	-0.500	-0.500	=	-0.492	-0.492	=
QuickSortRevStr	n^2	300	-0.500	-0.500	=	-0.492	-0.492	=
InsertionSort	n^2	300	-0.498	-0.419	✓	-0.463	-0.363	✓
Lpairs	n	300	-0.017	-0.017	=	-0.003	-0.003	=
LpairsAlt	n	300	-0.030	-0.030	=	-0.003	-0.003	=
Opairs	n^2	300	-0.246	-0.174	✓	-0.234	-0.065	✓
LinearSearch	n	300	-0.000	-0.000	=	-0.000	-0.000	=
QuickSelect	n^2	300	-0.635	-0.705	✗	-0.044	-0.192	✗
QuickSelectStr	n^2	300	-0.582	-0.993	✗	-0.020	-0.293	✗
Queue	n	300	-0.013	-0.013	=	-0.007	-0.004	✓
QuickSortPairs	n^2	300	-0.042	-0.042	=	-0.016	-0.016	=
QuickSortPairsStr	n^2	300	-0.507	-0.507	=	-0.495	-0.495	=
SplitSort	n^2	300	-0.048	-0.048	=	-0.021	-0.021	=
SplitSortStr	n^2	300	-0.518	-0.518	=	-0.502	-0.502	=
Compare	n	300	-0.997	-0.927	✓	-0.983	-0.705	✓
QuickSortLists	n^2m	(40, 40)	-0.602	-0.604	✗	-0.454	-0.470	✗
QuickSortListsStr	n^2m	(40, 40)	-0.601	-0.603	✗	-0.476	-0.480	✗
SortAll	nm^2	(40, 40)	-0.029	-0.017	✗	-0.022	-0.011	✓
SortAllStr	nm^2	(40, 40)	-0.151	-0.457	✗	-0.109	-0.453	✗

genetic algorithm explores a different region of the search space and examines a different set of generators.

An alternative approach to generator optimization is to randomly enumerate generators non-adaptively. In line 8 of Alg. 2, instead of performing genetic operations to create a new population from the old one, we randomly sample generators to form a new population, independently of the old population. In effect, this random-enumeration approach creates a large, finite set of generators by random sampling, evaluates each generator, and returns the highest-scoring generator.

Evaluated on the benchmark suite in §9.4.1, the random-enumeration approach has better performance than the genetic algorithm. Tab 9.2 compares the relative errors of the random enumeration of generators and the genetic algorithm. The symbols ✓, =, and ✗ indicate that the relative errors of the genetic algorithm are better than, equal to, and worse than those of the random enumeration, respectively. For the 50^{th}-percentile relative errors, the genetic algorithm has better errors than the random enumeration in 3/21 benchmarks (indicated by ✓) and worse errors in 6/21 benchmarks (indicated by ✗). They have the same relative errors in the remaining

12/21 benchmarks (indicated by =). For the 95^{th}-percentile relative errors, the genetic algorithm has better errors than the random enumeration in 5/21 benchmarks and worse errors in 5/21 benchmarks.

The superior performance of the random-enumeration algorithm may stem from the following factors:

- The genetic algorithm may get stuck in the current population, struggling to explore a new, remote region of the search space. On the other hand, the random-enumeration algorithm can explore new regions (albeit non-adaptively) with no trouble.
- The search space may not be large enough to demonstrate the benefit of the genetic algorithm, namely adaptive exploration of the search space.

Chapter 10

Conclusion

This chapter summarizes the contributions of this thesis and discusses ideas for future work.

10.1 Contributions

Hybrid resource analysis A chief contribution of the thesis is the development of hybrid resource analysis: resource analysis that integrates two resource-analysis techniques with complementary strengths and weaknesses. Recall the thesis statement from §1.2:

Thesis Statement *Hybrid resource analysis, which integrates two resource-analysis techniques with complementary strengths and weaknesses, can (i) analyze programs and infer symbolic cost bounds beyond the reach of automatic resource analysis (i.e., static and data-driven analyses); and (ii) add more automation to interactive resource analysis.*

To substantiate the statement, this thesis presents two hybrid-resource-analysis techniques: Hybrid AARA (§7) and resource decomposition (§8). They both integrate two resource-analysis techniques via user-adjustable interfaces. Hybrid AARA integrates Conventional AARA (§4) and (optimization-based and Bayesian) data-driven analyses (§7.3), and the user specifies which code fragment is to be analyzed by which analysis technique. In contrast to Hybrid AARA, resource decomposition is not tied to Conventional AARA—it can integrate any pair of resource analyses as long as they infer symbolic cost bounds. The user specifies how an overall cost bound $f(x, g(x))$ should be decomposed into two symbolic bounds, $f(x, r)$ and $g(x)$, which are analyzed by different analysis techniques.

The two hybrid analyses represent two distinct approaches to the integration of resource-analysis techniques. Hybrid AARA and resource decomposition have different designs of the interfaces between constituent analyses: Hybrid AARA adopts a type-based interface, while resource decomposition adopts a numeric-variable-based interface.

The type-based interface of Hybrid AARA uses resource-annotated types, borrowed from Conventional AARA, to record polynomial potential functions stored in inputs and outputs of expressions. Data-driven analysis infers a resource-annotated typing judgment, and Conventional AARA infers a typing tree with a placeholder (i.e., a leaf node) for the statistically inferred

typing judgment. The two inference results are composed by substitution. Data-driven analysis must be aware of the linear constraints associated with the resource-annotated typing tree, leading to a technical challenge in the inference algorithm (§7.4.2). This is addressed by leveraging a recently developed sampling algorithm that combines sampling and linear constraints: ReHMC [47, 49, 50, 171].

The numeric-variable-based interface of resource decomposition uses a numeric variable, as opposed to a resource-annotated type in Hybrid AARA, to represent a symbolic bound. The numeric-variable-based interface overcomes two limitations of the type-based interface of Hybrid AARA: (i) it cannot express, let alone infer, non-polynomial bounds; and (ii) it does not let constituent analyses infer quantities of different resource metrics.

Hybrid AARA and resource decomposition are evaluated on challenging benchmark suites. The evaluation of Hybrid AARA (§7.6) demonstrates that Hybrid AARA successfully infers accurate bounds when (i) Conventional AARA cannot analyze a program at all; and (ii) fully data-driven analyses fail to infer accurate bounds from observed cost measurements. For the resource-decomposition framework, the thesis showcases three concrete instantiations (§8.5–8.7). Their evaluations demonstrate the resource decomposition's ability to infer cost bounds that Hybrid AARA cannot handle without resorting to fully data-driven analysis.

Additional contributions This thesis makes the following additional contributions:

- §6.2 formulates resource analysis as decision problems. Furthermore, building on the resource-analysis decision problems in Gajser [87, 88], I introduce stronger variants of the decision problems where target programs are guaranteed to terminate. These variants are sensible in the presence of data-driven analysis, which requires target programs to terminate in order to measure their costs. §6.3 then proves the undecidability of resource-analysis decision problems introduced in §6.2.
- §6.4 proves that the typable fragment of Conventional AARA is polynomial-time complete. That is, any polynomial-time function can be encoded as a typable program in Conventional AARA.
- §9 develops a language of probabilistic program-input generators that generate values of a given algebraic data type. §9.4 demonstrates that the generators returned by a genetic algorithm can trigger higher computational costs than random-input generators.

10.2 Future Directions

This section discusses four future directions of research.

Data-driven analysis with constraints Hybrid BayesPC (§7.4.2) poses a technical challenge of combining sampling (used in Bayesian data-driven resource analysis) and constraint optimization (used in Conventional AARA). This motivates the adoption of Reflective HMC (ReHMC) [47, 49, 50, 171] in the prototype implementation of Hybrid BayesPC (§7.5.3).

Beyond Hybrid BayesPC, data-driven analysis subject to constraints is worth investigating. For instance, a number of works incorporate constraints into large language models (LLMs)

for code generation [30, 31, 175, 192, 227]. LLMs generate code token by token, where each token is sampled from a probability distribution. To ensure that the code generated by LLMs is syntactically well-formed and well-typed, Mündler et al. [175] incorporate type constraints to LLMs by deciding whether given partial code can be fleshed out to be well-typed.

Resource-component bounds parametric in local inputs As discussed in §8.8.3, in resource decomposition, a symbolic bound of a resource component must be parametric in the global input. If an inferred resource-component bound is parametric in a local input, it is necessary to additionally derive a relation between the local and global inputs. Hence, an interesting research question is to develop static analysis to derive such relations. One idea is to leverage existing works on sized types [21, 130, 218].

Average-case resource analysis Throughout this thesis, the goal of resource analysis is to infer a *worst-case* cost bound. However, a user may instead seek an *average-case* cost bound over a probability distribution of computational cost. To define such a distribution of cost, we can consider a deterministic program equipped with a probability distribution of inputs. A more general setting is a probabilistic program, whose execution is probabilistic.

It is challenging to extend hybrid resource analysis from worst-case cost bounds to average-case cost bounds. Wang et al. [220] extend Conventional AARA to infer upper bounds on expected costs (i.e., average-case costs) of probabilistic functional programs, where resource metrics are required to be monotone. This is because AARA's extension to probabilistic programs only correctly infers expected *net-cost* bounds, but not expected *high-water-mark-cost* bounds. Since Hybrid AARA (§4) builds on Conventional AARA, we can extend Hybrid AARA in the same manner as Wang et al. [220], resulting in hybrid resource analysis for average-case cost bounds under monotone resource metrics.

For resource decomposition (§8), it is unclear how to adapt it to average-case cost bounds. To illustrate a technical challenge, suppose we analyze a probabilistic recursive program P by resource decomposition, where a resource component is the recursion depth of the program P. For a run of the program P, define three random variables: (i) C represents the total cost; (ii) X represents the recursion depth; and (iii) Y represents the maximum cost of individual recursive calls. A natural modification of resource decomposition is to first infer the following two bounds and then return their product:

1. A bound on the expected recursion depth $\mathbb{E}[X]$; and
2. A bound on the expected maximum cost $\mathbb{E}[Y]$ of each individual recursive call.

However, the product $\mathbb{E}[X]\mathbb{E}[Y]$ is not necessarily a valid upper bound of the expected cost $\mathbb{E}[C]$ of the program P. Although $\mathbb{E}[C] \leq \mathbb{E}[XY]$ holds, $\mathbb{E}[XY] \leq \mathbb{E}[X]\mathbb{E}[Y]$ does not necessarily hold, because the covariance

$$\mathrm{cov}(X, Y) \coloneqq \mathbb{E}[XY] - \mathbb{E}[X]\mathbb{E}[Y] \tag{10.2.1}$$

can be negative, zero, or positive.

Hybrid analysis beyond resource analysis Beyond resource analysis, the idea of hybrid analysis may be applicable to non-quantitative program-analysis tasks. As discussed in §7.7.3,

two challenges exist in the application of hybrid analysis to non-quantitative program analysis where search spaces are typically discrete: (i) it is difficult to justify data-driven (and hybrid) analysis; and (ii) effective gradient-based inference algorithms are not applicable.

To overcome the first challenge, a suitable program-analysis task must not be safety critical. For instance, using data-driven analysis to infer standard functional types (e.g., `int` and `int` $\rightarrow$ `int`) is undesirable since functional types are safety critical in some sense: the difference between the 99% and 100% guarantees of sound inferred types can matter to the user. Instead, a more promising domain for data-driven analysis is the refinement type where refinements (i.e., logical formulas that functional types are annotated with) are not safety critical but nevertheless are useful to the user. For example, if data-driven analysis infers that the output of a given program $P(x)$ is a sorted list with probability 0.75 (and a non-sorted list with probability 0.25), the user can meaningfully use the information. Concretely, given this inference result, the user may decide to use insertion sort, instead of merge sort, to sort the output list of the program $P(x)$ since insertion sort is more efficient than merge sort for almost sorted lists.

For the second challenge, discrete search spaces can still be explored effectively. For instance, if the discrete search space is large but finite, deep neural networks, such as LLMs generating tokens for code and text, work well. Also, Bayesian inference for non-continuous probability distributions (e.g., probabilistic context-free grammars [132]) has been extensively investigated.

Appendix A

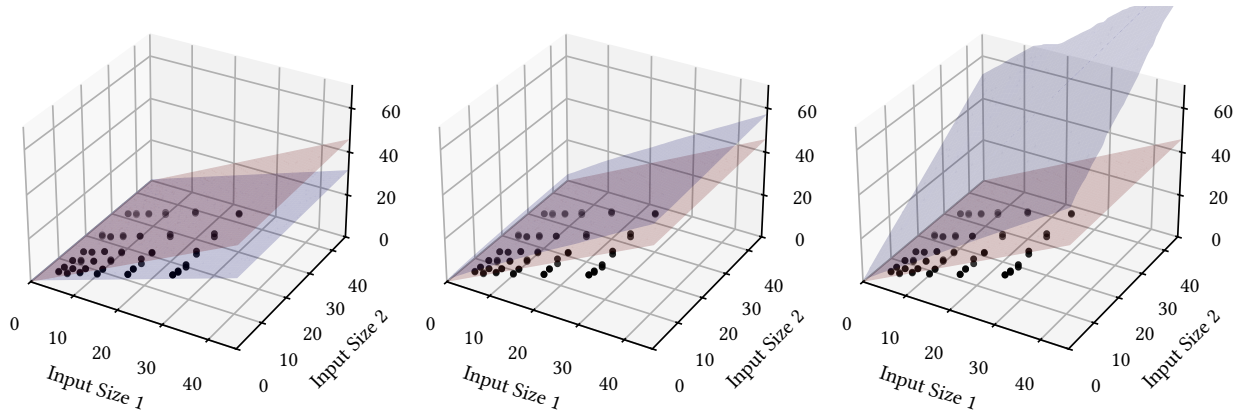
Supplements to Hybrid AARA

A.1 Full Experiment Results

This appendix contains the full evaluation results of all 10 benchmark programs described in §7.6.

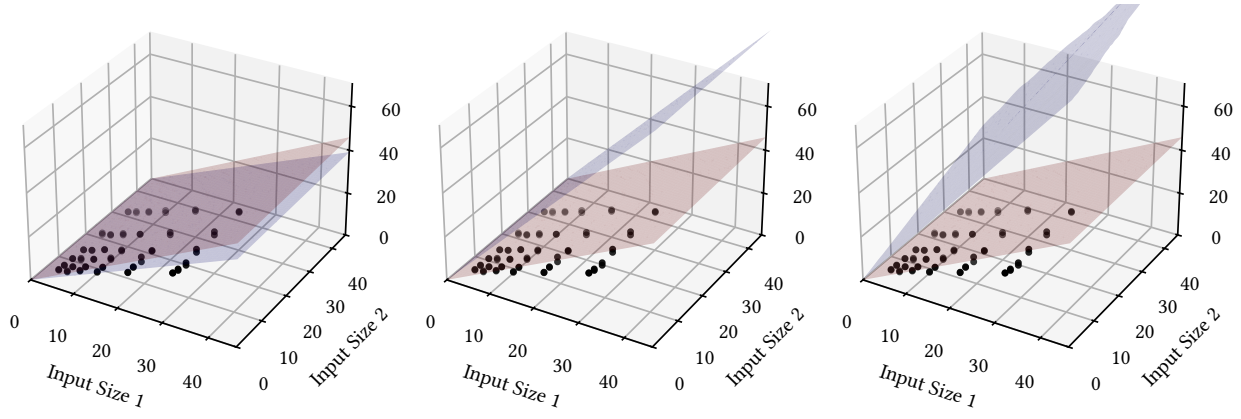
Table A.1: Estimation gaps of inferred cost bounds for MapAppend benchmark on various input sizes.

		Relative Gap in Inferred Cost Bound					
		Data-Driven			Hybrid		
	Percentile	5^{th}	50^{th}	95^{th}	5^{th}	50^{th}	95^{th}
Input Size	**Method**						
(10, 10)	Opt	-0.26	-0.26	-0.26	-0.15	-0.15	-0.15
	BayesWC	0.03	0.41	1.64	0.53	1.03	2.27
	BayesPC	0.85	1.62	2.61	1.18	1.92	2.91
(100, 100)	Opt	-0.32	-0.32	-0.32	-0.15	-0.15	-0.15
	BayesWC	-0.18	0.22	1.17	0.53	1.03	2.27
	BayesPC	0.75	1.54	2.53	1.12	1.89	2.89
(1000, 1000)	Opt	-0.32	-0.32	-0.32	-0.15	-0.15	-0.15
	BayesWC	-0.22	0.20	1.15	0.53	1.03	2.27
	BayesPC	0.74	1.54	2.52	1.11	1.88	2.89



(a) Opt (b) BayesWC (c) BayesPC

Figure A.1: MapAppend Data-Driven



(a) Opt (b) BayesWC (c) BayesPC

Figure A.2: MapAppend Hybrid

```
let incur_cost (hd : int) =
  let modulo = 5 in
  if (hd mod 100) = 0 then Raml.tick 1.0
  else (
    if (hd mod modulo) = 1 then
      Raml.tick 0.85
    else (
      if (hd mod modulo) = 2 then
        Raml.tick 0.65
      else Raml.tick 0.5))

let complex_function (hd : int) =
  let _ = incur_cost hd in
  if hd < 42 then hd / 2 else hd * 2

let rec map_append (xs : int list) (ys : int list) =
    match xs with
    | [] → ys
    | hd :: tl →
        let hd_new = complex_function hd in
        hd_new :: (map_append tl ys)

let map_append2 (xs : int list) (ys : int list) = Raml.
     stat (map_append xs ys)
```

(a) Fully data-driven resource analysis.

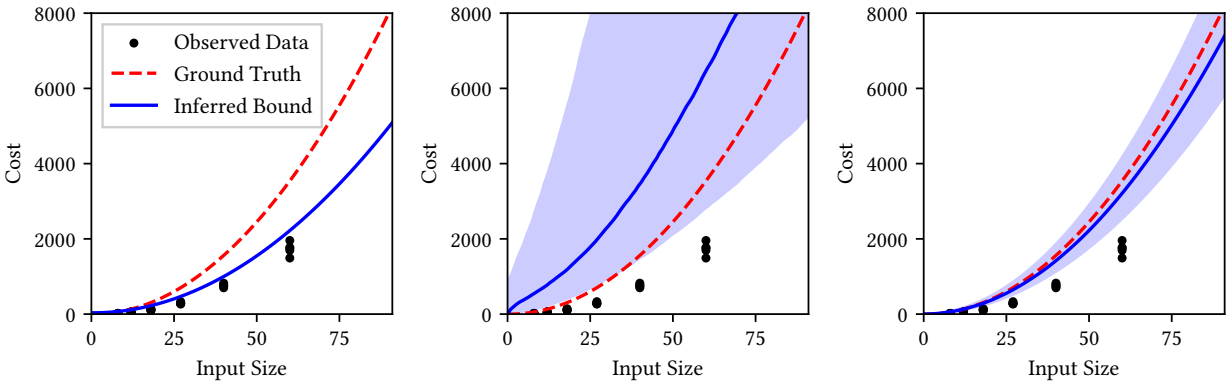
```
let step_function (x : int) (xs : int list) (ys : int
     list) =
  let x_new = complex_function x in (x_new, xs, ys)

let rec map_append (xs : int list)
  (ys : int list) =
  match xs with
  | [] → ys
  | hd :: tl →
    let hd_new, rec_xs, rec_ys = Raml.stat (
         step_function hd tl ys) in
    hd_new :: map_append rec_xs rec_ys
```

(b) Hybrid resource analysis.

Lst. A.1: Source code of MapAppend. The function `complex_function` is a computation that Conventional AARA cannot statically analyze. Conventional AARA fails to infer any polynomial cost bounds for the `map_append` function. (a) Fully data-driven resource analysis. (b) Hybrid resource analysis. We perform data-driven analysis on `step_function`.

Table A.2: Estimation gaps of inferred cost bounds for BubbleSort benchmark on various input sizes.

		Relative Gap in Inferred Cost Bound					
		Data-Driven			Hybrid		
Input Size	Percentile **Method**	5^{th}	50^{th}	95^{th}	5^{th}	50^{th}	95^{th}
10	Opt	0.01	0.01	0.01	∅	∅	∅
	BayesWC	0.44	6.29	60.73	∅	∅	∅
	BayesPC	-0.31	0.02	0.39	∅	∅	∅
100	Opt	-0.38	-0.38	-0.38	∅	∅	∅
	BayesWC	-0.48	0.41	8.34	∅	∅	∅
	BayesPC	-0.34	-0.10	0.17	∅	∅	∅
1000	Opt	-0.38	-0.38	-0.38	∅	∅	∅
	BayesWC	-0.93	-0.22	5.31	∅	∅	∅
	BayesPC	-0.35	-0.10	0.15	∅	∅	∅



(a) Opt (b) BayesWC (c) BayesPC

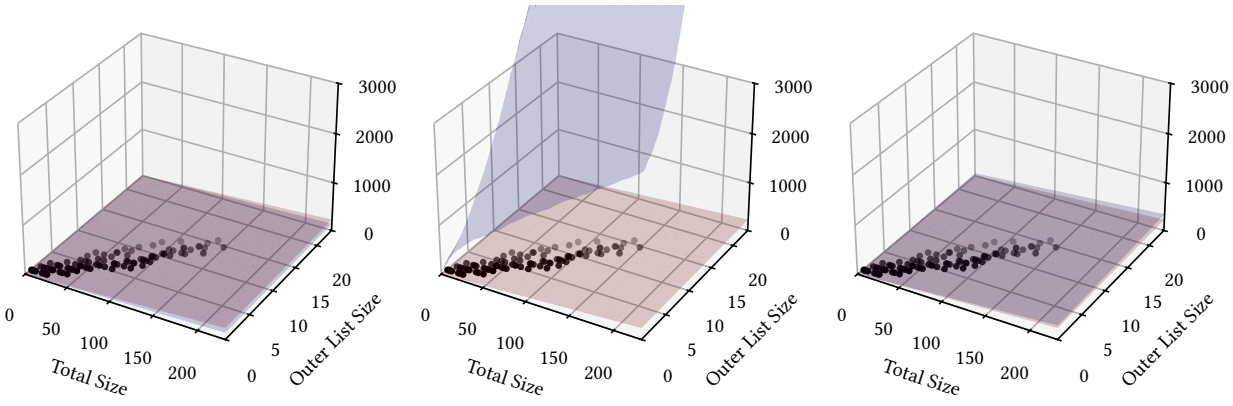
Figure A.3: BubbleSort Data-Driven

```
let incur_cost (hd : int) =
  if (hd mod 10) = 0 then Raml.tick 1.0 else Raml.tick 0.5

let rec scan_and_swap (xs : int list) =
  match xs with
  | [] → ([], false)
  | [ x ] → ([ x ], false)
  | x1 :: x2 :: tl →
      let _ = incur_cost x1 in
      if x1 <= x2 then
        let recursive_result, is_swapped = scan_and_swap (x2 :: tl) in
        (x1 :: recursive_result, is_swapped)
      else
        let recursive_result, _ = scan_and_swap (x1 :: tl) in
        (x2 :: recursive_result, true)

let rec bubble_sort (xs : int list) =
  let xs_scanned, is_swapped = scan_and_swap xs in
  if is_swapped then bubble_sort xs_scanned else xs_scanned

let bubble_sort2 (xs : int list) = Raml.stat (bubble_sort xs)
```

Lst. A.2: Source code of `BubbleSort` for fully data-driven analysis. Conventional AARA cannot infer any polynomial cost bound as it fails to bound the number of recursive calls.

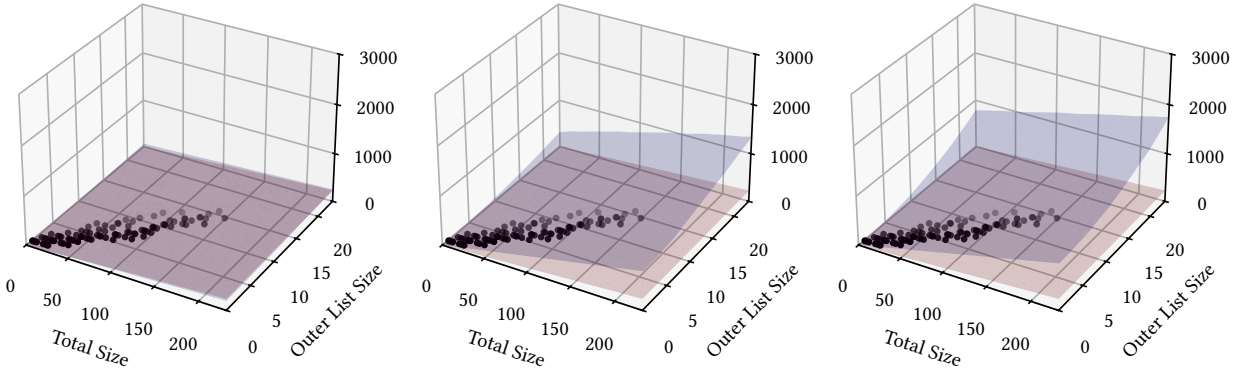
Table A.3: Estimation gaps of inferred cost bounds for Concat benchmark on various input sizes.

		Relative Gap in Inferred Cost Bound					
		Data-Driven			Hybrid		
	Percentile	5^{th}	50^{th}	95^{th}	5^{th}	50^{th}	95^{th}
Input Size	**Method**						
(50, 10)	Opt	-0.33	-0.33	-0.33	0.03	0.03	0.03
	BayesWC	14.05	66.64	744.65	1.74	4.80	19.86
	BayesPC	0.37	0.60	0.90	4.46	5.90	7.19
(500, 100)	Opt	-0.10	-0.10	-0.10	2.07	2.07	2.07
	BayesWC	12.54	183.95	3329.73	2.10	13.59	130.41
	BayesPC	0.50	1.25	4.50	16.22	32.27	47.71
(5000, 1000)	Opt	2.83	2.83	2.83	22.44	22.44	22.44
	BayesWC	11.04	931.52	32459.92	2.33	97.00	1309.28
	BayesPC	1.06	7.84	42.44	132.48	298.20	456.99



(a) Opt (b) BayesWC (c) BayesPC

Figure A.4: Concat Data-Driven



(a) Opt (b) BayesWC (c) BayesPC

Figure A.5: Concat Hybrid

```
let incur_cost (hd : int) =
  if (hd mod 5) = 0 then Raml.tick 1.0 else Raml.tick
      0.5

let complex_function (hd : int) =
  let _ = incur_cost hd in
  if hd < 42 then hd / 2 else hd * 2

let rec map_append (xs : int list) (ys : int list) =
  match xs with
  | [] → ys
  | hd :: tl →
      let hd_new = complex_function hd in
      hd_new :: map_append tl ys

let rec concat (xss : int list list) =
  match xss with [] →[] | hd :: tl →map_append hd (
      concat tl)

let concat2 (xss : int list list) = Raml.stat (concat
     xss)
```

(a) Fully data-driven resource analysis.

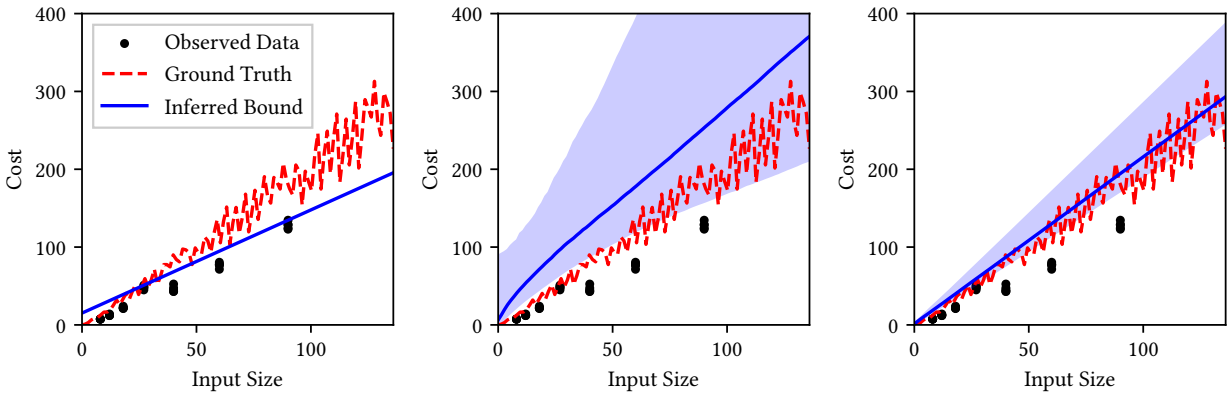
```
let rec concat (xss : int list list) =
  match xss with
  | [] → []
  | hd :: tl →
      let rec_tl = concat tl in
      Raml.stat (map_append hd rec_tl)
```

(b) Hybrid resource analysis.

Lst. A.3: Source code of Concat. The function `complex_function` is a computation that Conventional AARA cannot statically analyze. Conventional AARA fails to infer any polynomial cost bounds for the `concat` function. (a) Fully data-driven resource analysis. (b) Hybrid resource analysis. We perform data-driven analysis on `map_append`.

Table A.4: Estimation gaps of inferred cost bounds for EvenOddTail benchmark on various input sizes.

		Relative Gap in Inferred Cost Bound					
		Data-Driven			Hybrid		
	Percentile	5th	50th	95th	5th	50th	95th
Input Size	**Method**						
10	Opt	0.73	0.73	0.73	∅	∅	∅
	BayesWC	0.53	1.88	9.15	∅	∅	∅
	BayesPC	0.17	0.38	1.00	∅	∅	∅
100	Opt	-0.14	-0.14	-0.14	∅	∅	∅
	BayesWC	-0.08	0.62	3.80	∅	∅	∅
	BayesPC	0.10	0.25	0.90	∅	∅	∅
1000	Opt	-0.21	-0.21	-0.21	∅	∅	∅
	BayesWC	-0.62	0.52	3.75	∅	∅	∅
	BayesPC	0.11	0.27	0.92	∅	∅	∅



(a) Opt (b) BayesWC (c) BayesPC

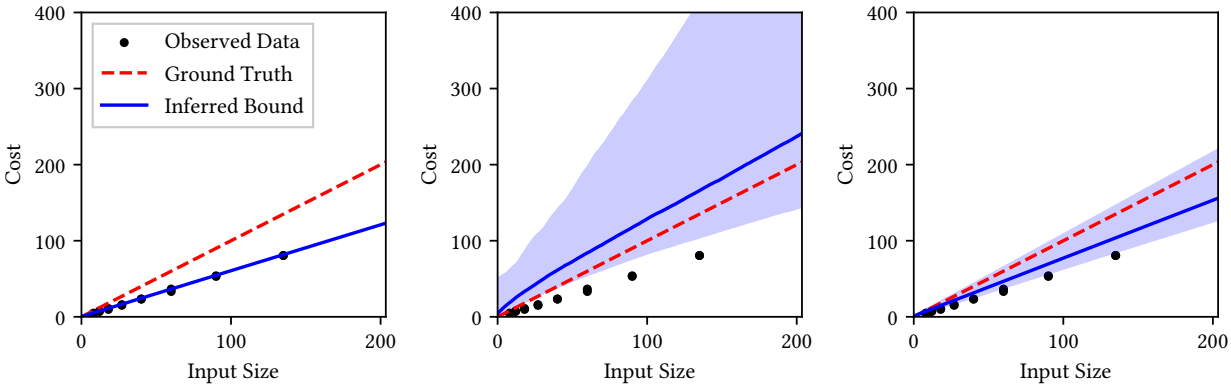
Figure A.6: EvenOddTail Data-Driven

```
exception Invalid_input

let incur_cost (hd : int) =
  if (hd mod 10) = 0 then Raml.tick 1.0 else Raml.tick 0.5

let rec linear_traversal (xs : int list) =
  match xs with
  | [] → []
  | hd :: tl →
      let _ = incur_cost hd in
      hd :: linear_traversal tl

let rec is_even (xs : int list) =
  match xs with [] →true | [ x ] →false | x1 :: x2 :: tl →is_even tl

let tail (xs : int list) =
  match xs with [] →raise Invalid_input | hd :: tl →tl

let rec split (xs : int list) =
  match xs with
  | [] → []
  | [ x ] → raise Invalid_input
  | x1 :: x2 :: tl →x1 :: split tl

let rec even_split_odd_tail (xs : int list) : int list =
  let xs_traversed = linear_traversal xs in
  match xs_traversed with
  | [] → []
  | hd :: tl →
      let xs_is_even = is_even xs_traversed in
      if xs_is_even then
        let split_result = split xs_traversed in
        even_split_odd_tail split_result
      else
        let tail_result = tail xs_traversed in
        even_split_odd_tail tail_result

let even_split_odd_tail2 (xs : int list) : int list =
  Raml.stat (even_split_odd_tail xs)
```

Lst. A.4: Source code of EvenOddTail for fully data-driven resource analysis. Conventional AARA can only infer a quadratic cost bound for EvenOddTail, but not the true linear cost bound.

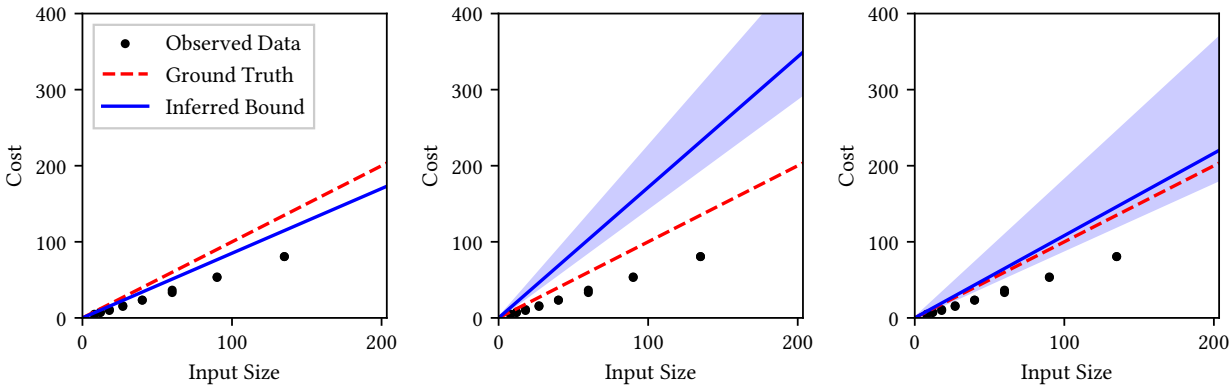
Table A.5: Estimation gaps of inferred cost bounds for InsertionSort2 benchmark on various input sizes.

		Relative Gap in Inferred Cost Bound					
		Data-Driven			Hybrid		
Input Size	Percentile **Method**	5^{th}	50^{th}	95^{th}	5^{th}	50^{th}	95^{th}
10	Opt	-0.37	-0.37	-0.37	-0.15	-0.15	-0.15
	BayesWC	0.05	1.17	8.68	0.39	0.72	1.47
	BayesPC	-0.33	-0.12	0.35	-0.14	0.08	0.84
100	Opt	-0.39	-0.39	-0.39	-0.15	-0.15	-0.15
	BayesWC	-0.23	0.29	3.58	0.39	0.72	1.47
	BayesPC	-0.39	-0.23	0.26	-0.14	0.08	0.84
1000	Opt	-0.40	-0.40	-0.40	-0.15	-0.15	-0.15
	BayesWC	-0.57	0.14	3.33	0.39	0.72	1.47
	BayesPC	-0.40	-0.24	0.25	-0.14	0.08	0.84



(a) Opt (b) BayesWC (c) BayesPC

Figure A.7: InsertionSort2 Data-Driven



(a) Opt (b) BayesWC (c) BayesPC

Figure A.8: InsertionSort2 Hybrid

```
let incur_cost (hd : int) =
  let modulo = 5 in
  if (hd mod 200) = 0 then Raml.tick 1.0
  else (if (hd mod modulo) = 1 then Raml.tick 0.85
        else if (hd mod modulo) = 2 then Raml.tick 0.65
             else Raml.tick 0.5)

let rec insert (x : int) (xs : int list) =
  match xs with
  | [] → [ x ]
  | hd :: tl →
      let _ = incur_cost hd in
      if x <= hd then x :: hd :: tl else hd :: insert x
           tl

let rec insertion_sort (xs : int list) =
  match xs with [] →[] | hd :: tl →insert hd (
       insertion_sort tl)

let rec insertion_sort_second_time (xs : int list) =
  match xs with
  | [] → []
  | hd :: tl → insert hd (insertion_sort_second_time tl)

let insertion_sort_second_time2 (xs : int list) =
  Raml.stat (insertion_sort_second_time xs)

let double_insertion_sort (xs : int list) =
  let sorted_xs = insertion_sort xs in
  Raml.stat (insertion_sort_second_time sorted_xs)
```

(a) Fully data-driven resource analysis.

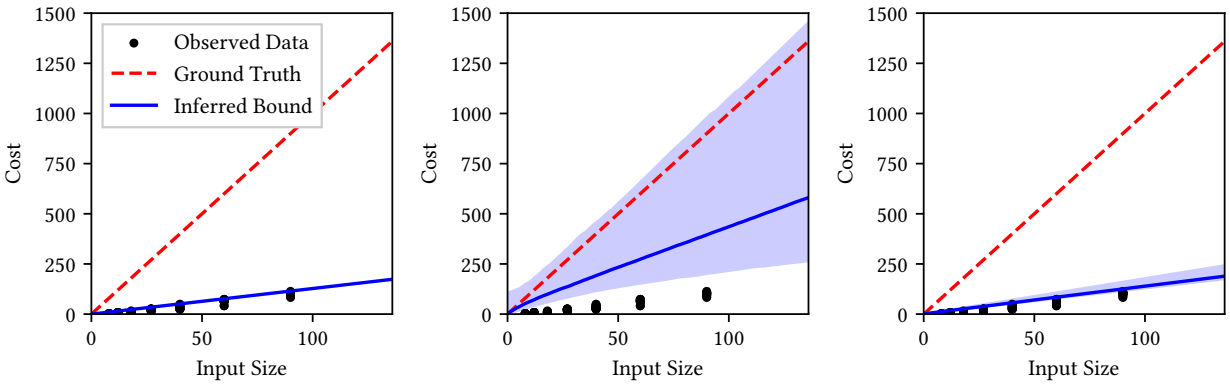
```
let rec insert (x : int) (xs : int list) =
  match xs with
  | [] → [ x ]
  | hd :: tl →
      let _ = incur_cost hd in
      if x <= hd then x :: hd :: tl else hd :: insert x
           tl

let rec insertion_sort (xs : int list) =
  match xs with [] →[] | hd :: tl →insert hd (
       insertion_sort tl)

let rec insert_second_time (x : int) (xs : int list) =
  match xs with
  | [] → [ x ]
  | hd :: tl →
      let _ = incur_cost hd in
      if x <= hd then x :: hd :: tl
      else hd :: (insert_second_time x tl)

let rec insertion_sort_second_time (xs : int list) =
  match xs with
  | [] → []
  | hd :: tl →
      let rec_result = insertion_sort_second_time tl in
      Raml.stat (insert_second_time hd rec_result)

let double_insertion_sort (xs : int list) =
  let sorted_xs = insertion_sort xs in
  insertion_sort_second_time sorted_xs
```

(b) Hybrid resource analysis.

Lst. A.5: Source code of InsertionSort2. The `insertion_sort` procedure is called twice in a row. Our goal is to analyze the cost of the second call to insertion sort, which has a linear worst-case cost bound because inputs are already sorted. Conventional AARA can only infer a quadratic bound for the second call to insertion sort (just like the first call to insertion sort), but not a linear cost bound, because it cannot determine that the inputs to the second insertion sort are already sorted. (a) Fully data-driven resource analysis. (b) Hybrid resource analysis. We perform data-driven analysis on the function `insert_second_time`.

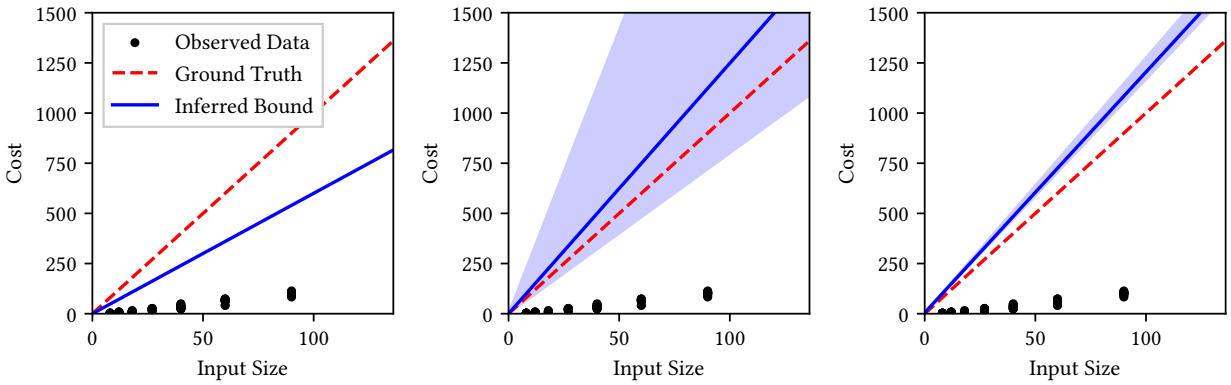
Table A.6: Estimation gaps of inferred cost bounds for MedianOfMedians benchmark on various input sizes.

		Relative Gap in Inferred Cost Bound					
		Data-Driven			Hybrid		
Input Size	Percentile **Method**	5^{th}	50^{th}	95^{th}	5^{th}	50^{th}	95^{th}
10	Opt	-0.42	-0.42	-0.42	-0.39	-0.39	-0.39
	BayesWC	-0.29	0.60	5.20	19.69	85.53	709.77
	BayesPC	-0.64	-0.55	-0.34	1.41	1.48	1.52
100	Opt	-0.95	-0.95	-0.95	-0.49	-0.49	-0.49
	BayesWC	-0.95	-0.89	-0.62	8.35	40.30	339.77
	BayesPC	-0.91	-0.80	-0.54	1.38	1.45	1.50
1000	Opt	-0.99	-0.99	-0.99	-0.50	-0.50	-0.50
	BayesWC	-1.00	-0.99	-0.82	2.48	31.90	328.10
	BayesPC	-0.94	-0.81	-0.55	1.38	1.45	1.50



(a) Opt (b) BayesWC (c) BayesPC

Figure A.9: MedianOfMedians Data-Driven



(a) Opt (b) BayesWC (c) BayesPC

Figure A.10: MedianOfMedians Hybrid

```
exception Invalid_input

let incur_cost (hd : int) =
  if (hd mod 10) = 0 then Raml.tick 1.0 else Raml.tick
       0.5

let rec append (xs : int list) (ys : int list) =
  match xs with [] →ys | hd :: tl →hd :: append tl ys

let rec insert (x : int) (list : int list) =
  match list with
  | [] → [ x ]
  | y :: ys → if x <= y then x :: y :: ys else y ::
       insert x ys

let rec insertion_sort (list : int list) =
  match list with [] →[] | x :: xs →insert x (
       insertion_sort xs)

let median_of_list_of_five (xs : int list) =
  let sorted_xs = insertion_sort xs in
  match sorted_xs with
  | [ x1; x2; x3; x4; x5 ] →(x3, [ x1; x2; x4; x5 ])
  | _ → raise Invalid_input

let rec partition_into_blocks (xs : int list) =
  match xs with
  | [] → ([], [])
  | x1 :: x2 :: x3 :: x4 :: x5 :: tl →
      let median, leftover = median_of_list_of_five [ x1;
           x2; x3; x4; x5 ] in
      let list_medians, list_leftover =
           partition_into_blocks tl in
      (median :: list_medians, append leftover
           list_leftover)
  | _ → raise Invalid_input

let rec partition (pivot : int) (xs : int list) =
  match xs with
  | [] → ([], [])
  | hd :: tl →
      let lower_list, upper_list = partition pivot tl in
      let _ = incur_cost hd in
      if hd <= pivot then (hd :: lower_list, upper_list)
      else (lower_list, hd :: upper_list)

let rec lower_list_length_after_partition (pivot : int)
     (xs : int list) =
  match xs with
  | [] → 0
  | hd :: tl →
      let lower_list_length =
           lower_list_length_after_partition pivot tl in
      if hd <= pivot then lower_list_length + 1 else
           lower_list_length

let rec list_length (xs : int list) =
  match xs with [] →0 | hd :: tl →1 + list_length tl

let rec find_minimum_acc (acc : int list) (candidate :
     int) (xs : int list) =
  match xs with
  | [] → (candidate, acc)
  | hd :: tl →
      if hd < candidate then find_minimum_acc (candidate
           :: acc) hd tl
      else find_minimum_acc (hd :: acc) candidate tl

let find_minimum (xs : int list) =
  match xs with
  | [] → raise Invalid_input
  | hd :: tl → find_minimum_acc [] hd tl

let rec preprocess_list_acc (minima_acc : int list) (xs
     : int list) =
  let xs_length = list_length xs in
  if xs_length mod 5 = 0 then (minima_acc, xs)
  else
    let minimum, leftover = find_minimum xs in
    preprocess_list_acc (minimum :: minima_acc) leftover

let rec get_nth_element (index : int) (xs : int list) =
  match xs with
  | [] → raise Invalid_input
  | hd :: tl → if index = 0 then hd else get_nth_element
       (index - 1) tl
```

Lst. A.6: Source code of helper functions used in MedianOfMedians (Listing A.7).

```
let rec median_of_medians (index : int) (xs : int list)
    =
  match xs with
  | [] → raise Invalid_input
  | _ →
      let minima, xs_trimmed = preprocess_list_acc [] xs
          in
      let mod_five = list_length minima in
      if index < mod_five then get_nth_element (mod_five
          - index - 1) minima
      else
        let index_trimmed = index - mod_five in
        let list_medians, _ = partition_into_blocks
            xs_trimmed in
        let num_medians = list_length list_medians in
        let index_median = num_medians / 2 in
        let median_of_medians =
          Raml.stat (median_of_medians index_median
              list_medians)
        in
        let lower_list_length =
          lower_list_length_after_partition
              median_of_medians xs_trimmed
        in
        if index_trimmed = lower_list_length - 1 then
          let _, _ = partition median_of_medians
              xs_trimmed in
          median_of_medians
        else if index_trimmed < lower_list_length - 1
            then
          let lower_list, _ = partition median_of_medians
              xs_trimmed in
          Raml.stat (median_of_medians index_trimmed
              lower_list)
        else
          let new_index = index_trimmed -
              lower_list_length in
          let _, upper_list = partition median_of_medians
              xs_trimmed in
          Raml.stat (median_of_medians new_index
              upper_list)

let median_of_medians2 (index : int) (xs : int list) =
  Raml.stat (median_of_medians index xs)
```

(a) Fully data-driven resource analysis.

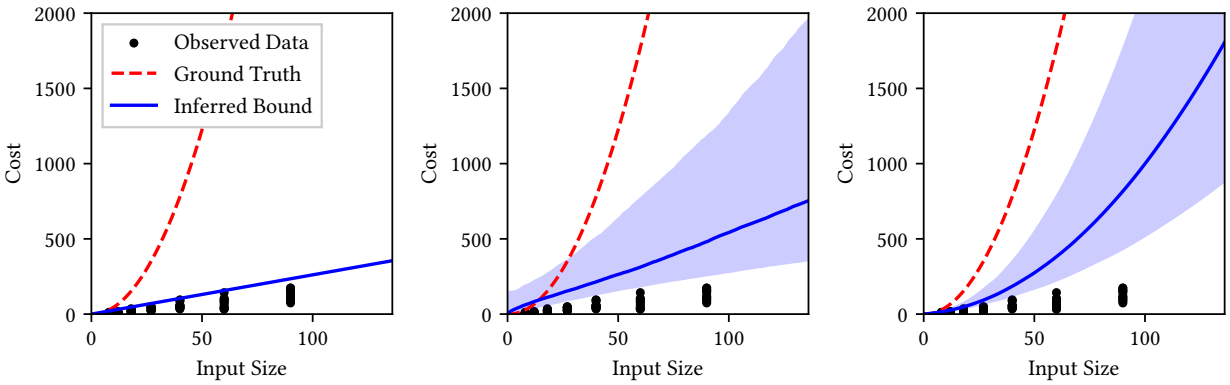
```
let rec median_of_medians (index : int) (xs : int list)
    =
  match xs with
  | [] → raise Invalid_input
  | _ →
      let minima, xs_trimmed = preprocess_list_acc [] xs
          in
      let mod_five = list_length minima in
      if index < mod_five then get_nth_element (mod_five
          - index - 1) minima
      else
        let index_trimmed = index - mod_five in
        let list_medians = partition_into_blocks
            xs_trimmed in
        let num_medians = list_length list_medians in
        let index_median = num_medians / 2 in
        let median_of_medians = median_of_medians
            index_median list_medians in
        let lower_list_length =
          lower_list_length_after_partition
              median_of_medians xs_trimmed
        in
        if index_trimmed = lower_list_length - 1 then
          let _, _ = Raml.stat (partition
              median_of_medians xs_trimmed) in
          median_of_medians
        else if index_trimmed < lower_list_length - 1
            then
          let lower_list, _ =
            Raml.stat (partition median_of_medians
                xs_trimmed)
          in
          median_of_medians index_trimmed lower_list
        else
          let new_index = index_trimmed -
              lower_list_length in
          let _, upper_list =
            Raml.stat (partition median_of_medians
                xs_trimmed)
          in
          median_of_medians new_index upper_list
```

(b) Hybrid resource analysis.

Lst. A.7: Source code of MedianOfMedians. Conventional AARA cannot infer any polynomial bound for MedianOfMedians. (a) Fully data-driven resource analysis. (b) Hybrid resource analysis. We conduct data-driven analysis on the function `partition`. Although Conventional AARA can derive a linear cost bound of the `partition` function, analyzing it using data-driven analysis gives a tighter cost bound. This tighter linear bound of the `partition` function is required for deriving an overall linear cost bound of MedianOfMedians.

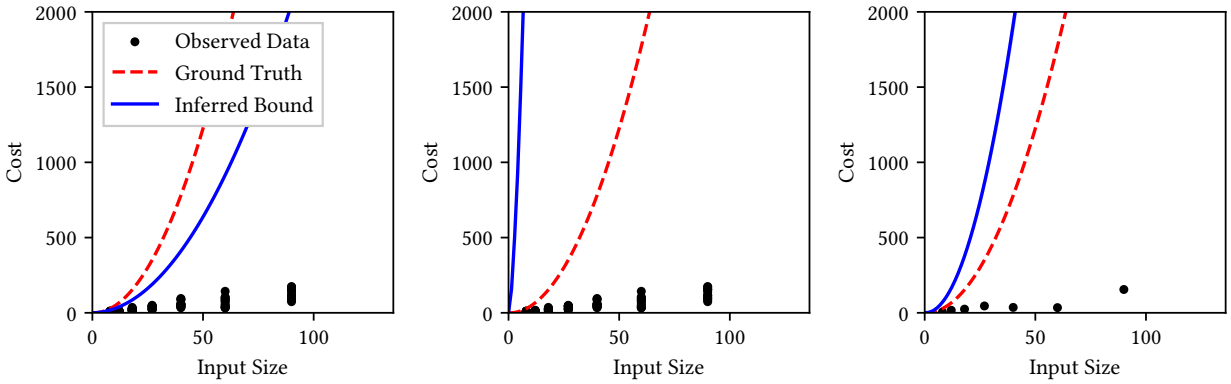
Table A.7: Estimation gaps of inferred cost bounds for QuickSelect benchmark on various input sizes.

		Relative Gap in Inferred Cost Bound					
		Data-Driven			Hybrid		
Input Size	Percentile **Method**	5^{th}	50^{th}	95^{th}	5^{th}	50^{th}	95^{th}
10	Opt	-0.42	-0.42	-0.42	-0.39	-0.39	-0.39
	BayesWC	-0.29	0.60	5.20	19.69	85.53	709.77
	BayesPC	-0.64	-0.55	-0.34	1.41	1.48	1.52
100	Opt	-0.95	-0.95	-0.95	-0.49	-0.49	-0.49
	BayesWC	-0.95	-0.89	-0.62	8.35	40.30	339.77
	BayesPC	-0.91	-0.80	-0.54	1.38	1.45	1.50
1000	Opt	-0.99	-0.99	-0.99	-0.50	-0.50	-0.50
	BayesWC	-1.00	-0.99	-0.82	2.48	31.90	328.10
	BayesPC	-0.94	-0.81	-0.55	1.38	1.45	1.50



(a) Opt (b) BayesWC (c) BayesPC

Figure A.11: QuickSelect Data-Driven



(a) Opt (b) BayesWC (c) BayesPC

Figure A.12: QuickSelect Hybrid

```
exception Invalid_input

let incur_cost (hd : int) =
  if (hd mod 10) = 0 then Raml.tick 1.0 else Raml.tick
       0.5

let rec append (xs : int list) (ys : int list) =
  match xs with [] →ys | hd :: tl →hd :: append tl ys

let rec partition (pivot : int) (xs : int list) =
  match xs with
  | [] → ([], [])
  | hd :: tl →
      let lower_list, upper_list = partition pivot tl in
      let _ = incur_cost hd in
      if hd <= pivot then (hd :: lower_list, upper_list)
      else (lower_list, hd :: upper_list)

let rec list_length (xs : int list) =
  match xs with [] →0 | hd :: tl →1 + list_length tl

let rec quickselect (index : int) (xs : int list) =
  match xs with
  | [] → raise Invalid_input
  | [ x ] → if index = 0 then x else raise
       Invalid_input
  | hd :: tl →
      let lower_list, upper_list = partition hd tl in
      let lower_list_length = list_length lower_list in
      if index < lower_list_length then quickselect index
            lower_list
      else if index = lower_list_length then hd
      else
        let new_index = index - lower_list_length - 1 in
        quickselect new_index upper_list

let quickselect2 (index : int) (xs : int list) =
  Raml.stat (quickselect index xs)
```

(a) Fully data-driven resource analysis.

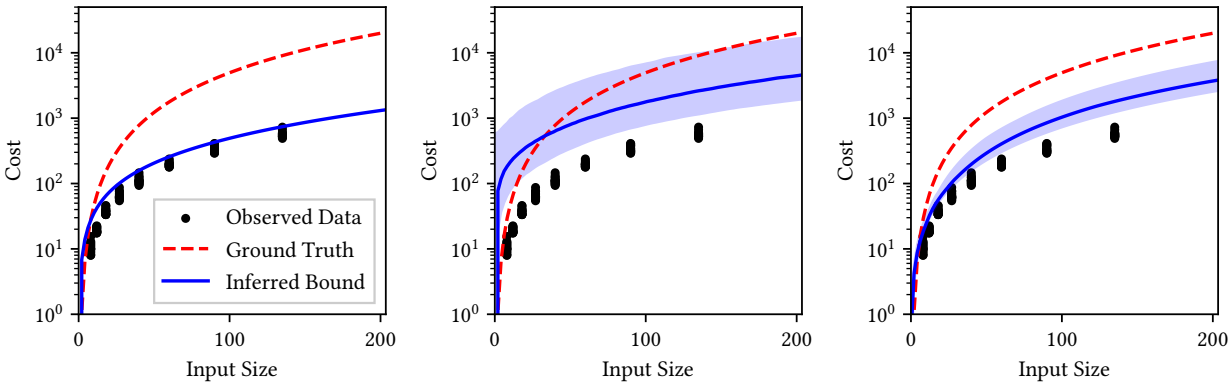
```
let rec quickselect (index : int) (xs : int list) =
  match xs with
  | [] → raise Invalid_input
  | [ x ] → if index = 0 then x else raise
       Invalid_input
  | hd :: tl →
      (* This is a workaround for an issue with the let-
           normal form inside
      Raml.stat(...) in the implementation *)
      let tl = tl in
      let lower_list, _ = partition_cost_free hd tl in
      let lower_list_length = list_length lower_list in
      if index < lower_list_length then
        let lower_list, _ = Raml.stat (partition hd tl)
             in
        quickselect index lower_list
      else if index = lower_list_length then
        let _, _ = Raml.stat (partition hd tl) in
        hd
      else
        let _, upper_list = Raml.stat (partition hd tl)
             in
        quickselect (index - lower_list_length - 1)
             upper_list
```

(b) Hybrid resource analysis.

Lst. A.8: Source code of QuickSelect. Conventional AARA cannot analyze this source code if the comparison function used inside QuickSelect is complex. (a) Fully data-driven resource analysis. (b) Hybrid resource analysis. We perform data-driven analysis on `partition`.

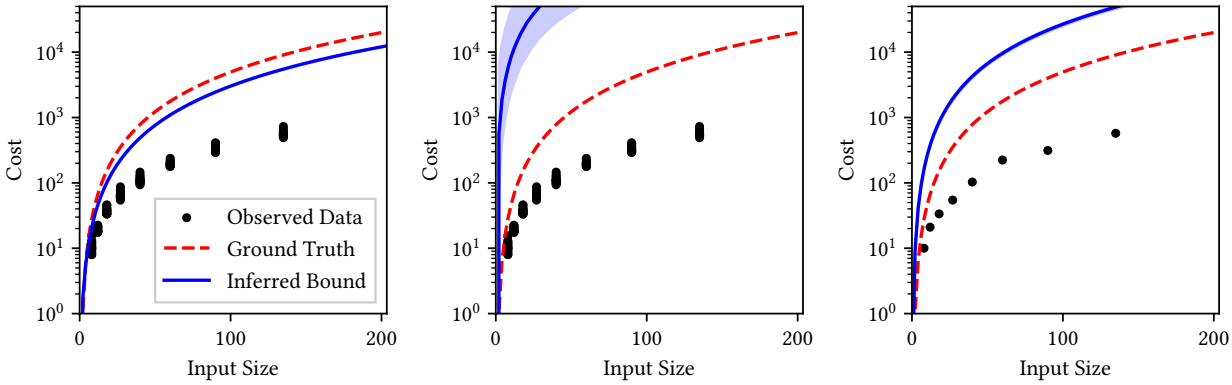
Table A.8: Estimation gaps of inferred cost bounds for QuickSort benchmark on various input sizes.

		Relative Gap in Inferred Cost Bound					
		Data-Driven			Hybrid		
Input Size	Percentile **Method**	5^{th}	50^{th}	95^{th}	5^{th}	50^{th}	95^{th}
10	Opt	-0.23	-0.23	-0.23	-0.29	-0.29	-0.29
	BayesWC	0.37	3.66	32.71	36.48	181.96	1776.52
	BayesPC	-0.52	-0.47	-0.22	4.12	4.73	4.96
100	Opt	-0.90	-0.90	-0.90	-0.39	-0.39	-0.39
	BayesWC	-0.87	-0.64	1.24	17.83	82.90	667.39
	BayesPC	-0.88	-0.79	-0.61	3.78	4.41	4.69
1000	Opt	-0.96	-0.96	-0.96	-0.40	-0.40	-0.40
	BayesWC	-0.98	-0.91	-0.09	5.07	60.66	610.58
	BayesPC	-0.93	-0.83	-0.63	3.75	4.38	4.66



(a) Opt (b) BayesWC (c) BayesPC

Figure A.13: QuickSort Data-Driven



(a) Opt (b) BayesWC (c) BayesPC

Figure A.14: QuickSort Hybrid

```
let incur_cost (hd : int) =
  if (hd mod 5) = 0 then Raml.tick 1.0 else Raml.tick
      0.5

let rec append (xs : int list) (ys : int list) =
  match xs with [] →ys | hd :: tl →hd :: append tl ys

let rec partition (pivot : int) (xs : int list) =
  match xs with
  | [] → ([], [])
  | hd :: tl →
      let lower_list, upper_list = partition pivot tl in
      let _ = incur_cost hd in
      if hd <= pivot then (hd :: lower_list, upper_list)
      else (lower_list, hd :: upper_list)

let rec quicksort (xs : int list) =
  match xs with
  | [] → []
  | hd :: tl →
      let lower_list, upper_list = partition hd tl in
      let lower_list_sorted = quicksort lower_list in
      let upper_list_sorted = quicksort upper_list in
      append lower_list_sorted (hd :: upper_list_sorted)

let quicksort2 (xs : int list) = Raml.stat (quicksort xs
    )
```

(a) Fully data-driven resource analysis.

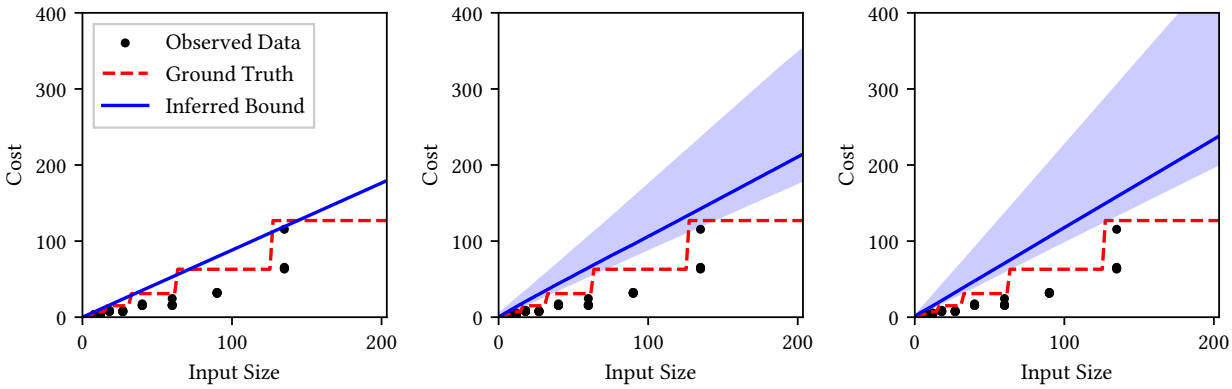
```
let rec quicksort (xs : int list) =
  match xs with
  | [] → []
  | hd :: tl →
      let lower_list, upper_list = Raml.stat (partition
          hd tl) in
      let lower_list_sorted = quicksort lower_list in
      let upper_list_sorted = quicksort upper_list in
      append lower_list_sorted (hd :: upper_list_sorted)
```

(b) Hybrid resource analysis.

Lst. A.9: Source code of `QuickSort`. Conventional AARA cannot analyze this source code if the comparison function used inside `QuickSort` is complex. (a) Fully data-driven resource analysis. (b) Hybrid resource analysis. We perform data-driven analysis on `partition`.

Table A.9: Estimation gaps of inferred cost bounds for Round benchmark on various input sizes.

		Relative Gap in Inferred Cost Bound					
		Data-Driven			Hybrid		
Input Size	Percentile **Method**	5^{th}	50^{th}	95^{th}	5^{th}	50^{th}	95^{th}
10	Opt	0.26	0.26	0.26	∅	∅	∅
	BayesWC	0.27	0.68	2.83	∅	∅	∅
	BayesPC	0.49	0.82	2.57	∅	∅	∅
100	Opt	0.40	0.40	0.40	∅	∅	∅
	BayesWC	0.40	0.68	2.33	∅	∅	∅
	BayesPC	0.55	0.87	2.86	∅	∅	∅
1000	Opt	0.73	0.73	0.73	∅	∅	∅
	BayesWC	0.67	1.06	3.11	∅	∅	∅
	BayesPC	0.89	1.29	3.75	∅	∅	∅



(a) Opt (b) BayesWC (c) BayesPC

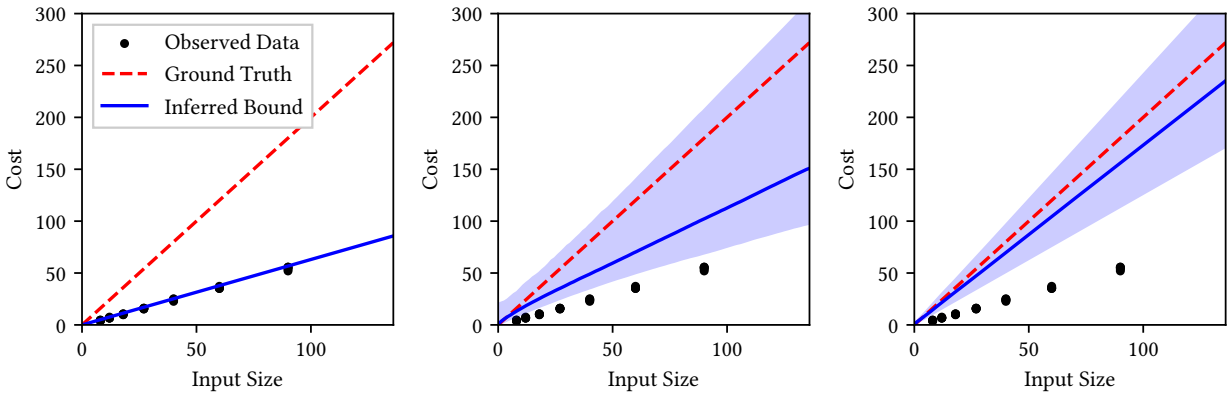
Figure A.15: Round Data-Driven

```
let incur_cost (hd : int) =
  if (hd mod 10) = 0 then Raml.tick 1.0 else Raml.tick 0.5

let rec double (xs : int list) =
  match xs with [] →[] | hd :: tl →hd :: hd :: double tl

let rec half (xs : int list) =
  match xs with [] →[] | [ x ] →[] | x1 :: x2 :: tl →x1 :: half tl

let rec round (xs : int list) =
  match xs with
  | [] → []
  | hd :: tl →
      let half_result = half tl in
      let recursive_result = round half_result in
      hd :: double recursive_result

let rec linear_traversal (xs : int list) =
  match xs with
  | [] → []
  | hd :: tl →
      let _ = incur_cost hd in
      hd :: linear_traversal tl

let round_followed_by_linear_traversal (xs : int list) =
  let round_result = round xs in
  linear_traversal round_result

let round2 (xs : int list) =
  Raml.stat (round_followed_by_linear_traversal xs)
```

Lst. A.10: Source code of Round for fully data-driven resource analysis. Conventional AARA cannot infer any polynomial cost bounds for this code [112, §5.4.3].

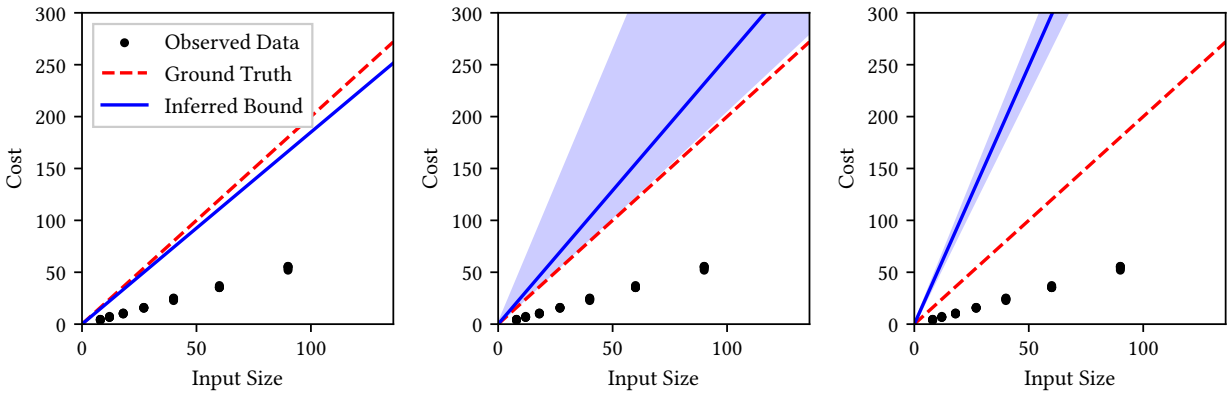
Table A.10: Estimation gaps of inferred cost bounds for ZAlgorithm benchmark on various input sizes.

		Relative Gap in Inferred Cost Bound					
		Data-Driven			Hybrid		
	Percentile	5th	50th	95th	5th	50th	95th
Input Size	**Method**						
10	Opt	-0.68	-0.68	-0.68	-0.08	-0.08	-0.08
	BayesWC	-0.53	-0.21	1.37	0.00	0.29	2.99
	BayesPC	-0.48	-0.10	0.33	1.18	1.49	1.78
100	Opt	-0.68	-0.68	-0.68	-0.08	-0.08	-0.08
	BayesWC	-0.65	-0.44	0.56	0.00	0.29	2.99
	BayesPC	-0.50	-0.13	0.23	1.18	1.49	1.78
1000	Opt	-0.68	-0.68	-0.68	-0.08	-0.08	-0.08
	BayesWC	-0.76	-0.47	0.56	0.00	0.29	2.99
	BayesPC	-0.50	-0.14	0.22	1.18	1.49	1.78



(a) Opt (b) BayesWC (c) BayesPC

Figure A.16: ZAlgorithm Data-Driven



(a) Opt (b) BayesWC (c) BayesPC

Figure A.17: ZAlgorithm Hybrid

```
exception Invalid_input

let incur_cost (hd : int) =
  let modulo = 5 in
  if (hd mod 100) = 0 then Raml.tick 1.0
  else (if (hd mod modulo) = 1 then Raml.tick 0.85
        else if (hd mod modulo) = 2 then Raml.tick 0.65 else Raml.tick 0.5)

let rec list_length (xs : int list) =
  match xs with [] →0 | hd :: tl →1 + list_length tl

let hd_exn (xs : int list) =
  match xs with [] →raise Invalid_input | hd :: _ →hd

let min (x1 : int) (x2 : int) = if x1 < x2 then x1 else x2

let rec drop_n_elements (xs : int list) (n : int) =
  match xs with
  | [] → []
  | hd :: tl → if n = 0 then hd :: tl else drop_n_elements tl (n - 1)

let rec longest_common_prefix (xs1 : int list) (xs2 : int list) =
  match xs1 with
  | [] → 0
  | hd1 :: tl1 →(
      match xs2 with
      | [] → 0
      | hd2 :: tl2 →
          if hd1 = hd2 then
            let _ = incur_cost (hd1 + hd2) in
            1 + longest_common_prefix tl1 tl2
          else 0 )
```

Lst. A.11: Source code of helper functions used in ZAlgorithm (Listing A.12).

```
let rec z_algorithm_acc (acc : int list) (
     original_string : int list)
    (current_string : int list) (left : int) (right :
         int) =
  match current_string with
  | [] → acc
  | hd :: tl →
      let _ = incur_cost hd
      in let current_index = list_length acc
      in let old_result =
        if left = 0 then 0 else hd_exn (drop_n_elements
             acc (left - 1))
      in let current_result_initial =
        if current_index < right then min (right -
             current_index) old_result
        else 0
      in let first_sublist =
        drop_n_elements original_string
             current_result_initial
      in let second_sublist =
        drop_n_elements current_string
             current_result_initial
      in let common_prefix_size =
        longest_common_prefix first_sublist
             second_sublist
      in let current_result = current_result_initial +
           common_prefix_size
      in let cumulative_result_updated = current_result
           :: acc
      in if current_index + current_result > right then
        z_algorithm_acc cumulative_result_updated
             original_string tl
          current_index
          (current_index + current_result)
        else
          z_algorithm_acc cumulative_result_updated
               original_string tl left right

let rec reverse_acc (acc : int list) (xs : int list) =
  match xs with [] →acc | hd :: tl →reverse_acc (hd ::
       acc) tl

let z_algorithm (xs : int list) =
  match xs with
  | [] → []
  | hd :: tl →
    reverse_acc []
      (z_algorithm_acc [ 0 ] xs tl 0 0)

let z_algorithm2 (xs : int list) = Raml.stat (
     z_algorithm xs)
```

(a) Fully data-driven resource analysis.

```
let rec z_algorithm_acc (acc : int list) (
     original_string : int list)
    (current_string : int list) (left : int) (right :
         int) =
  match current_string with
  | [] → acc
  | hd :: tl →
      let _ = incur_cost hd in
      let current_index = list_length acc in
      let old_result =
        if left = 0 then 0 else hd_exn (drop_n_elements
             acc (left - 1))
      in let current_result_initial =
        if current_index < right then min (right -
             current_index) old_result
        else 0
      in let first_sublist =
        drop_n_elements original_string
             current_result_initial
      in let second_sublist =
        drop_n_elements current_string
             current_result_initial
      in let common_prefix_size =
        Raml.stat (longest_common_prefix first_sublist
             second_sublist)
      in let current_result = current_result_initial +
           common_prefix_size
      in let cumulative_result_updated = current_result
           :: acc
      in if current_index + current_result > right then
        z_algorithm_acc cumulative_result_updated
             original_string tl
          current_index
          (current_index + current_result)
        else
          z_algorithm_acc cumulative_result_updated
            original_string tl left right

let rec reverse_acc (acc : int list) (xs : int list) =
  match xs with [] →acc | hd :: tl →reverse_acc (hd ::
       acc) tl

let z_algorithm (xs : int list) =
  match xs with
  | [] → []
  | hd :: tl →
    reverse_acc []
      (z_algorithm_acc [ 0 ] xs tl 0 0)
```

(b) Hybrid resource analysis.

Lst. A.12: Source code of ZAlgorithm. Conventional AARA can infer a quadratic cost bound, but not the true linear cost bound. (a) Fully data-driven resource analysis. (b) Hybrid resource analysis. We perform data-driven analysis on `longest_common_prefix`.

Appendix B

Supplements to Resource Decomposition

B.1 Supplementary Materials for Evaluation

This section describes and presents supplementary experiment results of the 14 benchmark programs that are analyzed using Bayesian data-driven analysis: (i) 13 benchmark programs for AARA+Bayesian analysis (§8.5); (ii) QuickSortTiML for TiML+Bayesian analysis (§8.7).

B.1.1 Benchmark Programs

The 14 benchmark programs used in §8.5 and §8.7 are described below.

- MergeSort: Run merge sort on an integer list.
- QuickSort: Run quicksort on an integer list. The first element is used as the partition pivot.
- QuickSortTiML: Run quicksort on a list of natural numbers. The comparison function first converts input numbers to binary encodings and then traverses them. The first element is used as the partition pivot.
- BubbleSort: Given an integer list, traverse it and swap any out-of-order pairs of elements. Repeat this traversal until no more swaps can be made. The worst-case linear recursion depth is not hard-coded in the implementation. Thus, this is a saturation-based algorithm: it keeps running until some data structure is *saturated* (e.g., no swaps can be made).
- HeapSort: Run heapsort on an integer list, using an array-based heap.
- HuffmanCode: Given a list of characters and their frequencies, construct a Huffman tree that represents the Huffman code. We implement a priority queue by an array-based heap.
- BalancedBST: Given integer lists x_1 and $x_2 \subseteq x_1$, first run merge sort on x_1 and then construct a balanced binary search tree (BST) from the sorted list (in linear time). Then successively look up the elements of x_2 in the BST.
- UnbalancedBST: Given integers lists x_1 and $x_2 \subseteq x_1$, create a (possibly unbalanced) BST for x_1. Then successively look up the elements of x_2 in the BST.

- RedBlackTree, AVLTree, SplayTree: Given integer lists x_1 and $x_2 \subseteq x_1$, construct the BST variant for x_1. Then successively look up the elements of x_2 in the BST.
- Prim: Given a graph represented by an adjacency list (i.e., an array of lists of neighboring vertices), run Prim's algorithm to compute a minimum spanning tree (MST). We implement a priority queue by an array-based heap.
- Dijkstra: Given a graph represented by an adjacency list, run Dijkstra's algorithm to compute shortest distances of all vertices from the source, which is the first vertex in the adjacency list. We implement a priority queue by an array-based heap.
- BellmanFord: Given a graph represented by an adjacency list, run the Bellman-Ford algorithm to compute shortest distances of all vertices from the source.

B.1.2 Full Experiment Results

Table B.1: Relative errors of inferred bounds for MergeSort with respect to the ground-truth bounds.

Quantity	Input Size	Relative Errors of Inferred Bounds		
		5^{th} percentile	50^{th} percentile	95^{th} percentile
Resource Guard 1	2^{10}	0.050	0.105	0.268
	2^{15}	0.024	0.096	0.283
	2^{20}	0.007	0.090	0.293
Total Cost	2^{10}	0.050	0.105	0.268
	2^{15}	0.024	0.096	0.283
	2^{20}	0.007	0.090	0.293

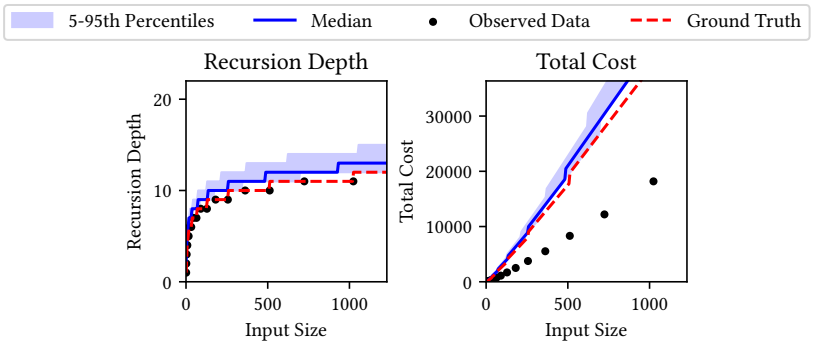


Figure B.1: Posterior distributions of resource guards and total cost in MergeSort.

Table B.2: Relative errors of inferred bounds for QuickSort with respect to the ground-truth bounds.

Quantity	Input Size	Relative Errors of Inferred Bounds		
		5^{th} percentile	50^{th} percentile	95^{th} percentile
Resource Guard 1	2^{10}	-0.228	-0.103	0.034
	2^{15}	-0.230	-0.104	0.033
	2^{20}	-0.230	-0.104	0.033
Total Cost	2^{10}	-0.228	-0.103	0.034
	2^{15}	-0.230	-0.104	0.033
	2^{20}	-0.230	-0.104	0.033

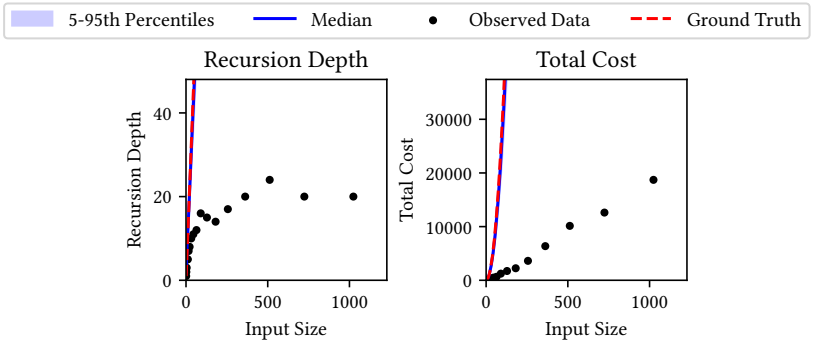


Figure B.2: Posterior distributions of resource guards and total cost in QuickSort.

Table B.3: Relative errors of inferred bounds for BubbleSort with respect to the ground-truth bounds.

Quantity	Input Size	Relative Errors of Inferred Bounds		
		5^{th} percentile	50^{th} percentile	95^{th} percentile
Resource Guard 1	2^{10}	-0.012	0.018	0.099
	2^{15}	-0.012	0.017	0.098
	2^{20}	-0.012	0.017	0.098
Total Cost	2^{10}	-0.012	0.018	0.099
	2^{15}	-0.012	0.017	0.098
	2^{20}	-0.012	0.017	0.098

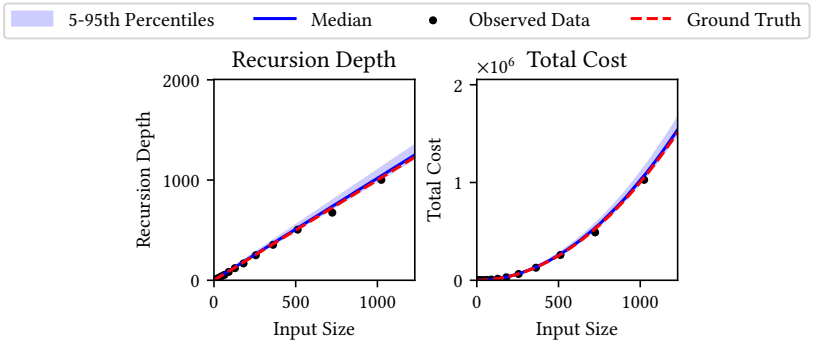


Figure B.3: Posterior distributions of resource guards and total cost in BubbleSort.

Table B.4: Relative errors of inferred bounds for HeapSort with respect to the ground-truth bounds.

Quantity	Input Size	Relative Errors of Inferred Bounds		
		5^{th} percentile	50^{th} percentile	95^{th} percentile
Resource Guard 1	$2^{10}-1$	0.008	0.077	0.264
	$2^{15}-1$	-0.040	0.051	0.286
	$2^{20}-1$	-0.067	0.038	0.299
Resource Guard 2	$2^{10}-1$	0.052	0.106	0.322
	$2^{15}-1$	0.009	0.091	0.349
	$2^{20}-1$	-0.015	0.083	0.367
Total Cost	$2^{10}-1$	0.035	0.084	0.214
	$2^{15}-1$	0.002	0.071	0.247
	$2^{20}-1$	-0.016	0.064	0.266

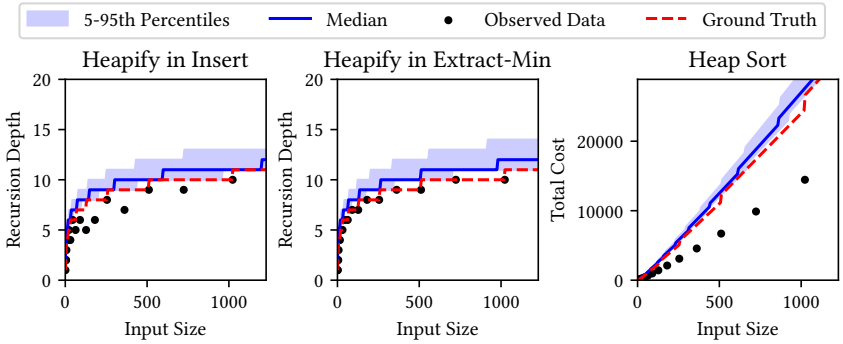


Figure B.4: Posterior distributions of resource guards and total cost in HeapSort.

Table B.5: Relative errors of inferred bounds for HuffmanCode with respect to the ground-truth bounds.

Quantity	Input Size	Relative Errors of Inferred Bounds		
		5^{th} percentile	50^{th} percentile	95^{th} percentile
Resource Guard 1	$2^{10}-1$	0.008	0.077	0.264
	$2^{15}-1$	-0.040	0.051	0.286
	$2^{20}-1$	-0.067	0.038	0.299
Resource Guard 2	$2^{10}-1$	0.051	0.104	0.276
	$2^{15}-1$	0.002	0.087	0.296
	$2^{20}-1$	-0.026	0.077	0.308
Total Cost	$2^{10}-1$	0.047	0.098	0.211
	$2^{15}-1$	0.010	0.081	0.220
	$2^{20}-1$	-0.012	0.072	0.229

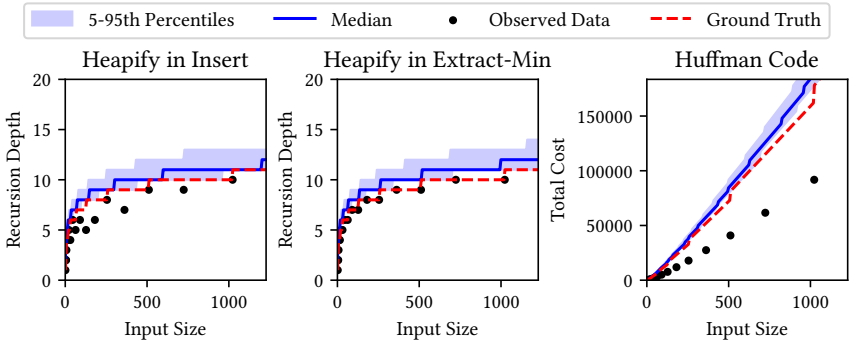


Figure B.5: Posterior distributions of resource guards and total cost in HuffmanCode.

Table B.6: Relative errors of inferred bounds for BalancedBST with respect to the ground-truth bounds.

Quantity	Input Size	Relative Errors of Inferred Bounds		
		5^{th} percentile	50^{th} percentile	95^{th} percentile
Counter 1	$(2^{10}-1, 2^{9})$	-0.004	0.019	0.074
	$(2^{15}-1, 2^{14})$	-0.015	0.017	0.083
	$(2^{20}-1, 2^{19})$	-0.021	0.016	0.088
Counter 2	$(2^{10}-1, 2^{9})$	0.001	0.040	0.124
	$(2^{15}-1, 2^{14})$	-0.006	0.048	0.153
	$(2^{20}-1, 2^{19})$	-0.010	0.053	0.168
Total Cost	$(2^{10}-1, 2^{9})$	0.001	0.022	0.069
	$(2^{15}-1, 2^{14})$	-0.007	0.021	0.079
	$(2^{20}-1, 2^{19})$	-0.012	0.021	0.084

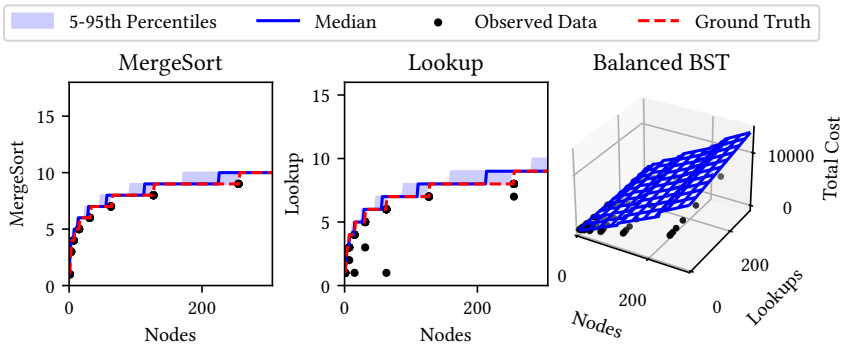


Figure B.6: Posterior distributions of resource guards and total cost in BalancedBST.

Table B.7: Relative errors of inferred bounds for UnbalancedBST with respect to the ground-truth bounds.

Quantity	Input Size	Relative Errors of Inferred Bounds		
		5^{th} percentile	50^{th} percentile	95^{th} percentile
Resource Guard 1	$(2^{10}-1, 2^{9})$	-0.288	-0.227	-0.133
	$(2^{15}-1, 2^{14})$	-0.289	-0.228	-0.134
	$(2^{20}-1, 2^{19})$	-0.289	-0.228	-0.134
Resource Guard 2	$(2^{10}-1, 2^{9})$	-0.294	-0.225	-0.122
	$(2^{15}-1, 2^{14})$	-0.295	-0.226	-0.122
	$(2^{20}-1, 2^{19})$	-0.295	-0.226	-0.122
Total Cost	$(2^{10}-1, 2^{9})$	-0.273	-0.221	-0.161
	$(2^{15}-1, 2^{14})$	-0.275	-0.222	-0.162
	$(2^{20}-1, 2^{19})$	-0.275	-0.222	-0.162

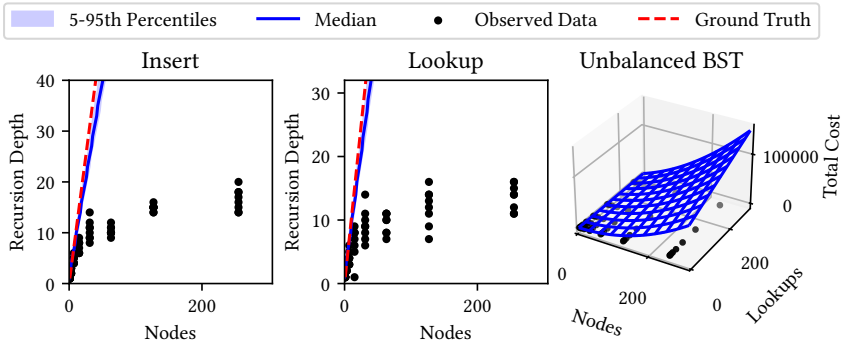


Figure B.7: Posterior distributions of resource guards and total cost in UnbalancedBST.

Table B.8: Relative errors of inferred bounds for RedBlackTree with respect to the ground-truth bounds.

Quantity	Input Size	Relative Errors of Inferred Bounds		
		5^{th} percentile	50^{th} percentile	95^{th} percentile
Resource Guard 1	$(2^{10}-1, 2^{9})$	-0.225	-0.184	-0.069
	$(2^{15}-1, 2^{14})$	-0.237	-0.187	-0.051
	$(2^{20}-1, 2^{19})$	-0.243	-0.188	-0.042
Resource Guard 2	$(2^{10}-1, 2^{9})$	-0.308	-0.259	-0.188
	$(2^{15}-1, 2^{14})$	-0.313	-0.247	-0.152
	$(2^{20}-1, 2^{19})$	-0.316	-0.241	-0.134
Total Cost	$(2^{10}-1, 2^{9})$	-0.227	-0.191	-0.095
	$(2^{15}-1, 2^{14})$	-0.237	-0.194	-0.080
	$(2^{20}-1, 2^{19})$	-0.242	-0.195	-0.072

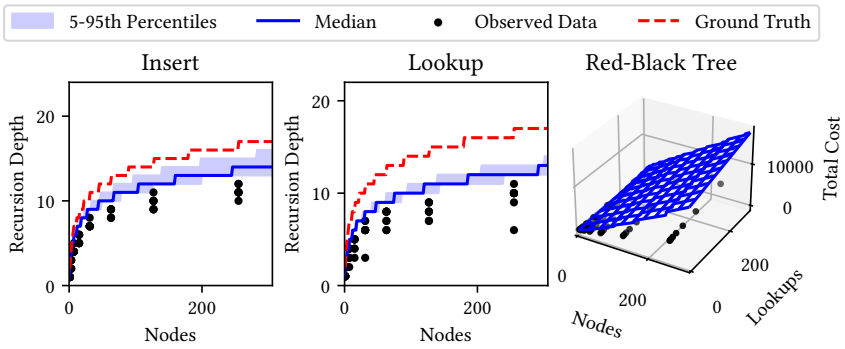


Figure B.8: Posterior distributions of resource guards and total cost in RedBlackTree.

Table B.9: Relative errors of inferred bounds for AVLTree with respect to the ground-truth bounds.

Quantity	Input Size	Relative Errors of Inferred Bounds		
		5^{th} percentile	50^{th} percentile	95^{th} percentile
Resource Guard 1	$(2^{10}-1, 2^{9})$	-0.160	-0.089	0.001
	$(2^{15}-1, 2^{14})$	-0.149	-0.057	0.060
	$(2^{20}-1, 2^{19})$	-0.143	-0.041	0.091
Resource Guard 2	$(2^{10}-1, 2^{9})$	-0.224	-0.207	-0.046
	$(2^{15}-1, 2^{14})$	-0.203	-0.178	0.041
	$(2^{20}-1, 2^{19})$	-0.192	-0.163	0.087
Total Cost	$(2^{10}-1, 2^{9})$	-0.160	-0.092	-0.007
	$(2^{15}-1, 2^{14})$	-0.149	-0.060	0.050
	$(2^{20}-1, 2^{19})$	-0.142	-0.043	0.081

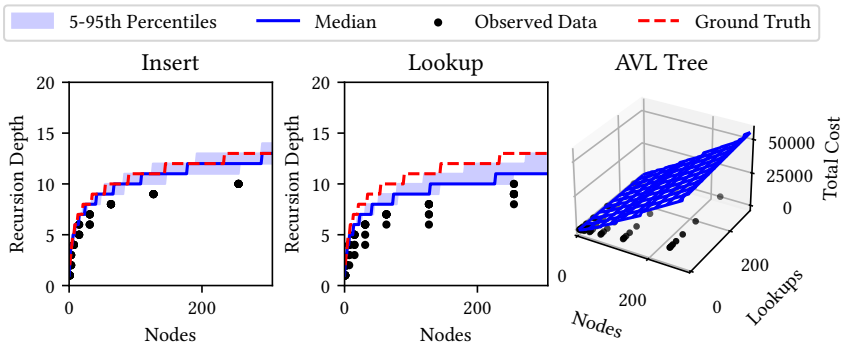


Figure B.9: Posterior distributions of resource guards and total cost in AVLTree.

Table B.10: Relative errors of inferred bounds for SplayTree with respect to the ground-truth bounds.

Quantity	Input Size	Relative Errors of Inferred Bounds		
		5^{th} percentile	50^{th} percentile	95^{th} percentile
Resource Guard 1	$(2^{10}-1, 2^{9})$	-0.726	-0.609	-0.514
	$(2^{15}-1, 2^{14})$	-0.728	-0.610	-0.514
	$(2^{20}-1, 2^{19})$	-0.728	-0.610	-0.514
Resource Guard 2	$(2^{10}-1, 2^{9})$	-0.707	-0.687	-0.658
	$(2^{15}-1, 2^{14})$	-0.710	-0.688	-0.659
	$(2^{20}-1, 2^{19})$	-0.710	-0.688	-0.659
Total Cost	$(2^{10}-1, 2^{9})$	-0.716	-0.638	-0.566
	$(2^{15}-1, 2^{14})$	-0.719	-0.640	-0.567
	$(2^{20}-1, 2^{19})$	-0.719	-0.640	-0.567

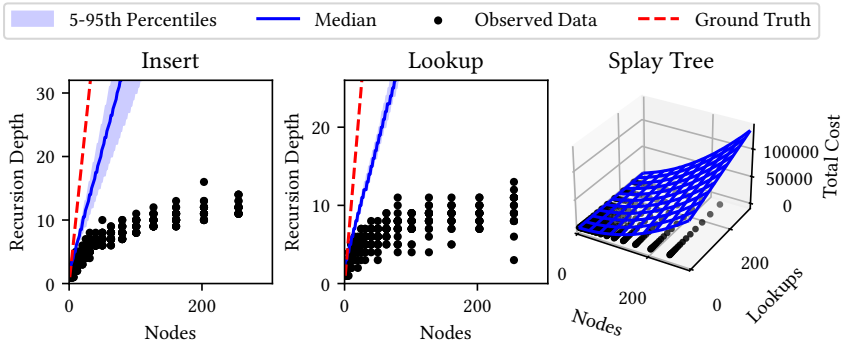


Figure B.10: Posterior distributions of resource guards and total cost in SplayTree.

Table B.11: Relative errors of inferred bounds for Prim with respect to the ground-truth bounds.

Quantity	Input Size	Relative Errors of Inferred Bounds		
		5^{th} percentile	50^{th} percentile	95^{th} percentile
Resource Guard 1	$(2^{10}, 2^{9})$	-0.042	-0.017	0.027
	$(2^{15}, 2^{14})$	-0.026	0.009	0.067
	$(2^{20}, 2^{19})$	-0.017	0.023	0.089
Resource Guard 2	$(2^{10}, 2^{9})$	-0.058	-0.030	0.013
	$(2^{15}, 2^{14})$	-0.059	-0.011	0.048
	$(2^{20}, 2^{19})$	-0.061	-0.001	0.067
Resource Guard 3	$(2^{10}, 2^{9})$	0.000	0.006	0.021
	$(2^{15}, 2^{14})$	-0.000	0.005	0.021
	$(2^{20}, 2^{19})$	-0.000	0.005	0.021
Total Cost	$(2^{10}, 2^{9})$	-0.048	-0.022	0.021
	$(2^{15}, 2^{14})$	-0.052	-0.005	0.054
	$(2^{20}, 2^{19})$	-0.053	0.005	0.072

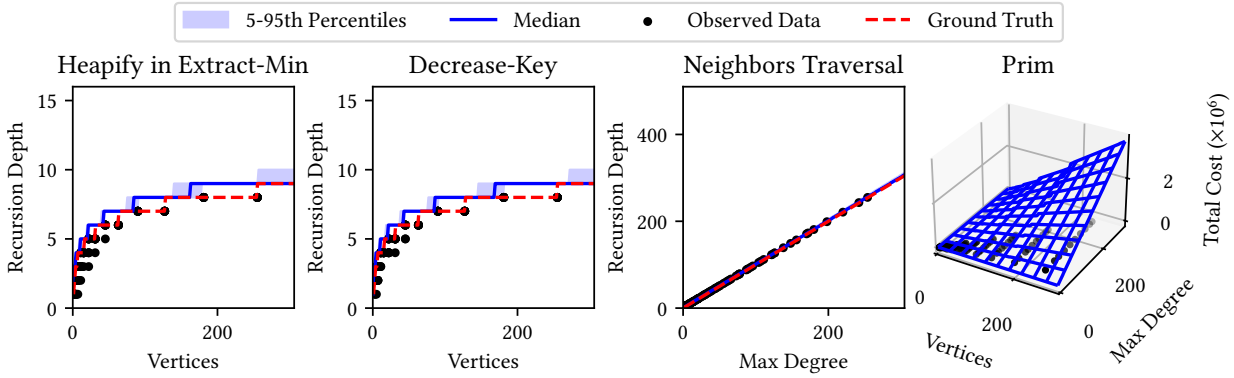


Figure B.11: Posterior distributions of resource guards and total cost in Prim.

Table B.12: Relative errors of inferred bounds for Dijkstra with respect to the ground-truth bounds.

Quantity	Input Size	Relative Errors of Inferred Bounds		
		5^{th} percentile	50^{th} percentile	95^{th} percentile
Resource Guard 1	$(2^{10}, 2^{9})$	-0.042	-0.017	0.027
	$(2^{15}, 2^{14})$	-0.026	0.009	0.067
	$(2^{20}, 2^{19})$	-0.017	0.023	0.089
Resource Guard 2	$(2^{10}, 2^{9})$	-0.045	-0.024	0.028
	$(2^{15}, 2^{14})$	-0.033	0.000	0.071
	$(2^{20}, 2^{19})$	-0.027	0.013	0.093
Resource Guard 3	$(2^{10}, 2^{9})$	0.000	0.006	0.021
	$(2^{15}, 2^{14})$	-0.000	0.005	0.021
	$(2^{20}, 2^{19})$	-0.000	0.005	0.021
Total Cost	$(2^{10}, 2^{9})$	-0.036	-0.015	0.032
	$(2^{15}, 2^{14})$	-0.026	0.007	0.073
	$(2^{20}, 2^{19})$	-0.020	0.019	0.096

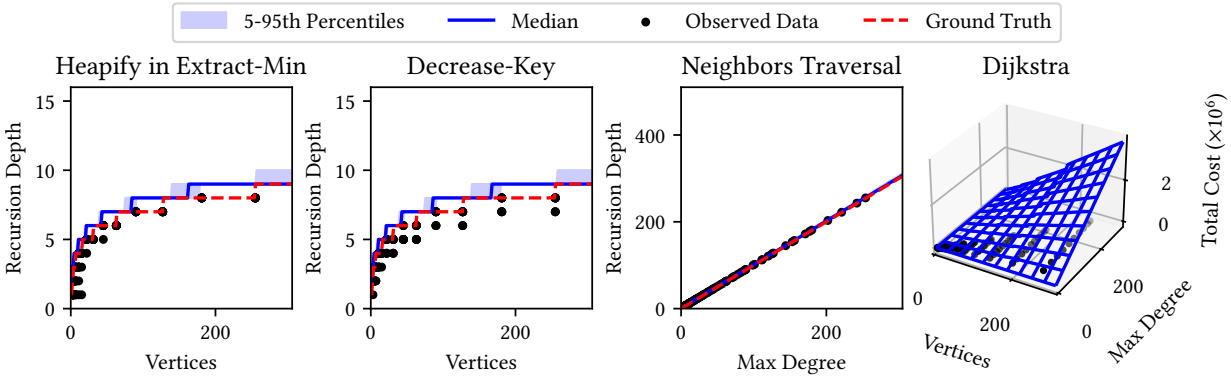


Figure B.12: Posterior distributions of resource guards and total cost in Dijkstra.

Table B.13: Relative errors of inferred bounds for BellmanFord with respect to the ground-truth bounds.

Quantity	Input Size	Relative Errors of Inferred Bounds		
		5^{th} percentile	50^{th} percentile	95^{th} percentile
Resource Guard 1	$(2^{10}, 2^{9})$	-0.003	0.002	0.021
	$(2^{15}, 2^{14})$	-0.004	0.001	0.020
	$(2^{20}, 2^{19})$	-0.004	0.001	0.020
Total Cost	$(2^{10}, 2^{9})$	-0.003	0.002	0.021
	$(2^{15}, 2^{14})$	-0.004	0.001	0.020
	$(2^{20}, 2^{19})$	-0.004	0.001	0.020

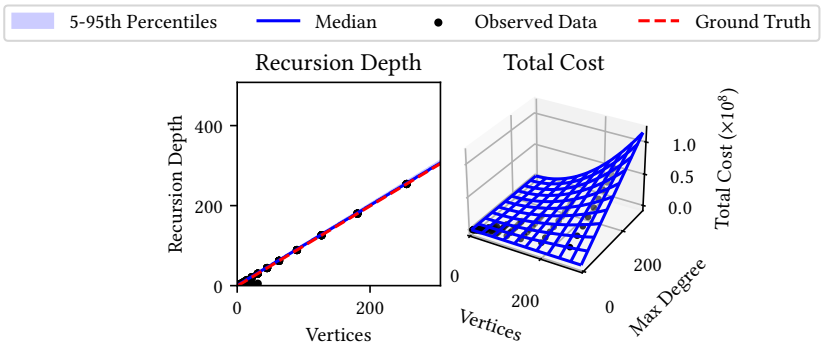


Figure B.13: Posterior distributions of resource guards and total cost in BellmanFord.

Table B.14: Relative errors of inferred bounds for QuickSortTiML with respect to the ground-truth bounds.

Quantity	Input Size	Relative Errors of Inferred Bounds		
		5^{th} percentile	50^{th} percentile	95^{th} percentile
Counter 1	$(2^{10}, 2^{9})$	0.001	0.008	0.028
	$(2^{15}, 2^{14})$	-0.008	0.008	0.039
	$(2^{20}, 2^{19})$	-0.017	0.009	0.048
Total Cost	$(2^{10}, 2^{9})$	0.001	0.006	0.023
	$(2^{15}, 2^{14})$	-0.007	0.007	0.034
	$(2^{20}, 2^{19})$	-0.016	0.008	0.044

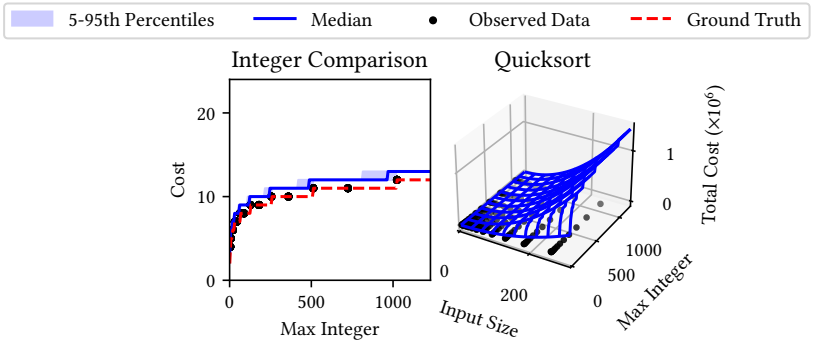


Figure B.14: Posterior distributions of resource guards and total cost in QuickSortTiML.

B.2 Source Code of Benchmark Programs

This section displays the source code of resource-decomposed and resource-guarded code of all 15 benchmark programs in §8.5.3: (i) 13 benchmarks for AARA+Bayesian analysis for AARA+Bayesian analysis(§8.5); (ii) Kruskal for AARA+Interactive analysis (§8.6); and (iii) QuickSortTiML for TiML+Bayesian analysis (§8.7).

B.2.1 Helper Functions for Resource Guards

```
exception Invalid_input

let decrement_counter current_original_counters =
  let current_counter, original_counter =
    current_original_counters
  in
  match current_counter with
  | [] → raise Invalid_input
  | _ :: counter_tl →(counter_tl, original_counter)

let increment_counter current_original_counters =
  let current_counter, original_counter =
    current_original_counters
  in
  (1 :: current_counter, original_counter)

let initialize_counter current_original_counters =
  let _, original_counter = current_original_counters in
  (original_counter, original_counter)

let set_counter_to_zero current_original_counters =
  let _, original_counter = current_original_counters in
  ([], original_counter)
```

Lst. B.1: Helper functions for manipulating one resource guard. These functions are used in the source code of those benchmarks that use one resource guard (e.g., MergeSort).

```
exception Invalid_input

let decrement_counter current_original_counters =
  let current_counter, original_counter =
    current_original_counters
  in
  match current_counter with
  | [] → raise Invalid_input
  | _ :: counter_tl →(counter_tl, original_counter)

let decrement_first_counter
    two_current_original_counters =
  let ( current_original_counters1,
        current_original_counters2 ) =
    two_current_original_counters
  in
  ( decrement_counter current_original_counters1,
    current_original_counters2 )

let decrement_second_counter
    two_current_original_counters =
  let ( current_original_counters1,
        current_original_counters2 ) =
    two_current_original_counters
  in
  ( current_original_counters1,
    decrement_counter current_original_counters2 )

let increment_counter current_original_counters =
  let current_counter, original_counter =
    current_original_counters
  in
  (1 :: current_counter, original_counter)

let increment_first_counter
    two_current_original_counters =
  let ( current_original_counters1,
        current_original_counters2 ) =
    two_current_original_counters
  in
  ( increment_counter current_original_counters1,
    current_original_counters2 )

let increment_second_counter
    two_current_original_counters =
  let ( current_original_counters1,
        current_original_counters2 ) =
    two_current_original_counters
  in
  ( current_original_counters1,
    increment_counter current_original_counters2 )

let initialize_counter current_original_counters =
  let _, original_counter =
    current_original_counters
  in
  (original_counter, original_counter)

let initialize_first_counter
    two_current_original_counters =
  let ( current_original_counters1,
        current_original_counters2 ) =
    two_current_original_counters
  in
  ( initialize_counter current_original_counters1,
    current_original_counters2 )

let initialize_second_counter
    two_current_original_counters =
  let ( current_original_counters1,
        current_original_counters2 ) =
    two_current_original_counters
  in
  ( current_original_counters1,
    initialize_counter current_original_counters2 )

let set_counter_to_zero current_original_counters =
  let _, original_counter =
    current_original_counters
  in
  ([], original_counter)

let set_first_counter_to_zero
    two_current_original_counters =
  let ( current_original_counters1,
        current_original_counters2 ) =
    two_current_original_counters
  in
  ( set_counter_to_zero current_original_counters1,
    current_original_counters2 )

let set_second_counter_to_zero
    two_current_original_counters =
  let ( current_original_counters1,
        current_original_counters2 ) =
    two_current_original_counters
  in
  ( current_original_counters1,
    set_counter_to_zero current_original_counters2 )
```

Lst. B.2: Helper functions for manipulating two resource guards. These functions are used in the source code of those benchmarks that use two resource guards (e.g., HeapSort).

```
exception Invalid_input

let decrement_counter current_original_counters =
  let current_counter, original_counter =
    current_original_counters
  in
  match current_counter with
  | [] → raise Invalid_input
  | _ :: counter_tl →(counter_tl, original_counter)

let decrement_first_counter
    three_current_original_counters =
  let ( current_original_counters1,
        current_original_counters2,
        current_original_counters3 ) =
    three_current_original_counters
  in
  ( decrement_counter current_original_counters1,
    current_original_counters2,
    current_original_counters3 )

let decrement_second_counter
    three_current_original_counters =
  let ( current_original_counters1,
        current_original_counters2,
        current_original_counters3 ) =
    three_current_original_counters
  in
  ( current_original_counters1,
    decrement_counter current_original_counters2,
    current_original_counters3 )

let decrement_third_counter
    three_current_original_counters =
  let ( current_original_counters1,
        current_original_counters2,
        current_original_counters3 ) =
    three_current_original_counters
  in
  ( current_original_counters1,
    current_original_counters2,
    decrement_counter current_original_counters3 )

let increment_counter current_original_counters =
  let current_counter, original_counter =
    current_original_counters
  in
  (1 :: current_counter, original_counter)

let increment_first_counter
    three_current_original_counters =
  let ( current_original_counters1,
        current_original_counters2,
        current_original_counters3 ) =
    three_current_original_counters
  in
  ( increment_counter current_original_counters1,
    current_original_counters2,
    current_original_counters3 )

let increment_second_counter
    three_current_original_counters =
  let ( current_original_counters1,
        current_original_counters2,
        current_original_counters3 ) =
    three_current_original_counters
  in
  ( current_original_counters1,
    increment_counter current_original_counters2,
    current_original_counters3 )

let increment_third_counter
    three_current_original_counters =
  let ( current_original_counters1,
        current_original_counters2,
        current_original_counters3 ) =
    three_current_original_counters
  in
  ( current_original_counters1,
    current_original_counters2,
    increment_counter current_original_counters3 )
```

Lst. B.3: Helper functions for manipulating three resource guards (part 1). These functions are used in the source code of those benchmarks that use three resource guard (e.g., Prim).

```
let initialize_counter current_original_counters =
  let _, original_counter =
    current_original_counters
  in
  (original_counter, original_counter)

let initialize_first_counter
    three_current_original_counters =
  let ( current_original_counters1,
        current_original_counters2,
        current_original_counters3 ) =
    three_current_original_counters
  in
  ( initialize_counter current_original_counters1,
    current_original_counters2,
    current_original_counters3 )

let initialize_second_counter
    three_current_original_counters =
  let ( current_original_counters1,
        current_original_counters2,
        current_original_counters3 ) =
    three_current_original_counters
  in
  ( current_original_counters1,
    initialize_counter current_original_counters2,
    current_original_counters3 )

let initialize_third_counter
    three_current_original_counters =
  let ( current_original_counters1,
        current_original_counters2,
        current_original_counters3 ) =
    three_current_original_counters
  in
  ( current_original_counters1,
    current_original_counters2,
    initialize_counter current_original_counters3 )
```

```
let set_counter_to_zero current_original_counters =
  let _, original_counter =
    current_original_counters
  in
  ([], original_counter)

let set_first_counter_to_zero
    three_current_original_counters =
  let ( current_original_counters1,
        current_original_counters2,
        current_original_counters3 ) =
    three_current_original_counters
  in
  ( set_counter_to_zero current_original_counters1,
    current_original_counters2,
    current_original_counters3 )

let set_second_counter_to_zero
    three_current_original_counters =
  let ( current_original_counters1,
        current_original_counters2,
        current_original_counters3 ) =
    three_current_original_counters
  in
  ( current_original_counters1,
    set_counter_to_zero current_original_counters2,
    current_original_counters3 )

let set_third_counter_to_zero
    three_current_original_counters =
  let ( current_original_counters1,
        current_original_counters2,
        current_original_counters3 ) =
    three_current_original_counters
  in
  ( current_original_counters1,
    current_original_counters2,
    set_counter_to_zero current_original_counters3 )
```

Lst. B.4: Helper functions for manipulating three resource guards (part 2).

B.2.2 MergeSort

```
let rec merge (xs : int list) (ys : int list) =
  let _ = Raml.tick 1.0 in
  match xs with
  | [] → ys
  | x :: xs_tl →(
     match ys with
     | [] → xs
     | y :: ys_tl →
        if x <= y then x :: merge xs_tl ys
        else y :: merge xs ys_tl )

let rec split xs =
  let _ = Raml.tick 1.0 in
  match xs with
  | [] → ([], [])
  | [ x ] → ([ x ], [])
  | x1 :: x2 :: tl →
     let lower, upper = split tl in
     (x1 :: lower, x2 :: upper)
```

```
let rec merge_sort xs =
  let _ = Raml.mark 0 1.0 in
  let _ = Raml.tick 1.0 in
  let result =
    match xs with
    | [] → []
    | [ x ] → [ x ]
    | _ →
       let lower, upper = split xs in
       let lower_sorted = merge_sort lower in
       let upper_sorted = merge_sort upper in
       merge lower_sorted upper_sorted
  in
  let _ = Raml.mark 0 (-1.0) in
  result

let main xs =
  let _ = Raml.activate_counter_variable 0 in
  let result = merge_sort xs in
  let _ = Raml.record_counter_variable 0 in
  result
```

Lst. B.5: Resource-decomposed code of MergeSort. The resource metric of our interest is the number of function calls. This cost is indicated by the construct `Raml.tick` throughout the source code. The construct `Raml.mark` specifies a resource component: the recursion depth of the function `merge_sort`.

```
let rec merge (xs : int list) (ys : int list)
    current_original_counters =
  let _ = Raml.tick 1.0 in
  match xs with
  | [] → (ys, current_original_counters)
  | x :: xs_tl →(
      match ys with
      | [] → (xs, current_original_counters)
      | y :: ys_tl →
          if x <= y then
            let ( recursive_result,
                  current_original_counters ) =
              merge xs_tl ys current_original_counters
            in
            ( x :: recursive_result,
              current_original_counters )
          else
            let ( recursive_result,
                  current_original_counters ) =
              merge xs ys_tl current_original_counters
            in
            ( y :: recursive_result,
              current_original_counters ) )

let rec split xs current_original_counters =
  let _ = Raml.tick 1.0 in
  match xs with
  | [] → (([], []), current_original_counters)
  | [ x ] → (([ x ], []), current_original_counters)
  | x1 :: x2 :: tl →
      let (lower, upper), current_original_counters =
        split tl current_original_counters
      in
      ( (x1 :: lower, x2 :: upper),
        current_original_counters )
```

```
let rec merge_sort xs current_original_counters =
  let _ = Raml.tick 1.0 in
  let new_counter =
    decrement_counter current_original_counters
  in
  let result, counter_final =
    match xs with
    | [] → ([], new_counter)
    | [ x ] → ([ x ], new_counter)
    | _ →
        let (lower, upper), counter_split =
          split xs new_counter
        in
        let lower_sorted, counter1 =
          merge_sort lower counter_split
        in
        let upper_sorted, counter2 =
          merge_sort upper counter1
        in
        merge lower_sorted upper_sorted counter2
  in
  (result, increment_counter counter_final)

(* Polynomial degree for AARA: 2 *)

let main xs current_original_counters =
  let initialized_counter =
    initialize_counter current_original_counters
  in
  let result, counter_after_sorting =
    merge_sort xs initialized_counter
  in
  (result, set_counter_to_zero counter_after_sorting)
```

Lst. B.6: Resource-guarded code of MergeSort. Helper functions for manipulating resource guards (i.e., `decrement_counter` and `increment_counter`) are defined in Listing B.1.

B.2.3 QuickSort

```
let rec partition (pivot : int) (xs : int list) =
  let _ = Raml.tick 1.0 in
  match xs with
  | [] → ([], [])
  | hd :: tl →
      let lower, upper = partition pivot tl in
      if hd < pivot then (hd :: lower, upper)
      else (lower, hd :: upper)

let rec append xs ys =
  let _ = Raml.tick 1.0 in
  match xs with
  | [] → ys
  | hd :: tl → hd :: append tl ys
```

```
let rec quicksort xs =
  let _ = Raml.mark 0 1.0 in
  let _ = Raml.tick 1.0 in
  let result =
    match xs with
    | [] → []
    | [ x ] → [ x ]
    | hd :: tl →
        let lower, upper = partition hd tl in
        let lower_sorted = quicksort lower in
        let upper_sorted = quicksort upper in
        append lower_sorted (hd :: upper_sorted)
  in
  let _ = Raml.mark 0 (-1.0) in
  result

let main xs =
  let _ = Raml.activate_counter_variable 0 in
  let result = quicksort xs in
  let _ = Raml.record_counter_variable 0 in
  result
```

Lst. B.7: Resource-decomposed code of QuickSort.

```
let rec partition (pivot : int) (xs : int list)
    current_original_counters =
  let _ = Raml.tick 1.0 in
  match xs with
  | [] → (([], []), current_original_counters)
  | hd :: tl →
      let (lower, upper), current_original_counters =
        partition pivot tl current_original_counters
      in
      let _ = Raml.tick 1.0 in
      if hd < pivot then
        ( (hd :: lower, upper),
          current_original_counters )
      else
        ( (lower, hd :: upper),
          current_original_counters )

let rec append xs ys current_original_counters =
  let _ = Raml.tick 1.0 in
  match xs with
  | [] → (ys, current_original_counters)
  | hd :: tl →
      let result_recursive, current_original_counters
          =
        append tl ys current_original_counters
      in
      ( hd :: result_recursive,
        current_original_counters )

let rec quicksort xs current_original_counters =
  let _ = Raml.tick 1.0 in
  let new_counter =
    decrement_counter current_original_counters
  in
  let result, counter_final =
    match xs with
    | [] → ([], new_counter)
    | [ x ] → ([ x ], new_counter)
    | hd :: tl →
        let (lower, upper), counter1 =
          partition hd tl new_counter
        in
        let lower_sorted, counter2 =
          quicksort lower counter1
        in
        let upper_sorted, counter3 =
          quicksort upper counter2
        in
        append lower_sorted
          (hd :: upper_sorted)
          counter3
  in
  (result, increment_counter counter_final)

(* Polynomial degree for AARA: 2 *)

let main xs current_original_counters =
  let initialized_counter =
    initialize_counter current_original_counters
  in
  let result, counter_after_sorting =
    quicksort xs initialized_counter
  in
  (result, set_counter_to_zero counter_after_sorting)
```

Lst. B.8: Resource-guarded code of QuickSort.

B.2.4 BubbleSort

```
let rec traverse_and_swap (xs : int list) =
  let _ = Raml.tick 1.0 in
  match xs with
  | [] | [ _ ] →(true, xs)
  | x1 :: x2 :: tl →
      if x1 <= x2 then
        let is_tl_sorted, tl_swapped =
          traverse_and_swap (x2 :: tl)
        in
        (is_tl_sorted, x1 :: tl_swapped)
      else
        let _, tl_swapped =
          traverse_and_swap (x1 :: tl)
        in
        (false, x2 :: tl_swapped)
```

```
let rec bubble_sort xs =
  let _ = Raml.mark 0 1.0 in
  let _ = Raml.tick 1.0 in
  let result =
    let is_xs_sorted, xs_swapped =
      traverse_and_swap xs
    in
    if is_xs_sorted then xs_swapped
    else bubble_sort xs_swapped
  in
  let _ = Raml.mark 0 (-1.0) in
  result

let main xs =
  let _ = Raml.activate_counter_variable 0 in
  let result = bubble_sort xs in
  let _ = Raml.record_counter_variable 0 in
  result
```

Lst. B.9: Resource-decomposed code of BubbleSort.

```
let rec traverse_and_swap (xs : int list)
    current_original_counters =
  let _ = Raml.tick 1.0 in
  match xs with
  | [] | [ _ ] →
      ((true, xs), current_original_counters)
  | x1 :: x2 :: tl →
      if x1 <= x2 then
        let ( (is_tl_sorted, tl_swapped),
              current_original_counters ) =
          traverse_and_swap (x2 :: tl)
            current_original_counters
        in
        ( (is_tl_sorted, x1 :: tl_swapped),
          current_original_counters )
      else
        let (_, tl_swapped), current_original_counters
          =
          traverse_and_swap (x1 :: tl)
            current_original_counters
        in
        ( (false, x2 :: tl_swapped),
          current_original_counters )
```

```
let rec bubble_sort xs current_original_counters =
  let _ = Raml.tick 1.0 in
  let new_counter =
    decrement_counter current_original_counters
  in
  let (is_xs_sorted, xs_swapped), counter_after_swap =
    traverse_and_swap xs new_counter
  in
  let result, counter_final =
    if is_xs_sorted then
      (xs_swapped, counter_after_swap)
    else bubble_sort xs_swapped counter_after_swap
  in
  (result, increment_counter counter_final)

(* Polynomial degree for AARA: 2 *)

let main xs current_original_counters =
  let initialized_counter =
    initialize_counter current_original_counters
  in
  let result, counter_after_sorting =
    bubble_sort xs initialized_counter
  in
  (result, set_counter_to_zero counter_after_sorting)
```

Lst. B.10: Resource-guarded code of BubbleSort.

B.2.5 HeapSort

```
exception Invalid_input

type binary_heap = int Rarray.t * Rnat.t

let rec heapify_build (heap : int Rarray.t * Rnat.t)
    (index : int) =
  let _ = Raml.mark 0 1.0 in
  let _ = Raml.tick 1.0 in
  let result =
    let left_index = (index * 2) + 1
    and right_index = (index * 2) + 2 in
    let array, length = heap in
    let smallest_index_left =
      if left_index < Rnat.to_int length then
        let element_index =
          Rarray.get array (Rnat.of_int index)
        in
        let element_left_index =
          Rarray.get array (Rnat.of_int left_index)
        in
        if element_left_index < element_index then
          left_index
        else index
      else index
    in
    let smallest_index_right =
      if right_index < Rnat.to_int length then
        let element_smallest_index_left =
          Rarray.get array
            (Rnat.of_int smallest_index_left)
        in
        let element_right_index =
          Rarray.get array (Rnat.of_int right_index)
        in
        if
          element_right_index
          < element_smallest_index_left
        then right_index
        else smallest_index_left
      else smallest_index_left
    in
    if smallest_index_right = index then ()
    else
      let element_at_index =
        Rarray.get array (Rnat.of_int index)
      in
      let element_at_smallest_index =
        Rarray.get array
          (Rnat.of_int smallest_index_right)
      in
      let _ =
        Rarray.set array
          (Rnat.of_int index)
          element_at_smallest_index
      in
      let _ =
        Rarray.set array
          (Rnat.of_int smallest_index_right)
          element_at_index
      in
      heapify_build heap smallest_index_right
  in
  let _ = Raml.mark 0 (-1.0) in
  result

let rec repeatedly_heapify heap index =
  let _ = Raml.tick 1.0 in
  Rnat.ifz index
    (fun () → ())
    (fun index_minus_one →
      let _ = Raml.activate_counter_variable 0 in
      let _ =
        heapify_build heap
          (Rnat.to_int index_minus_one)
      in
      let _ = Raml.record_counter_variable 0 in
      repeatedly_heapify heap index_minus_one)

let build_min_heap heap =
  let _ = Raml.tick 1.0 in
  let _, length = heap in
  repeatedly_heapify heap length

let rec copy_list_to_array xs array index =
  let _ = Raml.tick 1.0 in
  match xs with
  | [] → ()
  | hd :: tl →
      let _ =
        Rarray.set array (Rnat.of_int index) hd
      in
      copy_list_to_array tl array (index + 1)

let rec list_nat_length xs =
  let _ = Raml.tick 1.0 in
  match xs with
  | [] → Rnat.zero
  | _ :: tl → Rnat.succ (list_nat_length tl)

let create_heap_from_list xs =
  let _ = Raml.tick 1.0 in
  let nat_length = list_nat_length xs in
  let array = Rarray.make nat_length 0 in
  let _ = copy_list_to_array xs array 0 in
  let _ = build_min_heap (array, nat_length) in
  (array, nat_length)
```

Lst. B.11: Resource-decomposed code of HeapSort (part 1).

```
let rec heapify_extract (heap : int Rarray.t * Rnat.t)
    (index : int) =
  let _ = Raml.mark 1 1.0 in
  let _ = Raml.tick 1.0 in
  let result =
    let left_index = (index * 2) + 1
    and right_index = (index * 2) + 2 in
    let array, length = heap in
    let smallest_index_left =
      if left_index < Rnat.to_int length then
        let element_index =
          Rarray.get array (Rnat.of_int index)
        in
        let element_left_index =
          Rarray.get array (Rnat.of_int left_index)
        in
        if element_left_index < element_index then
          left_index
        else index
      else index
    in
    let smallest_index_right =
      if right_index < Rnat.to_int length then
        let element_smallest_index_left =
          Rarray.get array
            (Rnat.of_int smallest_index_left)
        in
        let element_right_index =
          Rarray.get array (Rnat.of_int right_index)
        in
        if
          element_right_index
          < element_smallest_index_left
        then right_index
        else smallest_index_left
      else smallest_index_left
    in
    if smallest_index_right = index then ()
    else
      let element_at_index =
        Rarray.get array (Rnat.of_int index)
      in
      let element_at_smallest_index =
        Rarray.get array
          (Rnat.of_int smallest_index_right)
      in
      let _ =
        Rarray.set array
          (Rnat.of_int index)
          element_at_smallest_index
      in
      let _ =
        Rarray.set array
          (Rnat.of_int smallest_index_right)
          element_at_index
      in
      heapify_extract heap smallest_index_right
  in
  let _ = Raml.mark 1 (-1.0) in
  result
```

```
let rec extract_list_from_heap heap =
  let _ = Raml.tick 1.0 in
  let array, length = heap in
  Rnat.ifz length
    (fun () → [])
    (fun length_minus_one →
      let min_element = Rarray.get array Rnat.zero in
      let last_element =
        Rarray.get array length_minus_one
      in
      let _ =
        Rarray.set array Rnat.zero last_element
      in
      let _ = Raml.activate_counter_variable 1 in
      let _ = heapify_extract heap 0 in
      let _ = Raml.record_counter_variable 1 in
      let recursive_result =
        extract_list_from_heap
          (array, length_minus_one)
      in
      min_element :: recursive_result)

(* Polynomial degree for AARA: 2 *)

let heap_sort xs =
  let _ = Raml.tick 1.0 in
  let heap = create_heap_from_list xs in
  extract_list_from_heap heap
```

Lst. B.12: Resource-decomposed code of HeapSort (part 2).

```
type binary_heap = int Rarray.t * Rnat.t

let rec heapify_build (heap : int Rarray.t * Rnat.t)
    (index : int) two_current_original_counters =
  let new_counter =
    decrement_first_counter
      two_current_original_counters
  in
  let result, counter_final =
    let _ = Raml.tick 1.0 in
    let left_index = (index * 2) + 1
    and right_index = (index * 2) + 2 in
    let array, length = heap in
    let smallest_index_left =
      if left_index < Rnat.to_int length then
        let element_index =
          Rarray.get array (Rnat.of_int index)
        in
        let element_left_index =
          Rarray.get array (Rnat.of_int left_index)
        in
        if element_left_index < element_index then
          left_index
        else index
      else index
    in
    let smallest_index_right =
      if right_index < Rnat.to_int length then
        let element_smallest_index_left =
          Rarray.get array
            (Rnat.of_int smallest_index_left)
        in
        let element_right_index =
          Rarray.get array (Rnat.of_int right_index)
        in
        if
          element_right_index
          < element_smallest_index_left
        then right_index
        else smallest_index_left
      else smallest_index_left
    in
    if smallest_index_right = index then
      ((), new_counter)
    else
      let element_at_index =
        Rarray.get array (Rnat.of_int index)
      in
      let element_at_smallest_index =
        Rarray.get array
          (Rnat.of_int smallest_index_right)
      in
      let _ =
        Rarray.set array
          (Rnat.of_int index)
          element_at_smallest_index
      in
      let _ =
        Rarray.set array
          (Rnat.of_int smallest_index_right)
          element_at_index
      in
      heapify_build heap smallest_index_right
        new_counter
  in
  (result, increment_first_counter counter_final)

let rec repeatedly_heapify heap index
    two_current_original_counters =
  let _ = Raml.tick 1.0 in
  Rnat.ifz index
    (fun () → ((), two_current_original_counters))
    (fun index_minus_one →
      let initialized_counter =
        initialize_first_counter
          two_current_original_counters
      in
      let _, counter1 =
        heapify_build heap
          (Rnat.to_int index_minus_one)
          initialized_counter
      in
      let counter2 =
        set_first_counter_to_zero counter1
      in
      repeatedly_heapify heap index_minus_one counter2)

let build_min_heap heap two_current_original_counters
    =
  let _ = Raml.tick 1.0 in
  let _, length = heap in
  repeatedly_heapify heap length
    two_current_original_counters

let rec copy_list_to_array xs array index =
  let _ = Raml.tick 1.0 in
  match xs with
  | [] → ()
  | hd :: tl →
      let _ =
        Rarray.set array (Rnat.of_int index) hd
      in
      copy_list_to_array tl array (index + 1)

let rec list_nat_length xs =
  let _ = Raml.tick 1.0 in
  match xs with
  | [] → Rnat.zero
  | _ :: tl → Rnat.succ (list_nat_length tl)

let create_heap_from_list xs
    two_current_original_counters =
  let _ = Raml.tick 1.0 in
  let nat_length = list_nat_length xs in
  let array = Rarray.make nat_length 0 in
  let _ = copy_list_to_array xs array 0 in
  let _, counter1 =
    build_min_heap
      (array, nat_length)
      two_current_original_counters
  in
  ((array, nat_length), counter1)
```

Lst. B.13: Resource-guarded code of HeapSort (part 1).

```
let rec heapify_extract (heap : int Rarray.t * Rnat.t)
    (index : int) two_current_original_counters =
  let new_counter =
    decrement_second_counter
      two_current_original_counters
  in
  let result, counter_final =
    let _ = Raml.tick 1.0 in
    let left_index = (index * 2) + 1
    and right_index = (index * 2) + 2 in
    let array, length = heap in
    let smallest_index_left =
      if left_index < Rnat.to_int length then
        let element_index =
          Rarray.get array (Rnat.of_int index)
        in
        let element_left_index =
          Rarray.get array (Rnat.of_int left_index)
        in
        if element_left_index < element_index then
          left_index
        else index
      else index
    in
    let smallest_index_right =
      if right_index < Rnat.to_int length then
        let element_smallest_index_left =
          Rarray.get array
            (Rnat.of_int smallest_index_left)
        in
        let element_right_index =
          Rarray.get array (Rnat.of_int right_index)
        in
        if
          element_right_index
          < element_smallest_index_left
        then right_index
        else smallest_index_left
      else smallest_index_left
    in
    if smallest_index_right = index then
      ((), new_counter)
    else
      let element_at_index =
        Rarray.get array (Rnat.of_int index)
      in
      let element_at_smallest_index =
        Rarray.get array
          (Rnat.of_int smallest_index_right)
      in
      let _ =
        Rarray.set array
          (Rnat.of_int index)
          element_at_smallest_index
      in
      let _ =
        Rarray.set array
          (Rnat.of_int smallest_index_right)
          element_at_index
      in
      heapify_extract heap smallest_index_right
        new_counter
  in
  (result, increment_second_counter counter_final)
```

```
let rec extract_list_from_heap heap
    two_current_original_counters =
  let _ = Raml.tick 1.0 in
  let array, length = heap in
  Rnat.ifz length
    (fun () → ([], two_current_original_counters))
    (fun length_minus_one →
      let min_element = Rarray.get array Rnat.zero in
      let last_element =
        Rarray.get array length_minus_one
      in
      let _ =
        Rarray.set array Rnat.zero last_element
      in
      let initialized_counter =
        initialize_second_counter
          two_current_original_counters
      in
      let _, counter1 =
        heapify_extract heap 0 initialized_counter
      in
      let counter2 =
        set_second_counter_to_zero counter1
      in
      let recursive_result, counter2 =
        extract_list_from_heap
          (array, length_minus_one)
          counter2
      in
      (min_element :: recursive_result, counter2))

let heap_sort xs two_current_original_counters =
  let _ = Raml.tick 1.0 in
  let heap, counter1 =
    create_heap_from_list xs
      two_current_original_counters
  in
  extract_list_from_heap heap counter1
```

Lst. B.14: Resource-guarded code of HeapSort (part 2).

B.2.6 HuffmanCode

```
exception Invalid_input

type code_tree =
  | LeafCode of int * int
  | NodeCode of int * code_tree * code_tree

let code_tree_count v =
  let _ = Raml.tick 1.0 in
  match v with
  | LeafCode (_, count) →count
  | NodeCode (count, _, _) →count

let merge_code_trees v1 v2 =
  let _ = Raml.tick 1.0 in
  let count1, count2 =
    (code_tree_count v1, code_tree_count v2)
  in
  NodeCode (count1 + count2, v1, v2)

type binary_heap = int Rarray.t * Rnat.t
```

Lst. B.15: Resource-decomposed code of HuffmanCode (part 1).

```
let rec heapify_build heap (index : int) =
  let _ = Raml.mark 0 1.0 in
  let _ = Raml.tick 1.0 in
  let result =
    let left_index = (index * 2) + 1
    and right_index = (index * 2) + 2 in
    let array, length = heap in
    let smallest_index_left =
      if left_index < Rnat.to_int length then
        let element_index =
          Rarray.get array (Rnat.of_int index)
        in
        let element_left_index =
          Rarray.get array (Rnat.of_int left_index)
        in
        if
          code_tree_count element_left_index
          < code_tree_count element_index
        then left_index
        else index
      else index
    in
    let smallest_index_right =
      if right_index < Rnat.to_int length then
        let element_smallest_index_left =
          Rarray.get array
            (Rnat.of_int smallest_index_left)
        in
        let element_right_index =
          Rarray.get array (Rnat.of_int right_index)
        in
        if
          code_tree_count element_right_index
          < code_tree_count
              element_smallest_index_left
        then right_index
        else smallest_index_left
      else smallest_index_left
    in
    if smallest_index_right = index then ()
    else
      let element_at_index =
        Rarray.get array (Rnat.of_int index)
      in
      let element_at_smallest_index =
        Rarray.get array
          (Rnat.of_int smallest_index_right)
      in
      let _ =
        Rarray.set array
          (Rnat.of_int index)
          element_at_smallest_index
      in
      let _ =
        Rarray.set array
          (Rnat.of_int smallest_index_right)
          element_at_index
      in
      heapify_build heap smallest_index_right
  in
  let _ = Raml.mark 0 (-1.0) in
  result

let rec repeatedly_heapify heap index =
  let _ = Raml.tick 1.0 in
  Rnat.ifz index
    (fun () → ())
    (fun index_minus_one →
      let _ = Raml.activate_counter_variable 0 in
      let _ =
        heapify_build heap
          (Rnat.to_int index_minus_one)
      in
      let _ = Raml.record_counter_variable 0 in
      repeatedly_heapify heap index_minus_one)

let build_min_heap heap =
  let _ = Raml.tick 1.0 in
  let _, length = heap in
  repeatedly_heapify heap length

let rec copy_list_to_array xs array index =
  let _ = Raml.tick 1.0 in
  match xs with
  | [] → ()
  | (character, count) :: tl →
      let _ =
        Rarray.set array
          (Rnat.of_int index)
          (LeafCode (character, count))
      in
      copy_list_to_array tl array (index + 1)

let rec list_nat_length xs =
  let _ = Raml.tick 1.0 in
  match xs with
  | [] → Rnat.zero
  | _ :: tl → Rnat.succ (list_nat_length tl)

let create_heap_from_list xs =
  let _ = Raml.tick 1.0 in
  let nat_length = list_nat_length xs in
  let array =
    Rarray.make nat_length (LeafCode (-1, -1))
  in
  let _ = copy_list_to_array xs array 0 in
  let _ = build_min_heap (array, nat_length) in
  (array, nat_length)

let get_min heap =
  let _ = Raml.tick 1.0 in
  let array, _ = heap in
  Rarray.get array Rnat.zero
```

Lst. B.16: Resource-decomposed code of HuffmanCode (part 2).

```
let rec heapify_extract heap (index : int) =
  let _ = Raml.mark 1 1.0 in
  let _ = Raml.tick 1.0 in
  let result =
    let left_index = (index * 2) + 1
    and right_index = (index * 2) + 2 in
    let array, length = heap in
    let smallest_index_left =
      if left_index < Rnat.to_int length then
        let element_index =
          Rarray.get array (Rnat.of_int index)
        in
        let element_left_index =
          Rarray.get array (Rnat.of_int left_index)
        in
        if
          code_tree_count element_left_index
          < code_tree_count element_index
        then left_index
        else index
      else index
    in
    let smallest_index_right =
      if right_index < Rnat.to_int length then
        let element_smallest_index_left =
          Rarray.get array
            (Rnat.of_int smallest_index_left)
        in
        let element_right_index =
          Rarray.get array (Rnat.of_int right_index)
        in
        if
          code_tree_count element_right_index
          < code_tree_count
              element_smallest_index_left
        then right_index
        else smallest_index_left
      else smallest_index_left
    in
    if smallest_index_right = index then ()
    else
      let element_at_index =
        Rarray.get array (Rnat.of_int index)
      in
      let element_at_smallest_index =
        Rarray.get array
          (Rnat.of_int smallest_index_right)
      in
      let _ =
        Rarray.set array
          (Rnat.of_int index)
          element_at_smallest_index
      in
      let _ =
        Rarray.set array
          (Rnat.of_int smallest_index_right)
          element_at_index
      in
      heapify_extract heap smallest_index_right
  in
  let _ = Raml.mark 1 (-1.0) in
  result

let heapify_index_zero heap =
  let _ = Raml.activate_counter_variable 1 in
  let result = heapify_extract heap 0 in
  let _ = Raml.record_counter_variable 1 in
  result

let rec recursively_construct_huffman_code heap =
  let _ = Raml.tick 1.0 in
  let array, length = heap in
  Rnat.ifz length
    (fun () → raise Invalid_input)
    (fun length_minus_one →
      let v1 = get_min heap in
      Rnat.ifz length_minus_one
        (fun () → v1)
        (fun _ →
          let last_element =
            Rarray.get array length_minus_one
          in
          let _ =
            Rarray.set array Rnat.zero last_element
          in
          let heap1 = (array, length_minus_one) in
          let _ = heapify_index_zero heap1 in
          let v2 = get_min heap1 in
          let merged_code_tree =
            merge_code_trees v1 v2
          in
          let _ =
            Rarray.set array Rnat.zero
              merged_code_tree
          in
          let _ = heapify_index_zero heap1 in
          recursively_construct_huffman_code heap1))

let huffman_code xs =
  let _ = Raml.tick 1.0 in
  let heap = create_heap_from_list xs in
  recursively_construct_huffman_code heap
```

Lst. B.17: Resource-decomposed code of HuffmanCode (part 3).

```
type code_tree =
  | LeafCode of int * int
  | NodeCode of int * code_tree * code_tree

let code_tree_count v =
  let _ = Raml.tick 1.0 in
  match v with
  | LeafCode (_, count) →count
  | NodeCode (count, _, _) →count

let merge_code_trees v1 v2 =
  let _ = Raml.tick 1.0 in
  let count1, count2 =
    (code_tree_count v1, code_tree_count v2)
  in
  NodeCode (count1 + count2, v1, v2)

type binary_heap = int Rarray.t * Rnat.t
```

Lst. B.18: Resource-guarded code of HuffmanCode (part 1).

```
let rec heapify_build heap (index : int)
    two_current_original_counters =
  let new_counter =
    decrement_first_counter
      two_current_original_counters
  in
  let result, counter_final =
    let _ = Raml.tick 1.0 in
    let left_index = (index * 2) + 1
    and right_index = (index * 2) + 2 in
    let array, length = heap in
    let smallest_index_left =
      if left_index < Rnat.to_int length then
        let element_index =
          Rarray.get array (Rnat.of_int index)
        in
        let element_left_index =
          Rarray.get array (Rnat.of_int left_index)
        in
        if
          code_tree_count element_left_index
          < code_tree_count element_index
        then left_index
        else index
      else index
    in
    let smallest_index_right =
      if right_index < Rnat.to_int length then
        let element_smallest_index_left =
          Rarray.get array
            (Rnat.of_int smallest_index_left)
        in
        let element_right_index =
          Rarray.get array (Rnat.of_int right_index)
        in
        if
          code_tree_count element_right_index
          < code_tree_count
              element_smallest_index_left
        then right_index
        else smallest_index_left
      else smallest_index_left
    in
    if smallest_index_right = index then
      ((), new_counter)
    else
      let element_at_index =
        Rarray.get array (Rnat.of_int index)
      in
      let element_at_smallest_index =
        Rarray.get array
          (Rnat.of_int smallest_index_right)
      in
      let _ =
        Rarray.set array
          (Rnat.of_int index)
          element_at_smallest_index
      in
      let _ =
        Rarray.set array
          (Rnat.of_int smallest_index_right)
          element_at_index
      in
      heapify_build heap smallest_index_right
        new_counter
  in
  (result, increment_first_counter counter_final)

let rec repeatedly_heapify heap index
    two_current_original_counters =
  let _ = Raml.tick 1.0 in
  Rnat.ifz index
    (fun () → ((), two_current_original_counters))
    (fun index_minus_one →
      let initialized_counter =
        initialize_first_counter
          two_current_original_counters
      in
      let _, counter1 =
        heapify_build heap
          (Rnat.to_int index_minus_one)
          initialized_counter
      in
      let counter2 =
        set_first_counter_to_zero counter1
      in
      repeatedly_heapify heap index_minus_one counter2)

let build_min_heap heap two_current_original_counters
    =
  let _ = Raml.tick 1.0 in
  let _, length = heap in
  repeatedly_heapify heap length
    two_current_original_counters

let rec copy_list_to_array xs array index =
  let _ = Raml.tick 1.0 in
  match xs with
  | [] → ()
  | (character, count) :: tl →
      let _ =
        Rarray.set array
          (Rnat.of_int index)
          (LeafCode (character, count))
      in
      copy_list_to_array tl array (index + 1)

let rec list_nat_length xs =
  let _ = Raml.tick 1.0 in
  match xs with
  | [] → Rnat.zero
  | _ :: tl → Rnat.succ (list_nat_length tl)

let create_heap_from_list xs
    two_current_original_counters =
  let _ = Raml.tick 1.0 in
  let nat_length = list_nat_length xs in
  let array =
    Rarray.make nat_length (LeafCode (-1, -1))
  in
  let _ = copy_list_to_array xs array 0 in
  let _, counter1 =
    build_min_heap
      (array, nat_length)
      two_current_original_counters
  in
  ((array, nat_length), counter1)

let get_min heap =
  let _ = Raml.tick 1.0 in
  let array, _ = heap in
  Rarray.get array Rnat.zero
```

Lst. B.19: Resource-guarded code of HuffmanCode (part 2).

```
let rec heapify_extract heap (index : int)
    two_current_original_counters =
  let new_counter =
    decrement_second_counter
      two_current_original_counters
  in
  let result, counter_final =
    let _ = Raml.tick 1.0 in
    let left_index = (index * 2) + 1
    and right_index = (index * 2) + 2 in
    let array, length = heap in
    let smallest_index_left =
      if left_index < Rnat.to_int length then
        let element_index =
          Rarray.get array (Rnat.of_int index)
        in
        let element_left_index =
          Rarray.get array (Rnat.of_int left_index)
        in
        if
          code_tree_count element_left_index
          < code_tree_count element_index
        then left_index
        else index
      else index
    in
    let smallest_index_right =
      if right_index < Rnat.to_int length then
        let element_smallest_index_left =
          Rarray.get array
            (Rnat.of_int smallest_index_left)
        in
        let element_right_index =
          Rarray.get array (Rnat.of_int right_index)
        in
        if
          code_tree_count element_right_index
          < code_tree_count
              element_smallest_index_left
        then right_index
        else smallest_index_left
      else smallest_index_left
    in
    if smallest_index_right = index then
      ((), new_counter)
    else
      let element_at_index =
        Rarray.get array (Rnat.of_int index)
      in
      let element_at_smallest_index =
        Rarray.get array
          (Rnat.of_int smallest_index_right)
      in
      let _ =
        Rarray.set array
          (Rnat.of_int index)
          element_at_smallest_index
      in
      let _ =
        Rarray.set array
          (Rnat.of_int smallest_index_right)
          element_at_index
      in
      heapify_extract heap smallest_index_right
        new_counter
  in
  (result, increment_second_counter counter_final)

let heapify_index_zero heap
    two_current_original_counters =
  let initialized_counter =
    initialize_second_counter
      two_current_original_counters
  in
  let result, counter1 =
    heapify_extract heap 0 initialized_counter
  in
  (result, set_second_counter_to_zero counter1)

let rec recursively_construct_huffman_code heap
    two_current_original_counters =
  let _ = Raml.tick 1.0 in
  let array, length = heap in
  Rnat.ifz length
    (fun () → raise Invalid_input)
    (fun length_minus_one →
      let v1 = get_min heap in
      Rnat.ifz length_minus_one
        (fun () →
          (v1, two_current_original_counters))
        (fun _ →
          let last_element =
            Rarray.get array length_minus_one
          in
          let _ =
            Rarray.set array Rnat.zero last_element
          in
          let heap1 = (array, length_minus_one) in
          let _, counter1 =
            heapify_index_zero heap1
              two_current_original_counters
          in
          let v2 = get_min heap1 in
          let merged_code_tree =
            merge_code_trees v1 v2
          in
          let _ =
            Rarray.set array Rnat.zero
              merged_code_tree
          in
          let _, counter2 =
            heapify_index_zero heap1 counter1
          in
          recursively_construct_huffman_code heap1
            counter2))

(* Polynomial degree for AARA: 2 *)

let huffman_code xs two_current_original_counters =
  let _ = Raml.tick 1.0 in
  let heap, counter1 =
    create_heap_from_list xs
      two_current_original_counters
  in
  recursively_construct_huffman_code heap counter1
```

Lst. B.20: Resource-guarded code of HuffmanCode (part 3).

B.2.7 BalancedBST and UnbalancedBST

```
exception Invalid_input

let rec merge (xs : int list) (ys : int list) =
  let _ = Raml.tick 1.0 in
  match xs with
  | [] → ys
  | x :: xs_tl →(
      match ys with
      | [] → xs
      | y :: ys_tl →
          if x <= y then x :: merge xs_tl ys
          else y :: merge xs ys_tl )

let rec split xs =
  let _ = Raml.tick 1.0 in
  match xs with
  | [] → ([], [])
  | [ x ] → ([ x ], [])
  | x1 :: x2 :: tl →
      let lower, upper = split tl in
      (x1 :: lower, x2 :: upper)

let rec merge_sort xs =
  let _ = Raml.mark 0 1.0 in
  let _ = Raml.tick 1.0 in
  let result =
    match xs with
    | [] → []
    | [ x ] → [ x ]
    | _ →
        let lower, upper = split xs in
        let lower_sorted = merge_sort lower in
        let upper_sorted = merge_sort upper in
        merge lower_sorted upper_sorted
  in
  let _ = Raml.mark 0 (-1.0) in
  result

type binary_tree =
  | Leaf
  | Node of int * binary_tree * binary_tree

type nat = Zero | Succ of nat

let rec divide_nat_by_two n =
  (* let _ = Raml.tick 1.0 in *)
  match n with
  | Zero → (Zero, Zero)
  | Succ n → (
      match n with
      | Zero → (Succ Zero, Zero)
      | Succ n →
          let n1, n2 = divide_nat_by_two n in
          (Succ n1, Succ n2) )

let rec balanced_binary_tree_from_sorted_list_helper n
    xs =
  let _ = Raml.tick 1.0 in
  match n with
  | Zero → (Leaf, xs)
  | Succ Zero →(
      match xs with
      | [] → raise Invalid_input
      | hd :: tl →(Node (hd, Leaf, Leaf), tl) )
  | Succ n_minus_one →(
      let n1, n2 = divide_nat_by_two n_minus_one in
      let t1, xs1 =
        balanced_binary_tree_from_sorted_list_helper
          n1 xs
      in
      match xs1 with
      | [] → raise Invalid_input
      | y :: ys →
          let t2, xs2 =
            balanced_binary_tree_from_sorted_list_helper
              n2 ys
          in
          (Node (y, t1, t2), xs2) )

let rec list_length_nat xs =
  let _ = Raml.tick 1.0 in
  match xs with
  | [] → Zero
  | _ :: tl → Succ (list_length_nat tl)

let balanced_binary_tree_from_sorted_list xs =
  let _ = Raml.tick 1.0 in
  let n = list_length_nat xs in
  let t, _ =
    balanced_binary_tree_from_sorted_list_helper n xs
  in
  t

let rec binary_search_tree_insert v tree =
  let _ = Raml.mark 0 1.0 in
  let _ = Raml.tick 1.0 in
  let result =
    match tree with
    | Leaf → Node (v, Leaf, Leaf)
    | Node (x, left, right) →
        if x = v then tree
        else if v < x then
          let left_inserted =
            binary_search_tree_insert v left
          in
          Node (x, left_inserted, right)
        else
          let right_inserted =
            binary_search_tree_insert v right
          in
          Node (x, left, right_inserted)
  in
  let _ = Raml.mark 0 (-1.0) in
  result
```

Lst. B.21: Resource-decomposed code of BalancedBST and UnbalancedBST (part 1). The function `divide_nat_by_two`, which divides a natural number (encoded using an inductive data type), does not incur costs. This is because we assume that integer division takes constant time. If we inserted `Raml.tick 1.0` in the function `divide_nat_by_two` to count its number of function calls, it would result in a quadratic cost bound for the function `balanced_binary_tree_from_sorted_list_helper`, which creates a balanced binary search tree from a sorted list of integers. Instead, we would like the construction of a balanced binary search tree to have a linear cost bound.

```
let rec binary_search_tree_lookup v tree =
  let _ = Raml.mark 1 1.0 in
  let _ = Raml.tick 1.0 in
  let result =
    match tree with
    | Leaf → false
    | Node (x, left, right) →
        if x = v then true
        else if v < x then
          binary_search_tree_lookup v left
        else binary_search_tree_lookup v right
  in
  let _ = Raml.mark 1 (-1.0) in
  result

let rec binary_search_tree_repeated_insert xs acc =
  let _ = Raml.tick 1.0 in
  match xs with
  | [] → acc
  | hd :: tl →
      let _ = Raml.activate_counter_variable 0 in
      let acc_updated =
        binary_search_tree_insert hd acc
      in
      let _ = Raml.record_counter_variable 0 in
      binary_search_tree_repeated_insert tl
        acc_updated
```

```
let rec binary_search_tree_repeated_lookup xs tree =
  let _ = Raml.tick 1.0 in
  match xs with
  | [] → []
  | hd :: tl →
      let _ = Raml.activate_counter_variable 1 in
      let is_found =
        binary_search_tree_lookup hd tree
      in
      let _ = Raml.record_counter_variable 1 in
      let recursive_result =
        binary_search_tree_repeated_lookup tl tree
      in
      is_found :: recursive_result

let unbalanced_binary_search_tree_main xs1 xs2 =
  let _ = Raml.tick 1.0 in
  let tree =
    binary_search_tree_repeated_insert xs1 Leaf
  in
  binary_search_tree_repeated_lookup xs2 tree

let balanced_binary_search_tree_main xs1 xs2 =
  let _ = Raml.tick 1.0 in
  let _ = Raml.activate_counter_variable 0 in
  let xs_sorted = merge_sort xs1 in
  let _ = Raml.record_counter_variable 0 in
  let tree =
    balanced_binary_tree_from_sorted_list xs_sorted
  in
  binary_search_tree_repeated_lookup xs2 tree
```

Lst. B.22: Resource-decomposed code of BalancedBST and UnbalancedBST (part 2).

```
let rec merge (xs : int list) (ys : int list) =
  let _ = Raml.tick 1.0 in
  match xs with
  | [] → ys
  | x :: xs_tl →(
      match ys with
      | [] → xs
      | y :: ys_tl →
          if x <= y then x :: merge xs_tl ys
          else y :: merge xs ys_tl )

let rec split xs =
  let _ = Raml.tick 1.0 in
  match xs with
  | [] → ([], [])
  | [ x ] → ([ x ], [])
  | x1 :: x2 :: tl →
      let lower, upper = split tl in
      (x1 :: lower, x2 :: upper)

let rec merge_sort xs current_original_counters =
  let new_counter =
    decrement_first_counter current_original_counters
  in
  let result, counter_final =
    let _ = Raml.tick 1.0 in
    match xs with
    | [] → ([], new_counter)
    | [ x ] → ([ x ], new_counter)
    | _ →
        let lower, upper = split xs in
        let lower_sorted, counter1 =
          merge_sort lower new_counter
        in
        let upper_sorted, counter2 =
          merge_sort upper counter1
        in
        (merge lower_sorted upper_sorted, counter2)
  in
  (result, increment_first_counter counter_final)

(* Binary search tree *)

type binary_tree =
  | Leaf
  | Node of int * binary_tree * binary_tree

type nat = Zero | Succ of nat

let rec divide_nat_by_two n =
  (* let _ = Raml.tick 1.0 in *)
  match n with
  | Zero → (Zero, Zero)
  | Succ n → (
      match n with
      | Zero → (Succ Zero, Zero)
      | Succ n →
          let n1, n2 = divide_nat_by_two n in
          (Succ n1, Succ n2) )

let rec balanced_binary_tree_from_sorted_list_helper n
    xs =
  let _ = Raml.tick 1.0 in
  match n with
  | Zero → (Leaf, xs)
  | Succ Zero →(
      match xs with
      | [] → raise Invalid_input
      | hd :: tl →(Node (hd, Leaf, Leaf), tl) )
  | Succ n_minus_one →(
      let n1, n2 = divide_nat_by_two n_minus_one in
      let t1, xs1 =
        balanced_binary_tree_from_sorted_list_helper
          n1 xs
      in
      match xs1 with
      | [] → raise Invalid_input
      | y :: ys →
          let t2, xs2 =
            balanced_binary_tree_from_sorted_list_helper
              n2 ys
          in
          (Node (y, t1, t2), xs2) )

let rec list_length_nat xs =
  let _ = Raml.tick 1.0 in
  match xs with
  | [] → Zero
  | _ :: tl → Succ (list_length_nat tl)

let balanced_binary_tree_from_sorted_list xs =
  let _ = Raml.tick 1.0 in
  let n = list_length_nat xs in
  let t, _ =
    balanced_binary_tree_from_sorted_list_helper n xs
  in
  t

let rec binary_search_tree_insert v tree
    current_original_counters =
  let _ = Raml.tick 1.0 in
  let new_counter =
    decrement_first_counter current_original_counters
  in
  let result, counter_final =
    match tree with
    | Leaf → (Node (v, Leaf, Leaf), new_counter)
    | Node (x, left, right) →
        if x = v then (tree, new_counter)
        else if v < x then
          let left_inserted, counter1 =
            binary_search_tree_insert v left
              new_counter
          in
          (Node (x, left_inserted, right), counter1)
        else
          let right_inserted, counter1 =
            binary_search_tree_insert v right
              new_counter
          in
          (Node (x, left, right_inserted), counter1)
  in
  (result, increment_first_counter counter_final)
```

Lst. B.23: Resource-guarded code of BalancedBST and UnbalancedBST (part 1).

```
let rec binary_search_tree_lookup v tree
    current_original_counters =
  let _ = Raml.tick 1.0 in
  let new_counter =
    decrement_second_counter current_original_counters
  in
  let result, counter_final =
    match tree with
    | Leaf → (false, new_counter)
    | Node (x, left, right) →
        if x = v then (true, new_counter)
        else if v < x then
          binary_search_tree_lookup v left new_counter
        else
          binary_search_tree_lookup v right
            new_counter
  in
  (result, increment_second_counter counter_final)

let rec binary_search_tree_repeated_insert xs acc
    current_original_counters =
  let _ = Raml.tick 1.0 in
  match xs with
  | [] → (acc, current_original_counters)
  | hd :: tl →
      let initialized_counter =
        initialize_first_counter
          current_original_counters
      in
      let acc_updated, counter1 =
        binary_search_tree_insert hd acc
          initialized_counter
      in
      let counter2 =
        set_first_counter_to_zero counter1
      in
      binary_search_tree_repeated_insert tl
        acc_updated counter2
```

(a) Part 3.

```
let rec binary_search_tree_repeated_lookup xs tree
    current_original_counters =
  let _ = Raml.tick 1.0 in
  match xs with
  | [] → ([], current_original_counters)
  | hd :: tl →
      let initialized_counter =
        initialize_second_counter
          current_original_counters
      in
      let is_found, counter1 =
        binary_search_tree_lookup hd tree
          initialized_counter
      in
      let counter2 =
        set_second_counter_to_zero counter1
      in
      let recursive_result, counter3 =
        binary_search_tree_repeated_lookup tl tree
          counter2
      in
      (is_found :: recursive_result, counter3)

(* Polynomial degree for AARA: 2 *)

let unbalanced_binary_search_tree_main xs1 xs2
    current_original_counters =
  let _ = Raml.tick 1.0 in
  let tree, counter1 =
    binary_search_tree_repeated_insert xs1 Leaf
      current_original_counters
  in
  binary_search_tree_repeated_lookup xs2 tree counter1

(* Polynomial degree for AARA: 2 *)

let balanced_binary_search_tree_main xs1 xs2
    current_original_counters =
  let _ = Raml.tick 1.0 in
  let counter1 =
    initialize_first_counter current_original_counters
  in
  let xs_sorted, counter2 = merge_sort xs1 counter1 in
  let counter3 = set_first_counter_to_zero counter2 in
  let tree =
    balanced_binary_tree_from_sorted_list xs_sorted
  in
  binary_search_tree_repeated_lookup xs2 tree counter3
```

(b) Part 4.

Lst. B.24: Resource-guarded code of BalancedBST and UnbalancedBST (part 2).

B.2.8 RedBlackTree

```
exception Invalid_input

type color = Red | Black

type red_black_tree =
  | Leaf
  | Node of
     color * int * red_black_tree * red_black_tree

let rec red_black_tree_lookup v tree =
  let _ = Raml.mark 1 1.0 in
  let _ = Raml.tick 1.0 in
  let result =
    match tree with
    | Leaf → false
    | Node (_, x, left, right) →
       if x = v then true
       else if v < x then
         red_black_tree_lookup v left
       else red_black_tree_lookup v right
  in
  let _ = Raml.mark 1 (-1.0) in
  result

let balance color v t1 t2 =
  let _ = Raml.tick 1.0 in
  match (color, v, t1, t2) with
  | Black, z, Node (Red, y, Node (Red, x, a, b), c), d
  | Black, z, Node (Red, x, a, Node (Red, y, b, c)), d
  | Black, x, a, Node (Red, z, Node (Red, y, b, c), d)
  | Black, x, a, Node (Red, y, b, Node (Red, z, c, d))
    →
     Node
       ( Red,
         y,
         Node (Black, x, a, b),
         Node (Black, z, c, d) )
  | a, b, c, d →Node (a, b, c, d)

let rec red_black_tree_insert_helper x tree =
  let _ = Raml.mark 0 1.0 in
  let _ = Raml.tick 1.0 in
  let result =
    match tree with
    | Leaf → Node (Red, x, Leaf, Leaf)
    | Node (color, y, a, b) →
       if x < y then
         balance color y
           (red_black_tree_insert_helper x a)
           b
       else if x > y then
         balance color y a
           (red_black_tree_insert_helper x b)
       else tree
  in
  let _ = Raml.mark 0 (-1.0) in
  result

let red_black_tree_insert x tree =
  let _ = Raml.tick 1.0 in
  let _ = Raml.activate_counter_variable 0 in
  let insert_result =
    red_black_tree_insert_helper x tree
  in
  let _ = Raml.record_counter_variable 0 in
  match insert_result with
  | Node (_, y, a, b) →Node (Black, y, a, b)
  | Leaf → raise Invalid_input

let rec red_black_tree_repeated_insert xs acc =
  let _ = Raml.tick 1.0 in
  match xs with
  | [] → acc
  | hd :: tl →
      let acc_updated =
        red_black_tree_insert hd acc
      in
      red_black_tree_repeated_insert tl acc_updated

let rec red_black_tree_repeated_lookup xs tree =
  let _ = Raml.tick 1.0 in
  match xs with
  | [] → []
  | hd :: tl →
      let _ = Raml.activate_counter_variable 1 in
      let is_found = red_black_tree_lookup hd tree in
      let _ = Raml.record_counter_variable 1 in
      let recursive_result =
        red_black_tree_repeated_lookup tl tree
      in
      is_found :: recursive_result

let red_black_tree_main xs1 xs2 =
  let _ = Raml.tick 1.0 in
  let tree =
    red_black_tree_repeated_insert xs1 Leaf
  in
  red_black_tree_repeated_lookup xs2 tree
```

Lst. B.25: Resource-decomposed code of RedBlackTree.

```
type color = Red | Black

type red_black_tree =
  | Leaf
  | Node of
      color * int * red_black_tree * red_black_tree

let rec red_black_tree_lookup v tree
    current_original_counters =
  let _ = Raml.tick 1.0 in
  let new_counter =
    decrement_second_counter current_original_counters
  in
  let result, counter_final =
    match tree with
    | Leaf → (false, new_counter)
    | Node (_, x, left, right) →
        if x = v then (true, new_counter)
        else if v < x then
          red_black_tree_lookup v left new_counter
        else red_black_tree_lookup v right new_counter
  in
  (result, increment_second_counter counter_final)

let balance color v t1 t2 current_original_counters =
  let _ = Raml.tick 1.0 in
  match (color, v, t1, t2) with
  | Black, z, Node (Red, y, Node (Red, x, a, b), c), d
  | Black, z, Node (Red, x, a, Node (Red, y, b, c)), d
  | Black, x, a, Node (Red, z, Node (Red, y, b, c), d)
  | Black, x, a, Node (Red, y, b, Node (Red, z, c, d))
    →
      ( Node
          ( Red,
            y,
            Node (Black, x, a, b),
            Node (Black, z, c, d) ),
        current_original_counters )
  | a, b, c, d →
      (Node (a, b, c, d), current_original_counters)

let rec red_black_tree_insert_helper x tree
    current_original_counters =
  let _ = Raml.tick 1.0 in
  let new_counter =
    decrement_first_counter current_original_counters
  in
  let result, counter_final =
    match tree with
    | Leaf → (Node (Red, x, Leaf, Leaf), new_counter)
    | Node (color, y, a, b) →
        let _ = Raml.tick 1.0 in
        if x < y then
          let result_recursive, counter1 =
            red_black_tree_insert_helper x a
              new_counter
          in
          balance color y result_recursive b counter1
        else if x > y then
          let result_recursive, counter1 =
            red_black_tree_insert_helper x b
              new_counter
          in
          balance color y a result_recursive counter1
        else (tree, new_counter)
  in
  (result, increment_first_counter counter_final)

let red_black_tree_insert x tree
    current_original_counters =
  let _ = Raml.tick 1.0 in
  let initialized_counter =
    initialize_first_counter current_original_counters
  in
  let insert_result, counter1 =
    red_black_tree_insert_helper x tree
      initialized_counter
  in
  let counter2 = set_first_counter_to_zero counter1 in
  match insert_result with
  | Node (_, y, a, b) →
      (Node (Black, y, a, b), counter2)
  | Leaf → raise Invalid_input

let rec red_black_tree_repeated_insert xs acc
    current_original_counters =
  let _ = Raml.tick 1.0 in
  match xs with
  | [] → (acc, current_original_counters)
  | hd :: tl →
      let acc_updated, counter1 =
        red_black_tree_insert hd acc
          current_original_counters
      in
      red_black_tree_repeated_insert tl acc_updated
        counter1

let rec red_black_tree_repeated_lookup xs tree
    current_original_counters =
  let _ = Raml.tick 1.0 in
  match xs with
  | [] → ([], current_original_counters)
  | hd :: tl →
      let initialized_counter =
        initialize_second_counter
          current_original_counters
      in
      let is_found, counter1 =
        red_black_tree_lookup hd tree
          initialized_counter
      in
      let counter2 =
        set_second_counter_to_zero counter1
      in
      let recursive_result, counter3 =
        red_black_tree_repeated_lookup tl tree
          counter2
      in
      (is_found :: recursive_result, counter3)

(* Polynomial degree for AARA: 2 *)

let red_black_tree_main xs1 xs2
    current_original_counters =
  let _ = Raml.tick 1.0 in
  let tree, counter1 =
    red_black_tree_repeated_insert xs1 Leaf
      current_original_counters
  in
  red_black_tree_repeated_lookup xs2 tree counter1
```

Lst. B.26: Resource-guarded code of RedBlackTree.

B.2.9 AVLTree

```
exception Invalid_input

type avl_tree =
  | Leaf
  | Node of int * int * avl_tree * avl_tree

let rec avl_tree_lookup v tree =
  let _ = Raml.mark 1 1.0 in
  let _ = Raml.tick 1.0 in
  let result =
    match tree with
    | Leaf → false
    | Node (x, _, l, r) →
        if x = v then true
        else if v < x then avl_tree_lookup v l
        else avl_tree_lookup v r
  in
  let _ = Raml.mark 1 (-1.0) in
  result

let depth tree =
  let _ = Raml.tick 1.0 in
  match tree with Node (_, d, _, _) →d | Leaf →0

let value tree =
  let _ = Raml.tick 1.0 in
  match tree with
  | Node (x, _, _, _) →x
  | Leaf → raise Invalid_input

let max (x : int) (y : int) =
  let _ = Raml.tick 1.0 in
  if x >= y then x else y

let balanceLL tree =
  let _ = Raml.tick 1.0 in
  match tree with
  | Node (x, _, Node (x1, _, ll, rl), r) →
      let rmax = max (depth rl) (depth r) + 1 in
      let cmax = max rmax (depth ll) + 1 in
      Node (x1, cmax, ll, Node (x, rmax, rl, r))
  | _ → raise Invalid_input

let balanceLR tree =
  let _ = Raml.tick 1.0 in
  match tree with
  | Node
      (x, _, Node (y, _, ll, Node (z, _, lrl, lrr)), r)
    →
      let lmax = max (depth ll) (depth lrl) + 1 in
      let rmax = max (depth lrr) (depth r) + 1 in
      let cmax = max lmax rmax + 1 in
      Node
        ( z,
          cmax,
          Node (y, lmax, ll, lrl),
          Node (x, rmax, lrr, r) )
  | _ → raise Invalid_input

let balanceRR tree =
  let _ = Raml.tick 1.0 in
  match tree with
  | Node (x, _, l, Node (xr, _, lr, rr)) →
      let lmax = max (depth l) (depth lr) + 1 in
      let cmax = max lmax (depth rr) + 1 in
      Node (xr, cmax, Node (x, lmax, l, lr), rr)
  | _ → raise Invalid_input

let balanceRL tree =
  let _ = Raml.tick 1.0 in
  match tree with
  | Node
      (x, _, l, Node (y, _, Node (z, _, rll, rlr), rr))
    →
      let lmax = max (depth l) (depth rll) + 1 in
      let rmax = max (depth rlr) (depth rr) + 1 in
      let cmax = max lmax rmax + 1 in
      Node
        ( z,
          cmax,
          Node (x, lmax, l, rll),
          Node (y, rmax, rlr, rr) )
  | _ → raise Invalid_input

let rec avl_tree_insert v tree =
  let _ = Raml.mark 0 1.0 in
  let _ = Raml.tick 1.0 in
  let result =
    match tree with
    | Node (x, _, l, r) →
        if x = v then tree
        else if v < x then
          let insL = avl_tree_insert v l in
          let dl = depth insL in
          let dr = depth r in
          let bal = dl - dr in
          if bal < 2 || bal > 2 then
            Node (x, max dr dl + 1, insL, r)
          else if v < value l then
            balanceLL (Node (x, dl + 1, insL, r))
          else if v > value l then
            balanceLR (Node (x, dl + 1, insL, r))
          else tree
        else
          let insR = avl_tree_insert v r in
          let dr = depth insR in
          let dl = depth l in
          let bal = dl - dr in
          if bal < -2 || bal > -2 then
            Node (x, max dr dl + 1, l, insR)
          else if v > value r then
            balanceRR (Node (x, dr + 1, l, insR))
          else if v < value r then
            balanceRL (Node (x, dr + 1, l, insR))
          else tree
    | Leaf → Node (v, 1, Leaf, Leaf)
  in
  let _ = Raml.mark 0 (-1.0) in
  result
```

Lst. B.27: Resource-decomposed code of AVLTree (part 1).

```
let rec min tree =
  let _ = Raml.tick 1.0 in
  match tree with
  | Node (x, _, Leaf, _) →x
  | Node (_, _, l, _) →min l
  | Leaf → raise Invalid_input

let left_subtree tree =
  let _ = Raml.tick 1.0 in
  match tree with
  | Node (_, _, l, _) →l
  | Leaf → raise Invalid_input

let right_subtree tree =
  let _ = Raml.tick 1.0 in
  match tree with
  | Node (_, _, _, r) →r
  | Leaf → raise Invalid_input
```

```
let rec avl_tree_repeated_insert xs acc =
  let _ = Raml.tick 1.0 in
  match xs with
  | [] → acc
  | hd :: tl →
      let _ = Raml.activate_counter_variable 0 in
      let acc_updated = avl_tree_insert hd acc in
      let _ = Raml.record_counter_variable 0 in
      avl_tree_repeated_insert tl acc_updated

let rec avl_tree_repeated_lookup xs tree =
  let _ = Raml.tick 1.0 in
  match xs with
  | [] → []
  | hd :: tl →
      let _ = Raml.activate_counter_variable 1 in
      let is_found = avl_tree_lookup hd tree in
      let _ = Raml.record_counter_variable 1 in
      let recursive_result =
        avl_tree_repeated_lookup tl tree
      in
      is_found :: recursive_result

let avl_tree_main xs1 xs2 =
  let _ = Raml.tick 1.0 in
  let tree = avl_tree_repeated_insert xs1 Leaf in
  avl_tree_repeated_lookup xs2 tree
```

Lst. B.28: Resource-decomposed code of AVLTree (part 2).

```
type avl_tree =
  | Leaf
  | Node of int * int * avl_tree * avl_tree

let rec avl_tree_lookup v tree
    current_original_counters =
  let _ = Raml.tick 1.0 in
  let new_counter =
    decrement_second_counter current_original_counters
  in
  let result, counter_final =
    match tree with
    | Leaf → (false, new_counter)
    | Node (x, _, l, r) →
        if x = v then (true, new_counter)
        else if v < x then
          avl_tree_lookup v l new_counter
        else avl_tree_lookup v r new_counter
  in
  (result, increment_second_counter counter_final)

let depth tree =
  let _ = Raml.tick 1.0 in
  match tree with Node (_, d, _, _) →d | Leaf →0

let value tree =
  let _ = Raml.tick 1.0 in
  match tree with
  | Node (x, _, _, _) →x
  | Leaf → raise Invalid_input

let max (x : int) (y : int) =
  let _ = Raml.tick 1.0 in
  if x >= y then x else y

let balanceLL tree =
  let _ = Raml.tick 1.0 in
  match tree with
  | Node (x, _, Node (xl, _, ll, rl), r) →
      let rmax = max (depth rl) (depth r) + 1 in
      let cmax = max rmax (depth ll) + 1 in
      Node (xl, cmax, ll, Node (x, rmax, rl, r))
  | _ → raise Invalid_input

let balanceLR tree =
  let _ = Raml.tick 1.0 in
  match tree with
  | Node
      (x, _, Node (y, _, ll, Node (z, _, lrl, lrr)), r)
    →
      let lmax = max (depth ll) (depth lrl) + 1 in
      let rmax = max (depth lrr) (depth r) + 1 in
      let cmax = max lmax rmax + 1 in
      Node
        ( z,
          cmax,
          Node (y, lmax, ll, lrl),
          Node (x, rmax, lrr, r) )
  | _ → raise Invalid_input

let balanceRR tree =
  let _ = Raml.tick 1.0 in
  match tree with
  | Node (x, _, l, Node (xr, _, lr, rr)) →
      let lmax = max (depth l) (depth lr) + 1 in
      let cmax = max lmax (depth rr) + 1 in
      Node (xr, cmax, Node (x, lmax, l, lr), rr)
  | _ → raise Invalid_input

let balanceRL tree =
  let _ = Raml.tick 1.0 in
  match tree with
  | Node
      (x, _, l, Node (y, _, Node (z, _, rll, rlr), rr))
    →
      let lmax = max (depth l) (depth rll) + 1 in
      let rmax = max (depth rlr) (depth rr) + 1 in
      let cmax = max lmax rmax + 1 in
      Node
        ( z,
          cmax,
          Node (x, lmax, l, rll),
          Node (y, rmax, rlr, rr) )
  | _ → raise Invalid_input

let rec avl_tree_insert v tree
    current_original_counters =
  let _ = Raml.tick 1.0 in
  let new_counter =
    decrement_first_counter current_original_counters
  in
  let result, counter_final =
    match tree with
    | Node (x, _, l, r) →
        if x = v then (tree, new_counter)
        else if v < x then
          let insL, counter1 =
            avl_tree_insert v l new_counter
          in
          let dl = depth insL in
          let dr = depth r in
          let bal = dl - dr in
          if bal < 2 || bal > 2 then
            ( Node (x, max dr dl + 1, insL, r),
              counter1 )
          else if v < value l then
            ( balanceLL (Node (x, dl + 1, insL, r)),
              counter1 )
          else if v > value l then
            ( balanceLR (Node (x, dl + 1, insL, r)),
              counter1 )
          else (tree, counter1)
        else
          let insR, counter1 =
            avl_tree_insert v r new_counter
          in
          let dr = depth insR in
          let dl = depth l in
          let bal = dl - dr in
          if bal < -2 || bal > -2 then
            ( Node (x, max dr dl + 1, l, insR),
              counter1 )
          else if v > value r then
            ( balanceRR (Node (x, dr + 1, l, insR)),
              counter1 )
          else if v < value r then
            ( balanceRL (Node (x, dr + 1, l, insR)),
              counter1 )
          else (tree, counter1)
    | Leaf → (Node (v, 1, Leaf, Leaf), new_counter)
  in
  (result, increment_first_counter counter_final)
```

Lst. B.29: Resource-guarded code of AVLTree (part 1).

```
let rec min tree =
  let _ = Raml.tick 1.0 in
  match tree with
  | Node (x, _, Leaf, _) →x
  | Node (_, _, l, _) →min l
  | Leaf → raise Invalid_input

let left_subtree tree =
  let _ = Raml.tick 1.0 in
  match tree with
  | Node (_, _, l, _) →l
  | Leaf → raise Invalid_input

let right_subtree tree =
  let _ = Raml.tick 1.0 in
  match tree with
  | Node (_, _, _, r) →r
  | Leaf → raise Invalid_input

let rec avl_tree_repeated_insert xs acc
    current_original_counters =
  let _ = Raml.tick 1.0 in
  match xs with
  | [] → (acc, current_original_counters)
  | hd :: tl →
      let initialized_counter =
        initialize_first_counter
          current_original_counters
      in
      let acc_updated, counter1 =
        avl_tree_insert hd acc initialized_counter
      in
      avl_tree_repeated_insert tl acc_updated counter1

let rec avl_tree_repeated_lookup xs tree
    current_original_counters =
  let _ = Raml.tick 1.0 in
  match xs with
  | [] → ([], current_original_counters)
  | hd :: tl →
      let initialized_counter =
        initialize_second_counter
          current_original_counters
      in
      let is_found, counter1 =
        avl_tree_lookup hd tree initialized_counter
      in
      let recursive_result, counter2 =
        avl_tree_repeated_lookup tl tree counter1
      in
      (is_found :: recursive_result, counter2)

(* Polynomial degree for AARA: 2 *)

let avl_tree_main xs1 xs2 current_original_counters =
  let _ = Raml.tick 1.0 in
  let tree, counter1 =
    avl_tree_repeated_insert xs1 Leaf
      current_original_counters
  in
  avl_tree_repeated_lookup xs2 tree counter1
```

Lst. B.30: Resource-guarded code of AVLTree (part 2).

B.2.10 SplayTree

```
exception Invalid_input

type splay_tree =
  | Leaf
  | Node of int * splay_tree * splay_tree

let extract_value_and_subtrees tree =
  let _ = Raml.tick 1.0 in
  match tree with
  | Leaf → raise Invalid_input
  | Node (x, left, right) →(x, left, right)

let rec splay_insert x tree =
  let _ = Raml.mark 0 1.0 in
  let _ = Raml.tick 1.0 in
  let result =
    match tree with
    | Leaf → Leaf
    | Node (y, l, r) →(
        if y = x then tree
        else if x < y then
          match l with
          | Leaf → tree
          | Node (z, ll, rr) →(
              if x = z then
                Node (z, ll, Node (y, rr, r))
              else if x < z then
                match ll with
                | Leaf → Node (z, ll, Node (y, rr, r))
                | _ →
                    let newV, newL, newR =
                      extract_value_and_subtrees
                        (splay_insert x ll)
                    in
                    Node
                      ( newV,
                        newL,
                        Node (z, newR, Node (y, rr, r))
                      )
              else
                match rr with
                | Leaf → Node (z, ll, Node (y, rr, r))
                | _ →
                    let newV, newL, newR =
                      extract_value_and_subtrees
                        (splay_insert x rr)
                    in
                    Node
                      ( newV,
                        Node (z, ll, newL),
                        Node (y, newR, r) ) )
        else
          match r with
          | Leaf → tree
          | Node (z, ll, rr) →(
              if x = z then
                Node (z, Node (y, l, ll), rr)
              else if x < z then
                match ll with
                | Leaf → Node (z, Node (y, l, ll), rr)
                | _ →
                    let newV, newL, newR =
                      extract_value_and_subtrees
                        (splay_insert x ll)
                    in
                    Node
                      ( newV,
                        Node (y, l, newL),
                        Node (z, newR, rr) )
              else
                match rr with
                | Leaf → Node (z, Node (y, l, ll), rr)
                | _ →
                    let newV, newL, newR =
                      extract_value_and_subtrees
                        (splay_insert x rr)
                    in
                    Node
                      ( newV,
                        Node (z, Node (y, l, ll), newL),
                        newR ) ) )
    in
    let _ = Raml.mark 0 (-1.0) in
    result

let splay_tree_insert x tree =
  let _ = Raml.tick 1.0 in
  match tree with
  | Leaf → Node (x, Leaf, Leaf)
  | _ →
      let _ = Raml.activate_counter_variable 0 in
      let tree_splayed = splay_insert x tree in
      let _ = Raml.record_counter_variable 0 in
      let y, l, r =
        extract_value_and_subtrees tree_splayed
      in
      if x = y then Node (y, l, r)
      else if x < y then Node (x, l, Node (y, Leaf, r))
      else Node (x, Node (y, l, Leaf), r)
```

Lst. B.31: Resource-decomposed code of SplayTree (part 1).

```
let rec splay_lookup x tree =
  let _ = Raml.mark 1 1.0 in
  let _ = Raml.tick 1.0 in
  let result =
    match tree with
    | Leaf → Leaf
    | Node (y, l, r) →(
        if y = x then tree
        else if x < y then
          match l with
          | Leaf → tree
          | Node (z, ll, rr) →(
              if x = z then
                Node (z, ll, Node (y, rr, r))
              else if x < z then
                match ll with
                | Leaf → Node (z, ll, Node (y, rr, r))
                | _ →
                    let newV, newL, newR =
                      extract_value_and_subtrees
                        (splay_lookup x ll)
                    in
                    Node
                      ( newV,
                        newL,
                        Node (z, newR, Node (y, rr, r))
                      )
              else
                match rr with
                | Leaf → Node (z, ll, Node (y, rr, r))
                | _ →
                    let newV, newL, newR =
                      extract_value_and_subtrees
                        (splay_lookup x rr)
                    in
                    Node
                      ( newV,
                        Node (z, ll, newL),
                        Node (y, newR, r) ) )
        else
          match r with
          | Leaf → tree
          | Node (z, ll, rr) →(
              if x = z then
                Node (z, Node (y, l, ll), rr)
              else if x < z then
                match ll with
                | Leaf → Node (z, Node (y, l, ll), rr)
                | _ →
                    let newV, newL, newR =
                      extract_value_and_subtrees
                        (splay_lookup x ll)
                    in
                    Node
                      ( newV,
                        Node (y, l, newL),
                        Node (z, newR, rr) )
              else
                match rr with
                | Leaf → Node (z, Node (y, l, ll), rr)
                | _ →
                    let newV, newL, newR =
                      extract_value_and_subtrees
                        (splay_lookup x rr)
                    in
                    Node
                      ( newV,
                        Node (z, Node (y, l, ll), newL),
                        newR ) ) )
  in
  let _ = Raml.mark 1 (-1.0) in
  result

let splay_tree_lookup v tree =
  let _ = Raml.tick 1.0 in
  let result =
    let _ = Raml.activate_counter_variable 1 in
    let tree_splayed = splay_lookup v tree in
    let _ = Raml.record_counter_variable 1 in
    let is_found =
      match tree_splayed with
      | Leaf → false
      | Node (x, _, _) →
          if x = v then true else false
    in
    (is_found, tree_splayed)
  in
  result

let rec splay_tree_repeated_insert xs acc =
  let _ = Raml.tick 1.0 in
  match xs with
  | [] → acc
  | hd :: tl →
      let acc_updated = splay_tree_insert hd acc in
      splay_tree_repeated_insert tl acc_updated

let rec splay_tree_repeated_lookup xs tree =
  let _ = Raml.tick 1.0 in
  match xs with
  | [] → ([], tree)
  | hd :: tl →
      let is_found, tree_updated =
        splay_tree_lookup hd tree
      in
      let recursive_result, tree_final =
        splay_tree_repeated_lookup tl tree_updated
      in
      (is_found :: recursive_result, tree_final)

let splay_tree_main xs1 xs2 =
  let _ = Raml.tick 1.0 in
  let tree = splay_tree_repeated_insert xs1 Leaf in
  splay_tree_repeated_lookup xs2 tree
```

Lst. B.32: Resource-decomposed code of SplayTree (part 2).

```
type splay_tree =
  | Leaf
  | Node of int * splay_tree * splay_tree

let extract_value_and_subtrees tree =
  let _ = Raml.tick 1.0 in
  match tree with
  | Leaf → raise Invalid_input
  | Node (x, left, right) →(x, left, right)

let rec splay_insert x tree current_original_counters
    =
  let _ = Raml.tick 1.0 in
  let new_counter =
    decrement_first_counter current_original_counters
  in
  let result, counter_final =
    match tree with
    | Leaf → (Leaf, new_counter)
    | Node (y, l, r) →(
        if y = x then (tree, new_counter)
        else if x < y then
          match l with
          | Leaf → (tree, new_counter)
          | Node (z, ll, rr) →(
              if x = z then
                ( Node (z, ll, Node (y, rr, r)),
                  new_counter )
              else if x < z then
                match ll with
                | Leaf →
                    ( Node (z, ll, Node (y, rr, r)),
                      new_counter )
                | _ →
                    let ll_splayed, counter1 =
                      splay_insert x ll new_counter
                    in
                    let newV, newL, newR =
                      extract_value_and_subtrees
                        ll_splayed
                    in
                    ( Node
                        ( newV,
                          newL,
                          Node
                            (z, newR, Node (y, rr, r))
                        ),
                      counter1 )
              else
                match rr with
                | Leaf →
                    ( Node (z, ll, Node (y, rr, r)),
                      new_counter )
                | _ →
                    let rr_splayed, counter1 =
                      splay_insert x rr new_counter
                    in
                    let newV, newL, newR =
                      extract_value_and_subtrees
                        rr_splayed
                    in
                    ( Node
                        ( newV,
                          Node (z, ll, newL),
                          Node (y, newR, r) ),
                      counter1 ) )
        else
          match r with
          | Leaf → (tree, new_counter)
          | Node (z, ll, rr) →(
              if x = z then
                ( Node (z, Node (y, l, ll), rr),
                  new_counter )
              else if x < z then
                match ll with
                | Leaf →
                    ( Node (z, Node (y, l, ll), rr),
                      new_counter )
                | _ →
                    let ll_splayed, counter1 =
                      splay_insert x ll new_counter
                    in
                    let newV, newL, newR =
                      extract_value_and_subtrees
                        ll_splayed
                    in
                    ( Node
                        ( newV,
                          Node (y, l, newL),
                          Node (z, newR, rr) ),
                      counter1 )
              else
                match rr with
                | Leaf →
                    ( Node (z, Node (y, l, ll), rr),
                      new_counter )
                | _ →
                    let rr_splayed, counter1 =
                      splay_insert x rr new_counter
                    in
                    let newV, newL, newR =
                      extract_value_and_subtrees
                        rr_splayed
                    in
                    ( Node
                        ( newV,
                          Node
                            (z, Node (y, l, ll), newL),
                          newR ),
                      counter1 ) ) )
  in
  (result, increment_first_counter counter_final)

let splay_tree_insert x tree current_original_counters
    =
  let _ = Raml.tick 1.0 in
  match tree with
  | Leaf →
      (Node (x, Leaf, Leaf), current_original_counters)
  | _ →
      let initialized_counter =
        initialize_first_counter
          current_original_counters
      in
      let tree_splayed, counter1 =
        splay_insert x tree initialized_counter
      in
      let counter2 =
        set_first_counter_to_zero counter1
      in
      let y, l, r =
        extract_value_and_subtrees tree_splayed
      in
      if x = y then (Node (y, l, r), counter2)
      else if x < y then
        (Node (x, l, Node (y, Leaf, r)), counter2)
      else (Node (x, Node (y, l, Leaf), r), counter2)
```

Lst. B.33: Resource-guarded code of SplayTree (part 1)

```
let rec splay_lookup x tree current_original_counters
    =
  let _ = Raml.tick 1.0 in
  let new_counter =
    decrement_second_counter current_original_counters
  in
  let result, counter_final =
    match tree with
    | Leaf → (Leaf, new_counter)
    | Node (y, l, r) →(
        if y = x then (tree, new_counter)
        else if x < y then
          match l with
          | Leaf → (tree, new_counter)
          | Node (z, ll, rr) →(
              if x = z then
                ( Node (z, ll, Node (y, rr, r)),
                  new_counter )
              else if x < z then
                match ll with
                | Leaf →
                    ( Node (z, ll, Node (y, rr, r)),
                      new_counter )
                | _ →
                    let ll_splayed, counter1 =
                      splay_lookup x ll new_counter
                    in
                    let newV, newL, newR =
                      extract_value_and_subtrees
                        ll_splayed
                    in
                    ( Node
                        ( newV,
                          newL,
                          Node
                            (z, newR, Node (y, rr, r))
                        ),
                      counter1 )
              else
                match rr with
                | Leaf →
                    ( Node (z, ll, Node (y, rr, r)),
                      new_counter )
                | _ →
                    let rr_splayed, counter1 =
                      splay_lookup x rr new_counter
                    in
                    let newV, newL, newR =
                      extract_value_and_subtrees
                        rr_splayed
                    in
                    ( Node
                        ( newV,
                          Node (z, ll, newL),
                          Node (y, newR, r) ),
                      counter1 ) )
```

```
        else
          match r with
          | Leaf → (tree, new_counter)
          | Node (z, ll, rr) →(
              if x = z then
                ( Node (z, Node (y, l, ll), rr),
                  new_counter )
              else if x < z then
                match ll with
                | Leaf →
                    ( Node (z, Node (y, l, ll), rr),
                      new_counter )
                | _ →
                    let ll_splayed, counter1 =
                      splay_lookup x ll new_counter
                    in
                    let newV, newL, newR =
                      extract_value_and_subtrees
                        ll_splayed
                    in
                    ( Node
                        ( newV,
                          Node (y, l, newL),
                          Node (z, newR, rr) ),
                      counter1 )
              else
                match rr with
                | Leaf →
                    ( Node (z, Node (y, l, ll), rr),
                      new_counter )
                | _ →
                    let rr_splayed, counter1 =
                      splay_lookup x rr new_counter
                    in
                    let newV, newL, newR =
                      extract_value_and_subtrees
                        rr_splayed
                    in
                    ( Node
                        ( newV,
                          Node
                            (z, Node (y, l, ll), newL),
                          newR ),
                      counter1 ) ) )
  in
  (result, increment_second_counter counter_final)

let splay_tree_lookup v tree current_original_counters
    =
  let _ = Raml.tick 1.0 in
  let initialized_counter =
    initialize_second_counter
      current_original_counters
  in
  let tree_splayed, counter1 =
    splay_lookup v tree initialized_counter
  in
  let counter2 =
    set_second_counter_to_zero counter1
  in
  let is_found =
    match tree_splayed with
    | Leaf → false
    | Node (x, _, _) →if x = v then true else false
  in
  ((is_found, tree_splayed), counter2)
```

Lst. B.34: Resource-guarded code of SplayTree (part 2).

```
let rec splay_tree_repeated_insert xs acc
    current_original_counters =
  let _ = Raml.tick 1.0 in
  match xs with
  | [] → (acc, current_original_counters)
  | hd :: tl →
      let acc_updated, counter1 =
        splay_tree_insert hd acc
          current_original_counters
      in
      splay_tree_repeated_insert tl acc_updated
        counter1

let rec splay_tree_repeated_lookup xs tree
    current_original_counters =
  let _ = Raml.tick 1.0 in
  match xs with
  | [] → (([], tree), current_original_counters)
  | hd :: tl →
      let (is_found, tree_updated), counter1 =
        splay_tree_lookup hd tree
          current_original_counters
      in
      let (recursive_result, tree_final), counter2 =
        splay_tree_repeated_lookup tl tree_updated
          counter1
      in
      ( (is_found :: recursive_result, tree_final),
        counter2 )
```

```
(* Polynomial degree for AARA: 2 *)

let splay_tree_main xs1 xs2 current_original_counters
    =
  let _ = Raml.tick 1.0 in
  let tree, counter1 =
    splay_tree_repeated_insert xs1 Leaf
      current_original_counters
  in
  splay_tree_repeated_lookup xs2 tree counter1
```

Lst. B.35: Resource-guarded code of SplayTree (part 3).

B.2.11 Prim

```
exception Invalid_input

type distance = Infinity | Some of float

let is_shorter_distance distance1 distance2 =
  let _ = Raml.tick 1.0 in
  match (distance1, distance2) with
  | Some d1, Some d2 →d1 < d2
  | Some _, Infinity →true
  | _, _ → false

type vertex_dist_pair =
  | Vertex_dist of int * distance

let vertex_pair p =
  let _ = Raml.tick 1.0 in
  match p with Vertex_dist (v, _) →v

let distance_pair p =
  let _ = Raml.tick 1.0 in
  match p with Vertex_dist (_, d) →d

type binary_heap =
  vertex_dist_pair Rarray.t * Rnat.t * int Rarray.t

let rec heapify heap (index : int) =
  let _ = Raml.mark 0 1.0 in
  let _ = Raml.tick 1.0 in
  let result =
    let left_index = (index * 2) + 1
    and right_index = (index * 2) + 2 in
    let array, length, map_vertices_to_indices =
      heap
    in
    let smallest_index_left =
      if left_index < Rnat.to_int length then
        let element_index =
          Rarray.get array (Rnat.of_int index)
        in
        let element_left_index =
          Rarray.get array (Rnat.of_int left_index)
        in
        if
          is_shorter_distance
            (distance_pair element_left_index)
            (distance_pair element_index)
        then left_index
        else index
      else index
    in
    let smallest_index_right =
      if right_index < Rnat.to_int length then
        let element_smallest_index_left =
          Rarray.get array
            (Rnat.of_int smallest_index_left)
        in
        let element_right_index =
          Rarray.get array (Rnat.of_int right_index)
        in
        if
          is_shorter_distance
            (distance_pair element_right_index)
            (distance_pair
               element_smallest_index_left)
        then right_index
        else smallest_index_left
      else smallest_index_left
    in
    if smallest_index_right = index then ()
    else
      let element_at_index =
        Rarray.get array (Rnat.of_int index)
      in
      let (Vertex_dist (v_index, _)) =
        element_at_index
      in
      let element_at_smallest_index =
        Rarray.get array
          (Rnat.of_int smallest_index_right)
      in
      let (Vertex_dist (v_smallest_index, _)) =
        element_at_smallest_index
      in
      let _ =
        Rarray.set array
          (Rnat.of_int index)
          element_at_smallest_index
      in
      let _ =
        Rarray.set array
          (Rnat.of_int smallest_index_right)
          element_at_index
      in
      let _ =
        Rarray.set map_vertices_to_indices
          (Rnat.of_int v_index)
          smallest_index_right
      in
      let _ =
        Rarray.set map_vertices_to_indices
          (Rnat.of_int v_smallest_index)
          index
      in
      heapify heap smallest_index_right
  in
  let _ = Raml.mark 0 (-1.0) in
  result

let get_min heap =
  let _ = Raml.tick 1.0 in
  let array, _, _ = heap in
  Rarray.get array Rnat.zero
```

Lst. B.36: Resource-decomposed code of Prim (part 1).

```
let delete_min heap =
  let _ = Raml.tick 1.0 in
  let array, length, map_vertices_to_indices = heap in
  Rnat.ifz length
    (fun () → raise Invalid_input)
    (fun length_minus_one →
      let first_element =
        Rarray.get array Rnat.zero
      in
      let (Vertex_dist (v_min, _)) = first_element in
      let last_element =
        Rarray.get array length_minus_one
      in
      let (Vertex_dist (v_last, _)) = last_element in
      let _ =
        Rarray.set array Rnat.zero last_element
      in
      let _ =
        Rarray.set map_vertices_to_indices
          (Rnat.of_int v_min)
          (-1)
      in
      let _ =
        Rarray.set map_vertices_to_indices
          (Rnat.of_int v_last)
          0
      in
      let _ = Raml.activate_counter_variable 0 in
      let _ = heapify heap 0 in
      let _ = Raml.record_counter_variable 0 in
      ( array,
        length_minus_one,
        map_vertices_to_indices ))

let rec decrease_key_helper heap index =
  let _ = Raml.mark 1 1.0 in
  let _ = Raml.tick 1.0 in
  let result =
    let array, _, map_vertices_to_indices = heap in
    if index = 0 then ()
    else
      let element_index =
        Rarray.get array (Rnat.of_int index)
      in
      let parent_index = (index - 1) / 2 in
      let parent_element =
        Rarray.get array (Rnat.of_int parent_index)
      in
      if
        is_shorter_distance
          (distance_pair element_index)
          (distance_pair parent_element)
      then
        let (Vertex_dist (v, _)) = element_index in
        let (Vertex_dist (v_parent, _)) =
          parent_element
        in
        let _ =
          Rarray.set array
            (Rnat.of_int index)
            parent_element
        in
        let _ =
          Rarray.set array
            (Rnat.of_int parent_index)
            element_index
        in
        let _ =
          Rarray.set map_vertices_to_indices
            (Rnat.of_int v) parent_index
        in
        let _ =
          Rarray.set map_vertices_to_indices
            (Rnat.of_int v_parent)
            index
        in
        decrease_key_helper heap parent_index
      else ()
  in
  let _ = Raml.mark 1 (-1.0) in
  result

let decrease_key heap vertex new_dist =
  let _ = Raml.tick 1.0 in
  let array, _, map_vertices_to_indices = heap in
  let array_index =
    Rarray.get map_vertices_to_indices
      (Rnat.of_int vertex)
  in
  if array_index = -1 then ()
  else
    let (Vertex_dist (_, dist)) =
      Rarray.get array (Rnat.of_int array_index)
    in
    if is_shorter_distance new_dist dist then
      let _ =
        Rarray.set array
          (Rnat.of_int array_index)
          (Vertex_dist (vertex, new_dist))
      in
      let _ = Raml.activate_counter_variable 1 in
      let result =
        decrease_key_helper heap array_index
      in
      let _ = Raml.record_counter_variable 1 in
      result
    else ()
```

Lst. B.37: Resource-decomposed code of Prim (part 2).

```
let rec initialize_array array index_nat =
  let _ = Raml.tick 1.0 in
  Rnat.ifz index_nat
    (fun () → ())
    (fun index_minus_one →
      let _ =
        Rarray.set array index_minus_one
          (Vertex_dist
             (Rnat.to_int index_minus_one, Infinity))
      in
      initialize_array array index_minus_one)

let rec initialize_map_vertices_to_indices array
    index_nat =
  let _ = Raml.tick 1.0 in
  Rnat.ifz index_nat
    (fun () → ())
    (fun index_minus_one →
      let _ =
        Rarray.set array index_minus_one
          (Rnat.to_int index_minus_one)
      in
      initialize_map_vertices_to_indices array
        index_minus_one)

let construct_initial_heap adjacency_list =
  let _ = Raml.tick 1.0 in
  let num_vertices = Rarray.length adjacency_list in
  let array =
    Rarray.make num_vertices
      (Vertex_dist (-1, Infinity))
  in
  let _ = initialize_array array num_vertices in
  let map_vertices_to_indices =
    Rarray.make num_vertices (-1)
  in
  let _ =
    initialize_map_vertices_to_indices
      map_vertices_to_indices num_vertices
  in
  (array, num_vertices, map_vertices_to_indices)
```

```
let extract_neighbors adjacency_list (vertex : int) =
  let _ = Raml.tick 1.0 in
  let list_neighbors =
    Rarray.get adjacency_list (Rnat.of_int vertex)
  in
  list_neighbors

let rec update_dist_all_neighbors heap list_neighbors
    =
  let _ = Raml.mark 2 1.0 in
  let _ = Raml.tick 1.0 in
  let result =
    match list_neighbors with
    | [] → ()
    | (vertex, dist) :: tl →
        let _ =
          decrease_key heap vertex (Some dist)
        in
        update_dist_all_neighbors heap tl
  in
  let _ = Raml.mark 2 (-1.0) in
  result

let rec repeatedly_get_min_node adjacency_list heap
    acc =
  let _ = Raml.tick 1.0 in
  let _, length, _ = heap in
  if Rnat.to_int length = 0 then acc
  else
    let min_node = get_min heap in
    let heap_updated = delete_min heap in
    let (Vertex_dist (vertex, dist)) = min_node in
    let list_neighbors =
      extract_neighbors adjacency_list vertex
    in
    let _ = Raml.activate_counter_variable 2 in
    let _ =
      update_dist_all_neighbors heap list_neighbors
    in
    let _ = Raml.record_counter_variable 2 in
    let acc_updated = (vertex, dist) :: acc in
    repeatedly_get_min_node adjacency_list
      heap_updated acc_updated

let prim_algorithm adjacency_list =
  let _ = Raml.tick 1.0 in
  let heap = construct_initial_heap adjacency_list in
  repeatedly_get_min_node adjacency_list heap []
```

Lst. B.38: Resource-decomposed code of Prim (part 3).

```
type distance = Infinity | Some of float

let is_shorter_distance distance1 distance2 =
  let _ = Raml.tick 1.0 in
  match (distance1, distance2) with
  | Some d1, Some d2 →d1 < d2
  | Some _, Infinity →true
  | _, _ → false

type vertex_dist_pair =
  | Vertex_dist of int * distance

let vertex_pair p =
  let _ = Raml.tick 1.0 in
  match p with Vertex_dist (v, _) →v

let distance_pair p =
  let _ = Raml.tick 1.0 in
  match p with Vertex_dist (_, d) →d

type binary_heap =
  vertex_dist_pair Rarray.t * Rnat.t * int Rarray.t

let rec heapify heap (index : int)
    three_current_original_counters =
  let new_counter =
    decrement_first_counter
      three_current_original_counters
  in
  let result, counter_final =
    let _ = Raml.tick 1.0 in
    let left_index = (index * 2) + 1
    and right_index = (index * 2) + 2 in
    let array, length, map_vertices_to_indices =
      heap
    in
    let smallest_index_left =
      if left_index < Rnat.to_int length then
        let element_index =
          Rarray.get array (Rnat.of_int index)
        in
        let element_left_index =
          Rarray.get array (Rnat.of_int left_index)
        in
        if
          is_shorter_distance
            (distance_pair element_left_index)
            (distance_pair element_index)
        then left_index
        else index
      else index
    in
    let smallest_index_right =
      if right_index < Rnat.to_int length then
        let element_smallest_index_left =
          Rarray.get array
            (Rnat.of_int smallest_index_left)
        in
        let element_right_index =
          Rarray.get array (Rnat.of_int right_index)
        in
        if
          is_shorter_distance
            (distance_pair element_right_index)
            (distance_pair
               element_smallest_index_left)
        then right_index
        else smallest_index_left
      else smallest_index_left
    in
    if smallest_index_right = index then
      ((), new_counter)
    else
      let element_at_index =
        Rarray.get array (Rnat.of_int index)
      in
      let (Vertex_dist (v_index, _)) =
        element_at_index
      in
      let element_at_smallest_index =
        Rarray.get array
          (Rnat.of_int smallest_index_right)
      in
      let (Vertex_dist (v_smallest_index, _)) =
        element_at_smallest_index
      in
      let _ =
        Rarray.set array
          (Rnat.of_int index)
          element_at_smallest_index
      in
      let _ =
        Rarray.set array
          (Rnat.of_int smallest_index_right)
          element_at_index
      in
      let _ =
        Rarray.set map_vertices_to_indices
          (Rnat.of_int v_index)
          smallest_index_right
      in
      let _ =
        Rarray.set map_vertices_to_indices
          (Rnat.of_int v_smallest_index)
          index
      in
      heapify heap smallest_index_right new_counter
  in
  (result, increment_first_counter counter_final)

let get_min heap =
  let _ = Raml.tick 1.0 in
  let array, _, _ = heap in
  Rarray.get array Rnat.zero
```

Lst. B.39: Resource-guarded code of Prim (part 1).

```
let delete_min heap three_current_original_counters =
  let _ = Raml.tick 1.0 in
  let array, length, map_vertices_to_indices = heap in
  Rnat.ifz length
    (fun () → raise Invalid_input)
    (fun length_minus_one →
      let first_element =
        Rarray.get array Rnat.zero
      in
      let (Vertex_dist (v_min, _)) = first_element in
      let last_element =
        Rarray.get array length_minus_one
      in
      let (Vertex_dist (v_last, _)) = last_element in
      let _ =
        Rarray.set array Rnat.zero last_element
      in
      let _ =
        Rarray.set map_vertices_to_indices
          (Rnat.of_int v_min)
          (-1)
      in
      let _ =
        Rarray.set map_vertices_to_indices
          (Rnat.of_int v_last)
          0
      in
      let initialized_counter =
        initialize_first_counter
          three_current_original_counters
      in
      let _, counter_final =
        heapify heap 0 initialized_counter
      in
      ( ( array,
          length_minus_one,
          map_vertices_to_indices ),
        set_first_counter_to_zero counter_final ))

let rec decrease_key_helper heap index
    three_current_original_counters =
  let new_counter =
    decrement_second_counter
      three_current_original_counters
  in
  let result, counter_final =
    let _ = Raml.tick 1.0 in
    let array, _, map_vertices_to_indices = heap in
    if index = 0 then ((), new_counter)
    else
      let element_index =
        Rarray.get array (Rnat.of_int index)
      in
      let parent_index = (index - 1) / 2 in
      let parent_element =
        Rarray.get array (Rnat.of_int parent_index)
      in
      if
        is_shorter_distance
          (distance_pair element_index)
          (distance_pair parent_element)
      then
        let (Vertex_dist (v, _)) = element_index in
        let (Vertex_dist (v_parent, _)) =
          parent_element
        in
        let _ =
          Rarray.set array
            (Rnat.of_int index)
            parent_element
        in
        let _ =
          Rarray.set array
            (Rnat.of_int parent_index)
            element_index
        in
        let _ =
          Rarray.set map_vertices_to_indices
            (Rnat.of_int v) parent_index
        in
        let _ =
          Rarray.set map_vertices_to_indices
            (Rnat.of_int v_parent)
            index
        in
        decrease_key_helper heap parent_index
          new_counter
      else ((), new_counter)
  in
  (result, increment_second_counter counter_final)

let decrease_key heap vertex new_dist
    three_current_original_counters =
  let _ = Raml.tick 1.0 in
  let array, _, map_vertices_to_indices = heap in
  let array_index =
    Rarray.get map_vertices_to_indices
      (Rnat.of_int vertex)
  in
  if array_index = -1 then
    ((), three_current_original_counters)
  else
    let (Vertex_dist (_, dist)) =
      Rarray.get array (Rnat.of_int array_index)
    in
    if is_shorter_distance new_dist dist then
      let _ =
        Rarray.set array
          (Rnat.of_int array_index)
          (Vertex_dist (vertex, new_dist))
      in
      let initialized_counter =
        initialize_second_counter
          three_current_original_counters
      in
      let result, counter_final =
        decrease_key_helper heap array_index
          initialized_counter
      in
      ( result,
        set_second_counter_to_zero counter_final )
    else ((), three_current_original_counters)
```

Lst. B.40: Resource-guarded code of Prim (part 2).

```
let rec initialize_array array index_nat =
  let _ = Raml.tick 1.0 in
  Rnat.ifz index_nat
    (fun () → ())
    (fun index_minus_one →
      let _ =
        Rarray.set array index_minus_one
          (Vertex_dist
             (Rnat.to_int index_minus_one, Infinity))
      in
      initialize_array array index_minus_one)

let rec initialize_map_vertices_to_indices array
    index_nat =
  let _ = Raml.tick 1.0 in
  Rnat.ifz index_nat
    (fun () → ())
    (fun index_minus_one →
      let _ =
        Rarray.set array index_minus_one
          (Rnat.to_int index_minus_one)
      in
      initialize_map_vertices_to_indices array
        index_minus_one)

let construct_initial_heap adjacency_list =
  let _ = Raml.tick 1.0 in
  let num_vertices = Rarray.length adjacency_list in
  let array =
    Rarray.make num_vertices
      (Vertex_dist (-1, Infinity))
  in
  let _ = initialize_array array num_vertices in
  let map_vertices_to_indices =
    Rarray.make num_vertices (-1)
  in
  let _ =
    initialize_map_vertices_to_indices
      map_vertices_to_indices num_vertices
  in
  (array, num_vertices, map_vertices_to_indices)

let extract_neighbors adjacency_list (vertex : int) =
  let _ = Raml.tick 1.0 in
  let list_neighbors =
    Rarray.get adjacency_list (Rnat.of_int vertex)
  in
  list_neighbors

let rec update_dist_all_neighbors heap list_neighbors
    three_current_original_counters =
  let new_counter =
    decrement_third_counter
      three_current_original_counters
  in
  let result, counter_final =
    let _ = Raml.tick 1.0 in
    match list_neighbors with
    | [] → ((), new_counter)
    | (vertex, dist) :: tl →
        let _, counter1 =
          decrease_key heap vertex (Some dist)
            new_counter
        in
        update_dist_all_neighbors heap tl counter1
  in
  (result, increment_third_counter counter_final)

let rec repeatedly_get_min_node adjacency_list heap
    acc three_current_original_counters =
  let _ = Raml.tick 1.0 in
  let _, length, _ = heap in
  if Rnat.to_int length = 0 then
    (acc, three_current_original_counters)
  else
    let min_node = get_min heap in
    let heap_updated, counter1 =
      delete_min heap three_current_original_counters
    in
    let (Vertex_dist (vertex, dist)) = min_node in
    let list_neighbors =
      extract_neighbors adjacency_list vertex
    in
    let initialized_counter =
      initialize_third_counter counter1
    in
    let _, counter2 =
      update_dist_all_neighbors heap list_neighbors
        initialized_counter
    in
    let counter3 =
      set_third_counter_to_zero counter2
    in
    let acc_updated = (vertex, dist) :: acc in
    repeatedly_get_min_node adjacency_list
      heap_updated acc_updated counter3

(* Polynomial degree for AARA: 3 *)

let prim_algorithm adjacency_list
    three_current_original_counters =
  let _ = Raml.tick 1.0 in
  let heap = construct_initial_heap adjacency_list in
  repeatedly_get_min_node adjacency_list heap []
    three_current_original_counters
```

Lst. B.41: Resource-guarded code of Prim (part 3).

B.2.12 Dijkstra

```
exception Invalid_input

type distance = Infinity | Some of float

let add_distances shortest_distance weight =
  let _ = Raml.tick 1.0 in
  match shortest_distance with
  | Infinity → Infinity
  | Some d → Some (d +. weight)

let is_shorter_distance distance1 distance2 =
  let _ = Raml.tick 1.0 in
  match (distance1, distance2) with
  | Some d1, Some d2 →d1 < d2
  | Some _, Infinity →true
  | _, _ → false

type vertex_dist_pair =
  | Vertex_dist of int * distance

let vertex_pair p =
  let _ = Raml.tick 1.0 in
  match p with Vertex_dist (v, _) →v

let distance_pair p =
  let _ = Raml.tick 1.0 in
  match p with Vertex_dist (_, d) →d

type binary_heap =
  vertex_dist_pair Rarray.t * Rnat.t * int Rarray.t

let rec heapify heap (index : int) =
  let _ = Raml.mark 0 1.0 in
  let _ = Raml.tick 1.0 in
  let result =
    let left_index = (index * 2) + 1
    and right_index = (index * 2) + 2 in
    let array, length, map_vertices_to_indices =
      heap
    in
    let smallest_index_left =
      if left_index < Rnat.to_int length then
        let element_index =
          Rarray.get array (Rnat.of_int index)
        in
        let element_left_index =
          Rarray.get array (Rnat.of_int left_index)
        in
        if
          is_shorter_distance
            (distance_pair element_left_index)
            (distance_pair element_index)
        then left_index
        else index
      else index
    in
    let smallest_index_right =
      if right_index < Rnat.to_int length then
        let element_smallest_index_left =
          Rarray.get array
            (Rnat.of_int smallest_index_left)
        in
        let element_right_index =
          Rarray.get array (Rnat.of_int right_index)
        in
        if
          is_shorter_distance
            (distance_pair element_right_index)
            (distance_pair
               element_smallest_index_left)
        then right_index
        else smallest_index_left
      else smallest_index_left
    in
    if smallest_index_right = index then ()
    else
      let element_at_index =
        Rarray.get array (Rnat.of_int index)
      in
      let (Vertex_dist (v_index, _)) =
        element_at_index
      in
      let element_at_smallest_index =
        Rarray.get array
          (Rnat.of_int smallest_index_right)
      in
      let (Vertex_dist (v_smallest_index, _)) =
        element_at_smallest_index
      in
      let _ =
        Rarray.set array
          (Rnat.of_int index)
          element_at_smallest_index
      in
      let _ =
        Rarray.set array
          (Rnat.of_int smallest_index_right)
          element_at_index
      in
      let _ =
        Rarray.set map_vertices_to_indices
          (Rnat.of_int v_index)
          smallest_index_right
      in
      let _ =
        Rarray.set map_vertices_to_indices
          (Rnat.of_int v_smallest_index)
          index
      in
      heapify heap smallest_index_right
  in
  let _ = Raml.mark 0 (-1.0) in
  result

let get_min heap =
  let _ = Raml.tick 1.0 in
  let array, _, _ = heap in
  Rarray.get array Rnat.zero
```

Lst. B.42: Resource-decomposed code of Dijkstra (part 1).

```
let delete_min heap =
  let _ = Raml.tick 1.0 in
  let array, length, map_vertices_to_indices = heap in
  Rnat.ifz length
    (fun () → raise Invalid_input)
    (fun length_minus_one →
      let first_element =
        Rarray.get array Rnat.zero
      in
      let (Vertex_dist (v_min, _)) = first_element in
      let last_element =
        Rarray.get array length_minus_one
      in
      let (Vertex_dist (v_last, _)) = last_element in
      let _ =
        Rarray.set array Rnat.zero last_element
      in
      let _ =
        Rarray.set map_vertices_to_indices
          (Rnat.of_int v_min)
          (-1)
      in
      let _ =
        Rarray.set map_vertices_to_indices
          (Rnat.of_int v_last)
          0
      in
      let _ = Raml.activate_counter_variable 0 in
      let _ = heapify heap 0 in
      let _ = Raml.record_counter_variable 0 in
      ( array,
        length_minus_one,
        map_vertices_to_indices ))

let rec decrease_key_helper heap index =
  let _ = Raml.mark 1 1.0 in
  let _ = Raml.tick 1.0 in
  let result =
    let array, _, map_vertices_to_indices = heap in
    if index = 0 then ()
    else
      let element_index =
        Rarray.get array (Rnat.of_int index)
      in
      let parent_index = (index - 1) / 2 in
      let parent_element =
        Rarray.get array (Rnat.of_int parent_index)
      in
      if
        is_shorter_distance
          (distance_pair element_index)
          (distance_pair parent_element)
      then
        let (Vertex_dist (v, _)) = element_index in
        let (Vertex_dist (v_parent, _)) =
          parent_element
        in
        let _ =
          Rarray.set array
            (Rnat.of_int index)
            parent_element
        in
        let _ =
          Rarray.set array
            (Rnat.of_int parent_index)
            element_index
        in
        let _ =
          Rarray.set map_vertices_to_indices
            (Rnat.of_int v) parent_index
        in
        let _ =
          Rarray.set map_vertices_to_indices
            (Rnat.of_int v_parent)
            index
        in
        decrease_key_helper heap parent_index
      else ()
  in
  let _ = Raml.mark 1 (-1.0) in
  result

let decrease_key heap vertex new_dist =
  let _ = Raml.tick 1.0 in
  let array, _, map_vertices_to_indices = heap in
  let array_index =
    Rarray.get map_vertices_to_indices
      (Rnat.of_int vertex)
  in
  if array_index = -1 then ()
  else
    let (Vertex_dist (_, dist)) =
      Rarray.get array (Rnat.of_int array_index)
    in
    if is_shorter_distance new_dist dist then
      let _ =
        Rarray.set array
          (Rnat.of_int array_index)
          (Vertex_dist (vertex, new_dist))
      in
      let _ = Raml.activate_counter_variable 1 in
      let result =
        decrease_key_helper heap array_index
      in
      let _ = Raml.record_counter_variable 1 in
      result
    else ()

let rec initialize_array array index_nat =
  let _ = Raml.tick 1.0 in
  Rnat.ifz index_nat
    (fun () → ())
    (fun index_minus_one →
      let _ =
        Rnat.ifz index_minus_one
          (fun () →
            Rarray.set array index_minus_one
              (Vertex_dist
                 (Rnat.to_int index_minus_one, Some 0.)))
          (fun _ →
            Rarray.set array index_minus_one
              (Vertex_dist
                 ( Rnat.to_int index_minus_one,
                   Infinity )))
      in
      initialize_array array index_minus_one)
```

Lst. B.43: Resource-decomposed code of Dijkstra (part 2).

```
let rec initialize_map_vertices_to_indices array
    index_nat =
  let _ = Raml.tick 1.0 in
  Rnat.ifz index_nat
    (fun () → ())
    (fun index_minus_one →
      let _ =
        Rarray.set array index_minus_one
          (Rnat.to_int index_minus_one)
      in
      initialize_map_vertices_to_indices array
        index_minus_one)

let construct_initial_heap adjacency_list =
  let _ = Raml.tick 1.0 in
  let num_vertices = Rarray.length adjacency_list in
  let array =
    Rarray.make num_vertices
      (Vertex_dist (-1, Infinity))
  in
  let _ = initialize_array array num_vertices in
  let map_vertices_to_indices =
    Rarray.make num_vertices (-1)
  in
  let _ =
    initialize_map_vertices_to_indices
      map_vertices_to_indices num_vertices
  in
  (array, num_vertices, map_vertices_to_indices)

let extract_neighbors adjacency_list (vertex : int) =
  let _ = Raml.tick 1.0 in
  let list_neighbors =
    Rarray.get adjacency_list (Rnat.of_int vertex)
  in
  list_neighbors
```

```
let rec update_dist_all_neighbors heap list_neighbors
    base_dist =
  let _ = Raml.mark 2 1.0 in
  let _ = Raml.tick 1.0 in
  let result =
    match list_neighbors with
    | [] → ()
    | (vertex, dist) :: tl →
        let _ =
          decrease_key heap vertex
            (add_distances base_dist dist)
        in
        update_dist_all_neighbors heap tl base_dist
  in
  let _ = Raml.mark 2 (-1.0) in
  result

let rec repeatedly_get_min_node adjacency_list heap
    acc =
  let _ = Raml.tick 1.0 in
  let _, length, _ = heap in
  if Rnat.to_int length = 0 then acc
  else
    let min_node = get_min heap in
    let heap_updated = delete_min heap in
    let (Vertex_dist (vertex, dist)) = min_node in
    let list_neighbors =
      extract_neighbors adjacency_list vertex
    in
    let _ = Raml.activate_counter_variable 2 in
    let _ =
      update_dist_all_neighbors heap list_neighbors
        dist
    in
    let _ = Raml.record_counter_variable 2 in
    let acc_updated = (vertex, dist) :: acc in
    repeatedly_get_min_node adjacency_list
      heap_updated acc_updated

let dijkstra_algorithm adjacency_list =
  let _ = Raml.tick 1.0 in
  let heap = construct_initial_heap adjacency_list in
  repeatedly_get_min_node adjacency_list heap []
```

Lst. B.44: Resource-decomposed code of Dijkstra (part 3).

```
type distance = Infinity | Some of float

let add_distances shortest_distance weight =
  let _ = Raml.tick 1.0 in
  match shortest_distance with
  | Infinity → Infinity
  | Some d → Some (d +. weight)

let is_shorter_distance distance1 distance2 =
  let _ = Raml.tick 1.0 in
  match (distance1, distance2) with
  | Some d1, Some d2 →d1 < d2
  | Some _, Infinity →true
  | _, _ → false

type vertex_dist_pair =
  | Vertex_dist of int * distance

let vertex_pair p =
  let _ = Raml.tick 1.0 in
  match p with Vertex_dist (v, _) →v

let distance_pair p =
  let _ = Raml.tick 1.0 in
  match p with Vertex_dist (_, d) →d

type binary_heap =
  vertex_dist_pair Rarray.t * Rnat.t * int Rarray.t

let rec heapify heap (index : int)
    three_current_original_counters =
  let new_counter =
    decrement_first_counter
      three_current_original_counters
  in
  let result, counter_final =
    let _ = Raml.tick 1.0 in
    let left_index = (index * 2) + 1
    and right_index = (index * 2) + 2 in
    let array, length, map_vertices_to_indices =
      heap
    in
    let smallest_index_left =
      if left_index < Rnat.to_int length then
        let element_index =
          Rarray.get array (Rnat.of_int index)
        in
        let element_left_index =
          Rarray.get array (Rnat.of_int left_index)
        in
        if
          is_shorter_distance
            (distance_pair element_left_index)
            (distance_pair element_index)
        then left_index
        else index
      else index
    in
    let smallest_index_right =
      if right_index < Rnat.to_int length then
        let element_smallest_index_left =
          Rarray.get array
            (Rnat.of_int smallest_index_left)
        in
        let element_right_index =
          Rarray.get array (Rnat.of_int right_index)
        in
        if
          is_shorter_distance
            (distance_pair element_right_index)
            (distance_pair
               element_smallest_index_left)
        then right_index
        else smallest_index_left
      else smallest_index_left
    in
    if smallest_index_right = index then
      ((), new_counter)
    else
      let element_at_index =
        Rarray.get array (Rnat.of_int index)
      in
      let (Vertex_dist (v_index, _)) =
        element_at_index
      in
      let element_at_smallest_index =
        Rarray.get array
          (Rnat.of_int smallest_index_right)
      in
      let (Vertex_dist (v_smallest_index, _)) =
        element_at_smallest_index
      in
      let _ =
        Rarray.set array
          (Rnat.of_int index)
          element_at_smallest_index
      in
      let _ =
        Rarray.set array
          (Rnat.of_int smallest_index_right)
          element_at_index
      in
      let _ =
        Rarray.set map_vertices_to_indices
          (Rnat.of_int v_index)
          smallest_index_right
      in
      let _ =
        Rarray.set map_vertices_to_indices
          (Rnat.of_int v_smallest_index)
          index
      in
      heapify heap smallest_index_right new_counter
  in
  (result, increment_first_counter counter_final)

let get_min heap =
  let _ = Raml.tick 1.0 in
  let array, _, _ = heap in
  Rarray.get array Rnat.zero
```

Lst. B.45: Resource-guarded code of Dijkstra (part 1).

```
let delete_min heap three_current_original_counters =
  let _ = Raml.tick 1.0 in
  let array, length, map_vertices_to_indices = heap in
  Rnat.ifz length
    (fun () → raise Invalid_input)
    (fun length_minus_one →
      let first_element =
        Rarray.get array Rnat.zero
      in
      let (Vertex_dist (v_min, _)) = first_element in
      let last_element =
        Rarray.get array length_minus_one
      in
      let (Vertex_dist (v_last, _)) = last_element in
      let _ =
        Rarray.set array Rnat.zero last_element
      in
      let _ =
        Rarray.set map_vertices_to_indices
          (Rnat.of_int v_min)
          (-1)
      in
      let _ =
        Rarray.set map_vertices_to_indices
          (Rnat.of_int v_last)
          0
      in
      let initialized_counter =
        initialize_first_counter
          three_current_original_counters
      in
      let _, counter_final =
        heapify heap 0 initialized_counter
      in
      ( ( array,
          length_minus_one,
          map_vertices_to_indices ),
        set_first_counter_to_zero counter_final ))

let rec decrease_key_helper heap index
    three_current_original_counters =
  let new_counter =
    decrement_second_counter
      three_current_original_counters
  in
  let result, counter_final =
    let _ = Raml.tick 1.0 in
    let array, _, map_vertices_to_indices = heap in
    if index = 0 then ((), new_counter)
    else
      let element_index =
        Rarray.get array (Rnat.of_int index)
      in
      let parent_index = (index - 1) / 2 in
      let parent_element =
        Rarray.get array (Rnat.of_int parent_index)
      in
      if
        is_shorter_distance
          (distance_pair element_index)
          (distance_pair parent_element)
      then
        let (Vertex_dist (v, _)) = element_index in
        let (Vertex_dist (v_parent, _)) =
          parent_element
        in
        let _ =
          Rarray.set array
            (Rnat.of_int index)
            parent_element
        in
        let _ =
          Rarray.set array
            (Rnat.of_int parent_index)
            element_index
        in
        let _ =
          Rarray.set map_vertices_to_indices
            (Rnat.of_int v) parent_index
        in
        let _ =
          Rarray.set map_vertices_to_indices
            (Rnat.of_int v_parent)
            index
        in
        decrease_key_helper heap parent_index
          new_counter
      else ((), new_counter)
  in
  (result, increment_second_counter counter_final)

let decrease_key heap vertex new_dist
    three_current_original_counters =
  let _ = Raml.tick 1.0 in
  let array, _, map_vertices_to_indices = heap in
  let array_index =
    Rarray.get map_vertices_to_indices
      (Rnat.of_int vertex)
  in
  if array_index = -1 then
    ((), three_current_original_counters)
  else
    let (Vertex_dist (_, dist)) =
      Rarray.get array (Rnat.of_int array_index)
    in
    if is_shorter_distance new_dist dist then
      let _ =
        Rarray.set array
          (Rnat.of_int array_index)
          (Vertex_dist (vertex, new_dist))
      in
      let initialized_counter =
        initialize_second_counter
          three_current_original_counters
      in
      let result, counter_final =
        decrease_key_helper heap array_index
          initialized_counter
      in
      ( result,
        set_second_counter_to_zero counter_final )
    else ((), three_current_original_counters)

let rec initialize_array array index_nat =
  let _ = Raml.tick 1.0 in
  Rnat.ifz index_nat
    (fun () → ())
    (fun index_minus_one →
      let _ =
        Rnat.ifz index_minus_one
          (fun () →
            Rarray.set array index_minus_one
              (Vertex_dist
                 (Rnat.to_int index_minus_one, Some 0.)))
          (fun _ →
            Rarray.set array index_minus_one
              (Vertex_dist
                 ( Rnat.to_int index_minus_one,
                   Infinity )))
      in
      initialize_array array index_minus_one)
```

Lst. B.46: Resource-guarded code of Dijkstra (part 2).

```
let rec initialize_map_vertices_to_indices array
    index_nat =
  let _ = Raml.tick 1.0 in
  Rnat.ifz index_nat
    (fun () → ())
    (fun index_minus_one →
      let _ =
        Rarray.set array index_minus_one
          (Rnat.to_int index_minus_one)
      in
      initialize_map_vertices_to_indices array
        index_minus_one)

let construct_initial_heap adjacency_list =
  let _ = Raml.tick 1.0 in
  let num_vertices = Rarray.length adjacency_list in
  let array =
    Rarray.make num_vertices
      (Vertex_dist (-1, Infinity))
  in
  let _ = initialize_array array num_vertices in
  let map_vertices_to_indices =
    Rarray.make num_vertices (-1)
  in
  let _ =
    initialize_map_vertices_to_indices
      map_vertices_to_indices num_vertices
  in
  (array, num_vertices, map_vertices_to_indices)

let extract_neighbors adjacency_list (vertex : int) =
  let _ = Raml.tick 1.0 in
  let list_neighbors =
    Rarray.get adjacency_list (Rnat.of_int vertex)
  in
  list_neighbors

let rec update_dist_all_neighbors heap list_neighbors
    base_dist three_current_original_counters =
  let new_counter =
    decrement_third_counter
      three_current_original_counters
  in
  let result, counter_final =
    let _ = Raml.tick 1.0 in
    match list_neighbors with
    | [] → ((), new_counter)
    | (vertex, dist) :: tl →
        let _, counter1 =
          decrease_key heap vertex
            (add_distances base_dist dist)
            new_counter
        in
        update_dist_all_neighbors heap tl base_dist
          counter1
  in
  (result, increment_third_counter counter_final)

let rec repeatedly_get_min_node adjacency_list heap
    acc three_current_original_counters =
  let _ = Raml.tick 1.0 in
  let _, length, _ = heap in
  if Rnat.to_int length = 0 then acc
  else
    let min_node = get_min heap in
    let heap_updated, counter1 =
      delete_min heap three_current_original_counters
    in
    let (Vertex_dist (vertex, dist)) = min_node in
    let list_neighbors =
      extract_neighbors adjacency_list vertex
    in
    let initialized_counter =
      initialize_third_counter counter1
    in
    let _, counter2 =
      update_dist_all_neighbors heap list_neighbors
        dist initialized_counter
    in
    let counter3 =
      set_third_counter_to_zero counter2
    in
    let acc_updated = (vertex, dist) :: acc in
    repeatedly_get_min_node adjacency_list
      heap_updated acc_updated counter3

(* Polynomial degree for AARA: 3 *)

let dijkstra_algorithm adjacency_list
    three_current_original_counters =
  let _ = Raml.tick 1.0 in
  let heap = construct_initial_heap adjacency_list in
  repeatedly_get_min_node adjacency_list heap []
    three_current_original_counters
```

Lst. B.47: Resource-guarded code of Dijkstra (part 3).

B.2.13 BellmanFord

```
exception Invalid_input

type distance = Infinity | Some of float

let add_distances shortest_distance weight =
  let _ = Raml.tick 1.0 in
  match shortest_distance with
  | Infinity → Infinity
  | Some d → Some (d +. weight)

let is_shorter_distance distance1 distance2 =
  let _ = Raml.tick 1.0 in
  match (distance1, distance2) with
  | Some d1, Some d2 →d1 < d2
  | Some _, Infinity →true
  | _, _ → false

let rec list_nat_length list =
  let _ = Raml.tick 1.0 in
  match list with
  | [] → Rnat.zero
  | _ :: tl → Rnat.succ (list_nat_length tl)

let initialize_array_dist adjacency_list =
  let _ = Raml.tick 1.0 in
  let num_vertices = list_nat_length adjacency_list in
  let array_dist =
    Rarray.make num_vertices Infinity
  in
  let _ = Rarray.set array_dist Rnat.zero (Some 0.) in
  array_dist
```

Lst. B.48: Resource-decomposed code of BellmanFord (part 1).

```
let update_array_dist_single_vertex array_dist vertex
    new_dist =
  let _ = Raml.tick 1.0 in
  let old_dist =
    Rarray.get array_dist (Rnat.of_int vertex)
  in
  if is_shorter_distance new_dist old_dist then
    let _ =
      Rarray.set array_dist
        (Rnat.of_int vertex)
        new_dist
    in
    true
  else false

let rec update_array_dist_all_neighbors_helper
    array_dist shortest_dist list_neighbors =
  let _ = Raml.tick 1.0 in
  match list_neighbors with
  | [] → false
  | (v, d) :: tl →
      let new_dist = add_distances shortest_dist d in
      let is_modified1 =
        update_array_dist_single_vertex array_dist v
          new_dist
      in
      let is_modified2 =
        update_array_dist_all_neighbors_helper
          array_dist shortest_dist tl
      in
      is_modified1 || is_modified2

let update_array_dist_all_neighbors array_dist vertex
    list_neighbors =
  let _ = Raml.tick 1.0 in
  let shortest_dist =
    Rarray.get array_dist (Rnat.of_int vertex)
  in
  update_array_dist_all_neighbors_helper array_dist
    shortest_dist list_neighbors

let rec update_array_dist_all_edges array_dist
    adjacency_list =
  let _ = Raml.tick 1.0 in
  match adjacency_list with
  | [] → false
  | (v, list_neighbors) :: tl →
      let is_modified1 =
        update_array_dist_all_neighbors array_dist v
          list_neighbors
      in
      let is_modified2 =
        update_array_dist_all_edges array_dist tl
      in
      is_modified1 || is_modified2

let rec recursively_update_array_dist array_dist
    adjacency_list budget =
  let _ = Raml.mark 0 1.0 in
  let _ = Raml.tick 1.0 in
  let result =
    let is_modified =
      update_array_dist_all_edges array_dist
        adjacency_list
    in
    if is_modified && budget > 1 then
      recursively_update_array_dist array_dist
        adjacency_list (budget - 1)
    else ()
  in
  let _ = Raml.mark 0 (-1.0) in
  result

let rec extract_output_from_array_dist array_dist
    index =
  let _ = Raml.tick 1.0 in
  Rnat.ifz index
    (fun () → [])
    (fun index_minus_one →
      let recursive_result =
        extract_output_from_array_dist array_dist
          index_minus_one
      in
      let dist =
        Rarray.get array_dist index_minus_one
      in
      (Rnat.to_int index_minus_one, dist)
      :: recursive_result)

let bellman_ford_algorithm adjacency_list =
  let _ = Raml.tick 1.0 in
  let array_dist =
    initialize_array_dist adjacency_list
  in
  let _ = Raml.activate_counter_variable 0 in
  let _ =
    recursively_update_array_dist array_dist
      adjacency_list
      (Rnat.to_int (Rarray.length array_dist) - 1)
  in
  let _ = Raml.record_counter_variable 0 in
  extract_output_from_array_dist array_dist
    (Rarray.length array_dist)
```

Lst. B.49: Resource-decomposed code of BellmanFord (part 2).

```
type distance = Infinity | Some of float

let add_distances shortest_distance weight =
  let _ = Raml.tick 1.0 in
  match shortest_distance with
  | Infinity → Infinity
  | Some d → Some (d +. weight)

let is_shorter_distance distance1 distance2 =
  let _ = Raml.tick 1.0 in
  match (distance1, distance2) with
  | Some d1, Some d2 →d1 < d2
  | Some _, Infinity →true
  | _, _ → false

let rec list_nat_length list =
  let _ = Raml.tick 1.0 in
  match list with
  | [] → Rnat.zero
  | _ :: tl → Rnat.succ (list_nat_length tl)

let initialize_array_dist adjacency_list =
  let _ = Raml.tick 1.0 in
  let num_vertices = list_nat_length adjacency_list in
  let array_dist =
    Rarray.make num_vertices Infinity
  in
  let _ = Rarray.set array_dist Rnat.zero (Some 0.) in
  array_dist

let update_array_dist_single_vertex array_dist
    (vertex : int) new_dist =
  let _ = Raml.tick 1.0 in
  let old_dist =
    Rarray.get array_dist (Rnat.of_int vertex)
  in
  if is_shorter_distance new_dist old_dist then
    let _ =
      Rarray.set array_dist
        (Rnat.of_int vertex)
        new_dist
    in
    true
  else false

let rec update_array_dist_all_neighbors_helper
    array_dist shortest_dist list_neighbors =
  let _ = Raml.tick 1.0 in
  match list_neighbors with
  | [] → false
  | (v, d) :: tl →
      let new_dist = add_distances shortest_dist d in
      let is_modified1 =
        update_array_dist_single_vertex array_dist v
          new_dist
      in
      let is_modified2 =
        update_array_dist_all_neighbors_helper
          array_dist shortest_dist tl
      in
      is_modified1 || is_modified2

let update_array_dist_all_neighbors array_dist vertex
    list_neighbors =
  let _ = Raml.tick 1.0 in
  let shortest_dist =
    Rarray.get array_dist (Rnat.of_int vertex)
  in
  update_array_dist_all_neighbors_helper array_dist
    shortest_dist list_neighbors

let rec update_array_dist_all_edges array_dist
    adjacency_list =
  let _ = Raml.tick 1.0 in
  match adjacency_list with
  | [] → false
  | (v, list_neighbors) :: tl →
      let is_modified1 =
        update_array_dist_all_neighbors array_dist v
          list_neighbors
      in
      let is_modified2 =
        update_array_dist_all_edges array_dist tl
      in
      is_modified1 || is_modified2

let rec recursively_update_array_dist array_dist
    adjacency_list budget current_original_counters =
  let new_counter =
    decrement_counter current_original_counters
  in
  let result, counter_final =
    let _ = Raml.tick 1.0 in
    let is_modified =
      update_array_dist_all_edges array_dist
        adjacency_list
    in
    if is_modified && budget > 1 then
      recursively_update_array_dist array_dist
        adjacency_list (budget - 1) new_counter
    else ((), new_counter)
  in
  (result, increment_counter counter_final)

let rec extract_output_from_array_dist array_dist
    index =
  let _ = Raml.tick 1.0 in
  Rnat.ifz index
    (fun () → [])
    (fun index_minus_one →
      let recursive_result =
        extract_output_from_array_dist array_dist
          index_minus_one
      in
      let dist =
        Rarray.get array_dist index_minus_one
      in
      (Rnat.to_int index_minus_one, dist)
      :: recursive_result)

(* Polynomial degree for AARA: 3 *)

let bellman_ford_algorithm adjacency_list
    current_original_counters =
  let _ = Raml.tick 1.0 in
  let array_dist =
    initialize_array_dist adjacency_list
  in
  let initialized_counter =
    initialize_counter current_original_counters
  in
  let _, counter1 =
    recursively_update_array_dist array_dist
      adjacency_list
      (Rnat.to_int (Rarray.length array_dist) - 1)
      initialized_counter
  in
  let counter2 = set_counter_to_zero counter1 in
  ( extract_output_from_array_dist array_dist
      (Rarray.length array_dist),
    counter2 )
```

Lst. B.50: Resource-guarded code of BellmanFord.

B.2.14 Kruskal

```
exception Invalid_input

type rank = Rank of int

type vertex = Vertex of int

type elem = Link of vertex | Root of rank * vertex

let make (v : int) : elem =
  let _ = Raml.mark 0 1.0 in
  let _ = Raml.tick 1.0 in
  Root (Rank 0, Vertex v)

let rec find (x : vertex)
    (array_vertices : elem Rarray.t) =
  let _ = Raml.mark 0 1.0 in
  let _ = Raml.tick 1.0 in
  let (Vertex v_int) = x in
  let v_element =
    Rarray.get array_vertices (Rnat.of_int v_int)
  in
  match v_element with
  | Root (_, v) →v
  | Link (Vertex v_parent_int) →
      let rep =
        find (Vertex v_parent_int) array_vertices
      in
      let _ =
        Rarray.set array_vertices
          (Rnat.of_int v_int)
          (Link rep)
      in
      rep

let eq (x : vertex) (y : vertex)
    (array_vertices : elem Rarray.t) : bool =
  let _ = Raml.mark 0 1.0 in
  let _ = Raml.tick 1.0 in
  let (Vertex x_rep_int) = find x array_vertices in
  let (Vertex y_rep_int) = find y array_vertices in
  x_rep_int = y_rep_int

let link (x : vertex) (y : vertex)
    (array_vertices : elem Rarray.t) : vertex =
  let _ = Raml.mark 0 1.0 in
  let _ = Raml.tick 1.0 in
  let (Vertex x_int) = x in
  let (Vertex y_int) = y in
  if x_int = y_int then x
  else
    let x_element =
      Rarray.get array_vertices (Rnat.of_int x_int)
    in
    let y_element =
      Rarray.get array_vertices (Rnat.of_int y_int)
    in
    match (x_element, y_element) with
    | Root (Rank rx, vx), Root (Rank ry, _) →
        if rx < ry then
          let _ =
            Rarray.set array_vertices
              (Rnat.of_int x_int)
              (Link y)
          in
          y
        else if rx > ry then
          let _ =
            Rarray.set array_vertices
              (Rnat.of_int y_int)
              (Link x)
          in
          x
        else
          let _ =
            Rarray.set array_vertices
              (Rnat.of_int y_int)
              (Link x)
          in
          let _ =
            Rarray.set array_vertices
              (Rnat.of_int x_int)
              (Root (Rank (rx + 1), vx))
          in
          x
    | _, _ → raise Invalid_input

let union (x : vertex) (y : vertex)
    (array_vertices : elem Rarray.t) : vertex =
  let _ = Raml.mark 0 1.0 in
  let _ = Raml.tick 1.0 in
  let x_rep = find x array_vertices in
  let y_rep = find y array_vertices in
  link x_rep y_rep array_vertices

let rec append_list_edges vertex xs ys =
  let _ = Raml.tick 1.0 in
  match xs with
  | [] → ys
  | (neighbor, weight) :: tl →
      (vertex, neighbor, weight)
      :: append_list_edges vertex tl ys
```

Lst. B.51: Resource-decomposed code of Kruskal (part 1).

```
let rec concat_list_edges adjacency_list =
  let _ = Raml.tick 1.0 in
  match adjacency_list with
  | [] → []
  | (vertex, hg_neighbor_list) :: tl →
      let tl_list_edges = concat_list_edges tl in
      append_list_edges vertex hg_neighbor_list
        tl_list_edges

let rec split xs =
  let _ = Raml.tick 1.0 in
  match xs with
  | [] → ([], [])
  | [ x ] → ([ x ], [])
  | x1 :: x2 :: tl →
      let lower, upper = split tl in
      (x1 :: lower, x2 :: upper)

let rec merge xs ys =
  let _ = Raml.tick 1.0 in
  match xs with
  | [] → ys
  | (xv1, xv2, xw) :: xs_tl →(
      match ys with
      | [] → ys
      | (yv1, yv2, yw) :: ys_tl →
          if (xw : float) <= (yw : float) then
            (xv1, xv2, xw) :: merge xs_tl ys
          else (yv1, yv2, yw) :: merge xs ys_tl )

let rec merge_sort list_edges =
  let _ = Raml.mark 1 1.0 in
  let _ = Raml.tick 1.0 in
  let result =
    match list_edges with
    | [] → []
    | [ x ] → [ x ]
    | _ →
        let lower, upper = split list_edges in
        let lower_sorted = merge_sort lower in
        let upper_sorted = merge_sort upper in
        merge lower_sorted upper_sorted
  in
  let _ = Raml.mark 1 (-1.0) in
  result

let rec list_length xs =
  let _ = Raml.tick 1.0 in
  match xs with
  | [] → 0
  | _ :: tl → 1 + list_length tl
```

```
let rec initialize_union_find_help adjacency_list
    array_vertices =
  let _ = Raml.tick 1.0 in
  match adjacency_list with
  | [] → ()
  | (v, _) :: tl →
      let v_elem = make v in
      let _ =
        Rarray.set array_vertices (Rnat.of_int v)
          v_elem
      in
      initialize_union_find_help tl array_vertices

let initialize_union_find adjacency_list =
  let _ = Raml.tick 1.0 in
  let num_vertices = list_length adjacency_list in
  let array_vertices =
    Rarray.make
      (Rnat.of_int num_vertices)
      (Link (Vertex 0))
  in
  let _ =
    initialize_union_find_help adjacency_list
      array_vertices
  in
  array_vertices

let rec traverse_sorted_list_edges list_edges
    array_vertices acc =
  let _ = Raml.tick 1.0 in
  match list_edges with
  | [] → acc
  | (v1, v2, w) :: tl →
      if eq (Vertex v1) (Vertex v2) array_vertices
      then
        traverse_sorted_list_edges tl array_vertices
          acc
      else
        let _ =
          union (Vertex v1) (Vertex v2) array_vertices
        in
        traverse_sorted_list_edges tl array_vertices
          ((v1, v2, w) :: acc)

let kruskal_algorithm adjacency_list =
  let _ = Raml.activate_counter_variable 0 in
  let _ = Raml.tick 1.0 in
  let list_edges = concat_list_edges adjacency_list in
  let _ = Raml.activate_counter_variable 1 in
  let sorted_list_edges = merge_sort list_edges in
  let _ = Raml.record_counter_variable 1 in
  let array_vertices =
    initialize_union_find adjacency_list
  in
  let selected_edges =
    traverse_sorted_list_edges sorted_list_edges
      array_vertices []
  in
  let _ = Raml.record_counter_variable 0 in
  selected_edges
```

Lst. B.52: Resource-decomposed code of Kruskal (part 2).

```
type rank = Rank of int

type vertex = Vertex of int

type elem = Link of vertex | Root of rank * vertex

let make (v : int) current_original_counters =
  let new_counter =
    decrement_first_counter current_original_counters
  in
  let _ = Raml.tick 1.0 in
  let result = Root (Rank 0, Vertex v) in
  (result, new_counter)

let rec find (x : vertex)
    (array_vertices : elem Rarray.t)
    current_original_counters =
  let new_counter =
    decrement_first_counter current_original_counters
  in
  let _ = Raml.tick 1.0 in
  let (Vertex v_int) = x in
  let v_element =
    Rarray.get array_vertices (Rnat.of_int v_int)
  in
  match v_element with
  | Root (_, v) →(v, new_counter)
  | Link (Vertex v_parent_int) →
      let rep, final_counter =
        find (Vertex v_parent_int) array_vertices
          new_counter
      in
      let _ =
        Rarray.set array_vertices
          (Rnat.of_int v_int)
          (Link rep)
      in
      (rep, final_counter)

let eq (x : vertex) (y : vertex)
    (array_vertices : elem Rarray.t)
    current_original_counters =
  let new_counter =
    decrement_first_counter current_original_counters
  in
  let _ = Raml.tick 1.0 in
  let Vertex x_rep_int, counter1 =
    find x array_vertices new_counter
  in
  let Vertex y_rep_int, counter2 =
    find y array_vertices counter1
  in
  let result = x_rep_int = y_rep_int in
  (result, counter2)

let link (x : vertex) (y : vertex)
    (array_vertices : elem Rarray.t)
    current_original_counters =
  let new_counter =
    decrement_first_counter current_original_counters
  in
  let _ = Raml.tick 1.0 in
  let (Vertex x_int) = x in
  let (Vertex y_int) = y in
  if x_int = y_int then (x, new_counter)
  else
    let x_element =
      Rarray.get array_vertices (Rnat.of_int x_int)
    in
    let y_element =
      Rarray.get array_vertices (Rnat.of_int y_int)
    in
    match (x_element, y_element) with
    | Root (Rank rx, vx), Root (Rank ry, _) →
        if rx < ry then
          let _ =
            Rarray.set array_vertices
              (Rnat.of_int x_int)
              (Link y)
          in
          (y, new_counter)
        else if rx > ry then
          let _ =
            Rarray.set array_vertices
              (Rnat.of_int y_int)
              (Link x)
          in
          (x, new_counter)
        else
          let _ =
            Rarray.set array_vertices
              (Rnat.of_int y_int)
              (Link x)
          in
          let _ =
            Rarray.set array_vertices
              (Rnat.of_int x_int)
              (Root (Rank (rx + 1), vx))
          in
          (x, new_counter)
    | _, _ → raise Invalid_input

let union (x : vertex) (y : vertex)
    (array_vertices : elem Rarray.t)
    current_original_counters =
  let new_counter =
    decrement_first_counter current_original_counters
  in
  let _ = Raml.tick 1.0 in
  let x_rep, counter1 =
    find x array_vertices new_counter
  in
  let y_rep, counter2 =
    find y array_vertices counter1
  in
  link x_rep y_rep array_vertices counter2

let rec append_list_edges vertex xs ys =
  let _ = Raml.tick 1.0 in
  match xs with
  | [] → ys
  | (neighbor, weight) :: tl →
      (vertex, neighbor, weight)
      :: append_list_edges vertex tl ys

let rec concat_list_edges adjacency_list =
  let _ = Raml.tick 1.0 in
  match adjacency_list with
  | [] → []
  | (vertex, hg_neighbor_list) :: tl →
      let tl_list_edges = concat_list_edges tl in
      append_list_edges vertex hg_neighbor_list
        tl_list_edges
```

Lst. B.53: Resource-guarded code of Kruskal (part 1).

```
let rec split xs =
  let _ = Raml.tick 1.0 in
  match xs with
  | [] → ([], [])
  | [ x ] → ([ x ], [])
  | x1 :: x2 :: tl →
      let lower, upper = split tl in
      (x1 :: lower, x2 :: upper)

let rec merge xs ys =
  let _ = Raml.tick 1.0 in
  match xs with
  | [] → ys
  | (xv1, xv2, xw) :: xs_tl →(
      match ys with
      | [] → ys
      | (yv1, yv2, yw) :: ys_tl →
          if (xw : float) <= (yw : float) then
            (xv1, xv2, xw) :: merge xs_tl ys
          else (yv1, yv2, yw) :: merge xs ys_tl )

let rec merge_sort list_edges
    current_original_counters =
  let new_counter =
    decrement_second_counter current_original_counters
  in
  let _ = Raml.tick 1.0 in
  let result, final_counter =
    match list_edges with
    | [] → ([], new_counter)
    | [ x ] → ([ x ], new_counter)
    | _ →
        let lower, upper = split list_edges in
        let lower_sorted, counter1 =
          merge_sort lower new_counter
        in
        let upper_sorted, counter2 =
          merge_sort upper counter1
        in
        (merge lower_sorted upper_sorted, counter2)
  in
  (result, increment_second_counter final_counter)

let rec list_length xs =
  let _ = Raml.tick 1.0 in
  match xs with
  | [] → 0
  | _ :: tl → 1 + list_length tl

let rec initialize_union_find_help adjacency_list
    array_vertices current_original_counters =
  let _ = Raml.tick 1.0 in
  match adjacency_list with
  | [] → ((), current_original_counters)
  | (v, _) :: tl →
      let v_elem, final_counter =
        make v current_original_counters
      in
      let _ =
        Rarray.set array_vertices (Rnat.of_int v)
          v_elem
      in
      initialize_union_find_help tl array_vertices
        final_counter

let initialize_union_find adjacency_list
    current_original_counters =
  let _ = Raml.tick 1.0 in
  let num_vertices = list_length adjacency_list in
  let array_vertices =
    Rarray.make
      (Rnat.of_int num_vertices)
      (Link (Vertex 0))
  in
  let _, final_counter =
    initialize_union_find_help adjacency_list
      array_vertices current_original_counters
  in
  (array_vertices, final_counter)

let rec traverse_sorted_list_edges list_edges
    array_vertices acc current_original_counters =
  let _ = Raml.tick 1.0 in
  match list_edges with
  | [] → (acc, current_original_counters)
  | (v1, v2, w) :: tl →
      let eq_result, counter1 =
        eq (Vertex v1) (Vertex v2) array_vertices
          current_original_counters
      in
      if eq_result then
        traverse_sorted_list_edges tl array_vertices
          acc counter1
      else
        let _, counter2 =
          union (Vertex v1) (Vertex v2) array_vertices
            counter1
        in
        traverse_sorted_list_edges tl array_vertices
          ((v1, v2, w) :: acc)
          counter2

(* Polynomial degree for AARA: 3 *)

let kruskal_algorithm adjacency_list
    current_original_counters =
  let initialized_counter1 =
    initialize_first_counter current_original_counters
  in
  let initialized_counter2 =
    initialize_second_counter initialized_counter1
  in
  let _ = Raml.tick 1.0 in
  let list_edges = concat_list_edges adjacency_list in
  let sorted_list_edges, counter1 =
    merge_sort list_edges initialized_counter2
  in
  let array_vertices, counter2 =
    initialize_union_find adjacency_list counter1
  in
  let selected_edges, _ =
    traverse_sorted_list_edges sorted_list_edges
      array_vertices [] counter2
  in
  selected_edges
```

Lst. B.54: Resource-guarded code of Kruskal (part 2).

B.2.15 QuickSortTiML

```
open Core

type 'a pList = PList of 'a list * 'a list

let convert_int_to_bit_vector x =
  (* let _ = Raml.tick 1.0 in *)
  let rec convert_int_acc x acc =
    if x = 0 then acc
    else if x = 1 then 1 :: acc
    else convert_int_acc (x / 2) ((x mod 2) :: acc)
  in
  convert_int_acc x []

(* Compare two bit vectors representing natural
   numbers. The result of comparison between x and y
   is (i) -1 if x < y; (ii) 0 if x = y; and (iii) 1
   if x > y. *)

let compare_bit_vectors_maximum_length xs ys =
  let rec compare_bits_helper xs ys acc =
    let _ = Raml.tick 1.0 in
    let _ = Raml.mark 0 1.0 in
    match (xs, ys) with
    | [], [] → acc
    | [], _ :: tl →compare_bits_helper [] tl (-1)
    | _ :: tl, [] →compare_bits_helper tl [] 1
    | hd1 :: tl1, hd2 :: tl2 →
        if hd1 = hd2 then
          compare_bits_helper tl1 tl2 acc
        else if hd1 < hd2 then
          compare_bits_helper tl1 tl2 (-1)
        else compare_bits_helper tl1 tl2 1
  in
  let xs_reversed = List.rev xs in
  let ys_reversed = List.rev ys in
  compare_bits_helper xs_reversed ys_reversed 0

let le_complex (x, y) =
  let _ = Raml.tick 1.0 in
  let x_bit_vector = convert_int_to_bit_vector x in
  let y_bit_vector = convert_int_to_bit_vector y in
  let _ = Raml.activate_counter_variable 0 in
  let comparison_result =
    compare_bit_vectors_maximum_length x_bit_vector
      y_bit_vector
  in
  let _ = Raml.record_counter_variable 0 in
  if comparison_result < 1 then true else false

let rec partition (xs, pivot) =
  let _ = Raml.tick 1.0 in
  match xs with
  | [] → PList ([], [])
  | hd :: tl →(
      match partition (tl, pivot) with
      | PList (left, right) →
          if le_complex (hd, pivot) then
            PList (hd :: left, right)
          else PList (left, hd :: right))

let rec append (xs, ys) =
  let _ = Raml.tick 1.0 in
  match xs with
  | [] → ys
  | hd :: tl →hd :: append (tl, ys)

let rec quicksort xs =
  let _ = Raml.tick 1.0 in
  match xs with
  | [] → []
  | hd :: tl →(
      match partition (tl, hd) with
      | PList (left, right) →
          let left_sorted = quicksort left in
          let right_sorted = quicksort right in
          append (left_sorted, hd :: right_sorted))
```

Lst. B.55: Resource-decomposed code of QuickSortTiML. The function `le_complex` compares two natural numbers by first converting them to bit vectors and then traversing them. The cost of calling `le_complex` is given by the number of calling this function. It disregards the cost of calling helper functions: `convert_int_to_bit_vector` and `List.rev` (provided by the `Core` library).

```
(* Basic types and function for the benchmark quicksort_timl *)

structure Basic = struct

datatype bool = true | false

(* length-indexed list *)
datatype list 'a : {Nat} =
        Nil of list 'a {0}
      | Cons {n : Nat} of 'a * list 'a {n} →list 'a {n + 1}

fun le_complex ['a] {r: Nat} (x: 'a, y: 'a, resource_guard : list unit {r})
    return bool using $r =
    builtin

end
```

Lst. B.56: Basic types and function used in the resource-guarded code of QuickSortTiML written in TiML [222].

```
(* Quick sort *)

structure QSort = struct

open Basic

datatype plist 'a : {Nat} =
        PList {p q : Nat} of list 'a {p} * list 'a {q} →plist 'a {p + q}

idx T_partition = fn m n => $n + ($m + 1.0) * $n
fun partition ['a] {r len : Nat} (xs : list 'a {len}, pivot : 'a, resource_guard: list unit {r})
    return plist 'a {len} using T_partition r len =
  case xs of
      [] => PList ([], [])
    | hd :: tl =>
          case partition (tl, pivot, resource_guard) of
              PList (left, right) =>
              if le_complex (hd, pivot, resource_guard) then
                  PList (hd :: left, right)
              else
                  PList (left, hd :: right)

idx T_append = fn n => 1.0 * $n
fun append ['a] {len1 len2 : Nat} (xs : list 'a {len1}, ys : list 'a {len2})
    return list 'a {len1 + len2} using T_append len1 =
  case xs of
      [] => ys
    | hd :: tl => hd :: append (tl, ys)

idx T_quicksort = fn m n => $n * ($n + 1.0) * ($m + 2.0) + 2.0 * $n
fun quicksort ['a] {r len : Nat} (xs : list 'a {len}, resource_guard: list unit {r})
    return list 'a {len} using T_quicksort r len =
  case xs of
      [] => []
    | hd :: tl =>
      (* need time annotation here to forget the two local index variables which are the lengths of
        the two partitions. It is very hard for the typechecker to figure out how to replace these
        two lengths with the total length of the input list *)
      case partition (tl, hd, resource_guard)
          return using $len * ($len + 1.0) * ($r + 2.0) + 2.0 * $len - 2.0 * $len * ($r + 1.0) of
          PList (left, right) =>
          let
              val sorted_left = quicksort (left, resource_guard)
              val sorted_right = quicksort (right, resource_guard)
          in
              append (sorted_left, hd :: sorted_right)
          end

end
```

Lst. B.57: Resource-guarded code of QuickSortTiML written in TiML [222]. By default, TiML concerns the resource metric of the number of function calls. The resource-decomposed program (Listing B.55) and this resource-guarded program are written in different programming languages: RaML, which extends OCaml, and TiML, which extends Standard ML. Nonetheless, as the two languages are syntactically similar to each other, it is easy to see that they both implement quicksort in an identical manner. Furthermore, the resource-decomposed program is annotated with the construct `Raml.tick` such that it has the same resource metric as the resource-guarded program.

Appendix C

Supplements to Generator Optimization

C.1 Full Experiment Results

Table C.1: Relative errors of program inputs' costs generated in QuickSort.

Input Size	Percentile	Relative Errors of Generated Costs		
		Random-Input Generator	Genetic Algorithm	Random Enumeration
200	5^{th}	-0.9289	-0.0392	-0.0392
	50^{th}	-0.9232	-0.0196	-0.0196
	95^{th}	-0.9124	-0.0033	-0.0033
300	5^{th}	-0.9489	-0.0424	-0.0424
	50^{th}	-0.9447	-0.0209	-0.0209
	95^{th}	-0.9386	-0.0069	-0.0069
400	5^{th}	-0.9591	-0.0390	-0.0390
	50^{th}	-0.9562	-0.0210	-0.0210
	95^{th}	-0.9511	-0.0065	-0.0065

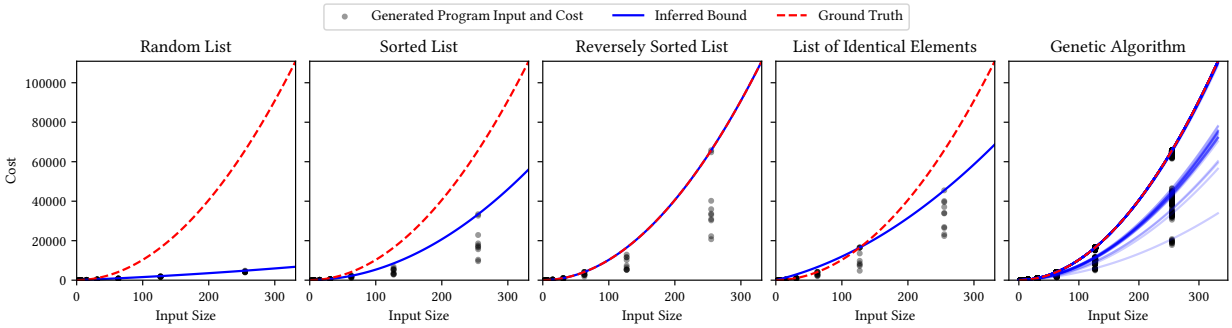


Figure C.1: Generated runtime-cost data and inferred cost bounds in QuickSort.

Table C.2: Relative errors of program inputs' costs generated in QuickSortRev.

Input Size	Percentile	Relative Errors of Generated Costs		
		Random-Input Generator	Genetic Algorithm	Random Enumeration
200	5^{th}	-0.9293	-0.0451	-0.0451
	50^{th}	-0.9222	-0.0196	-0.0196
	95^{th}	-0.9077	-0.0033	-0.0033
300	5^{th}	-0.9481	-0.0418	-0.0418
	50^{th}	-0.9444	-0.0204	-0.0204
	95^{th}	-0.9383	-0.0069	-0.0069
400	5^{th}	-0.9593	-0.0388	-0.0388
	50^{th}	-0.9561	-0.0213	-0.0213
	95^{th}	-0.9494	-0.0081	-0.0081

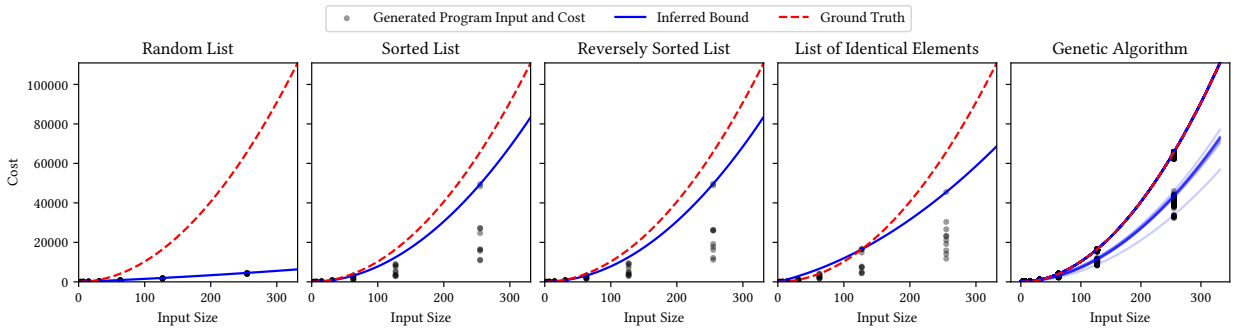


Figure C.2: Generated runtime-cost data and inferred cost bounds in QuickSortRev.

Table C.3: Relative errors of program inputs' costs generated in QuickSortStr.

Input Size	Percentile	Relative Errors of Generated Costs		
		Random-Input Generator	Genetic Algorithm	Random Enumeration
200	5^{th}	-0.9295	-0.5047	-0.5083
	50^{th}	-0.9232	-0.4950	-0.4950
	95^{th}	-0.9126	-0.4854	-0.4861
300	5^{th}	-0.9492	-0.5096	-0.5088
	50^{th}	-0.9452	-0.4999	-0.4999
	95^{th}	-0.9389	-0.4920	-0.4923
400	5^{th}	-0.9596	-0.5117	-0.5117
	50^{th}	-0.9567	-0.5024	-0.5024
	95^{th}	-0.9516	-0.4963	-0.4959

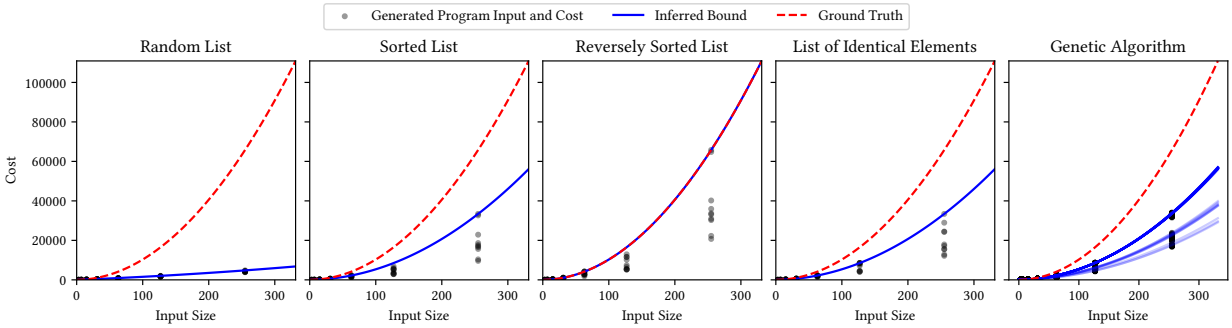


Figure C.3: Generated runtime-cost data and inferred cost bounds in QuickSortStr.

Table C.4: Relative errors of program inputs' costs generated in QuickSortRevStr.

Input Size	Percentile	Relative Errors of Generated Costs		
		Random-Input Generator	Genetic Algorithm	Random Enumeration
200	5^{th}	-0.9298	-0.5083	-0.5088
	50^{th}	-0.9227	-0.4950	-0.4950
	95^{th}	-0.9114	-0.4866	-0.4852
300	5^{th}	-0.9487	-0.5097	-0.5067
	50^{th}	-0.9447	-0.4999	-0.4997
	95^{th}	-0.9388	-0.4921	-0.4919
400	5^{th}	-0.9596	-0.5099	-0.5126
	50^{th}	-0.9565	-0.5027	-0.5017
	95^{th}	-0.9501	-0.4941	-0.4954

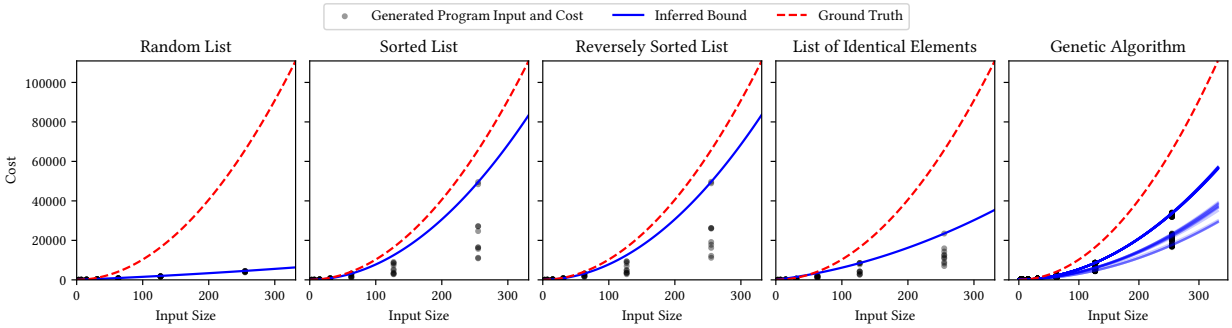


Figure C.4: Generated runtime-cost data and inferred cost bounds in QuickSortRevStr.

Table C.5: Relative errors of program inputs' costs generated in InsertionSort.

Input Size	Percentile	Relative Errors of Generated Costs		
		Random-Input Generator	Genetic Algorithm	Random Enumeration
200	5^{th}	-0.5294	-0.4828	-0.5254
	50^{th}	-0.4905	-0.4131	-0.4950
	95^{th}	-0.4572	-0.3541	-0.4564
300	5^{th}	-0.5228	-0.4866	-0.5282
	50^{th}	-0.4977	-0.4187	-0.4980
	95^{th}	-0.4625	-0.3627	-0.4629
400	5^{th}	-0.5251	-0.4929	-0.5229
	50^{th}	-0.4958	-0.4455	-0.4928
	95^{th}	-0.4666	-0.3730	-0.4748

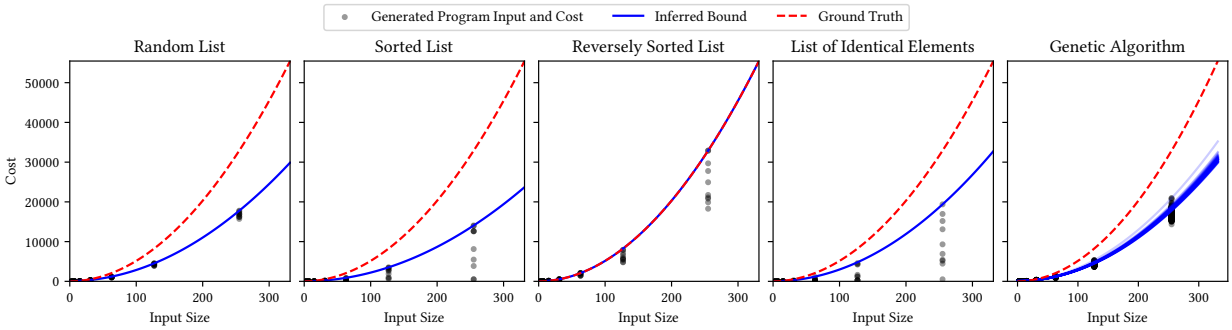


Figure C.5: Generated runtime-cost data and inferred cost bounds in InsertionSort.

Table C.6: Relative errors of program inputs' costs generated in Lpairs.

Input Size	Percentile	Relative Errors of Generated Costs		
		Random-Input Generator	Genetic Algorithm	Random Enumeration
200	5^{th}	-0.5980	-0.0452	-0.0452
	50^{th}	-0.5075	-0.0251	-0.0251
	95^{th}	-0.4271	-0.0050	-0.0050
300	5^{th}	-0.5652	-0.0368	-0.0368
	50^{th}	-0.4983	-0.0167	-0.0167
	95^{th}	-0.4378	-0.0033	-0.0033
400	5^{th}	-0.5594	-0.0326	-0.0326
	50^{th}	-0.5038	-0.0175	-0.0175
	95^{th}	-0.4586	-0.0025	-0.0025

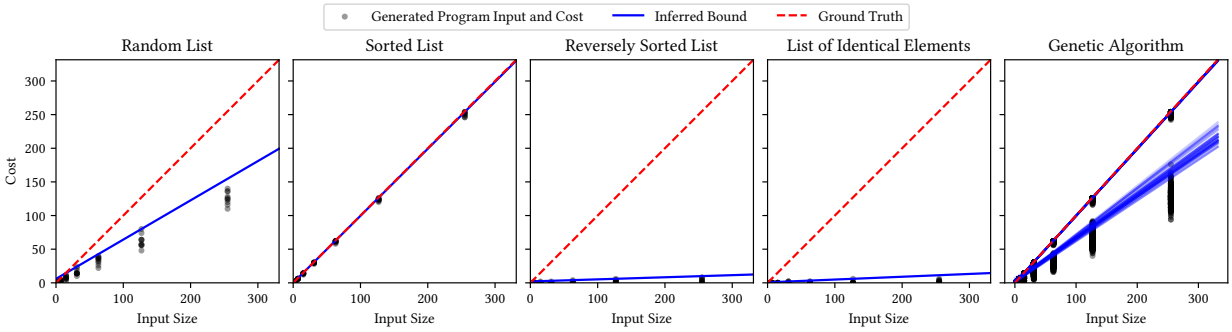


Figure C.6: Generated runtime-cost data and inferred cost bounds in Lpairs.

Table C.7: Relative errors of program inputs' costs generated in LpairsAlt.

Input Size	Percentile	Relative Errors of Generated Costs		
		Random-Input Generator	Genetic Algorithm	Random Enumeration
200	5^{th}	-0.5879	-0.0653	-0.0653
	50^{th}	-0.5176	-0.0251	-0.0251
	95^{th}	-0.4171	-0.0050	-0.0050
300	5^{th}	-0.5585	-0.0702	-0.0702
	50^{th}	-0.5050	-0.0301	-0.0301
	95^{th}	-0.4445	-0.0033	-0.0033
400	5^{th}	-0.5589	-0.0526	-0.0526
	50^{th}	-0.4987	-0.0326	-0.0326
	95^{th}	-0.4386	-0.0025	-0.0025

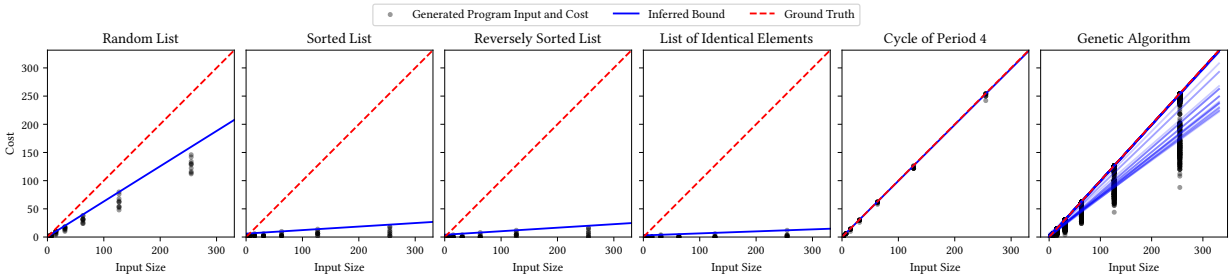


Figure C.7: Generated runtime-cost data and inferred cost bounds in LpairsAlt.

Table C.8: Relative errors of program inputs' costs generated in Opairs.

Input Size	Percentile	Relative Errors of Generated Costs		
		Random-Input Generator	Genetic Algorithm	Random Enumeration
200	5^{th}	-0.2634	-0.2319	-0.2655
	50^{th}	-0.2465	-0.1417	-0.2462
	95^{th}	-0.2268	-0.0019	-0.2286
300	5^{th}	-0.2636	-0.2495	-0.2621
	50^{th}	-0.2459	-0.1742	-0.2462
	95^{th}	-0.2332	-0.0650	-0.2337
400	5^{th}	-0.2629	-0.2702	-0.2614
	50^{th}	-0.2482	-0.1860	-0.2466
	95^{th}	-0.2335	-0.0936	-0.2346

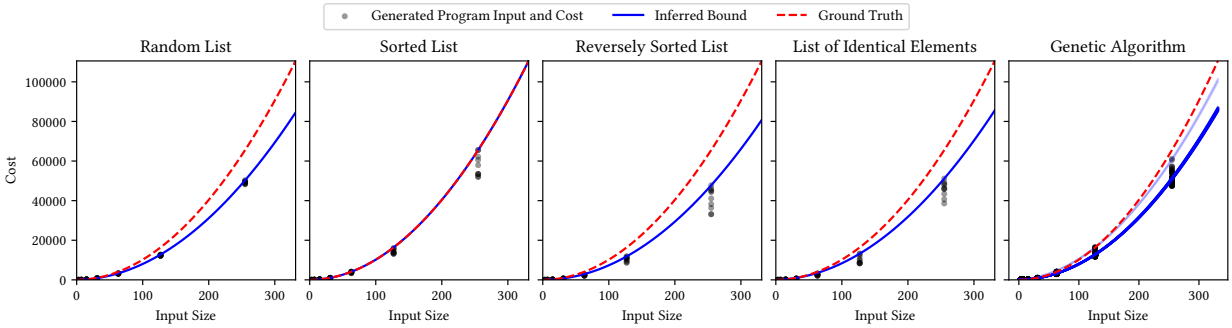


Figure C.8: Generated runtime-cost data and inferred cost bounds in Opairs.

Table C.9: Relative errors of program inputs' costs generated in QuickSelect.

Input Size	Percentile	Relative Errors of Generated Costs		
		Random-Input Generator	Genetic Algorithm	Random Enumeration
200	5^{th}	-0.9806	-0.9899	-0.9899
	50^{th}	-0.9782	-0.6782	-0.6375
	95^{th}	-0.9759	-0.0099	-0.0292
300	5^{th}	-0.9876	-0.9932	-0.9932
	50^{th}	-0.9854	-0.7053	-0.6350
	95^{th}	-0.9841	-0.1918	-0.0445
400	5^{th}	-0.9904	-0.9949	-0.9948
	50^{th}	-0.9893	-0.7364	-0.6950
	95^{th}	-0.9886	-0.2254	-0.2755

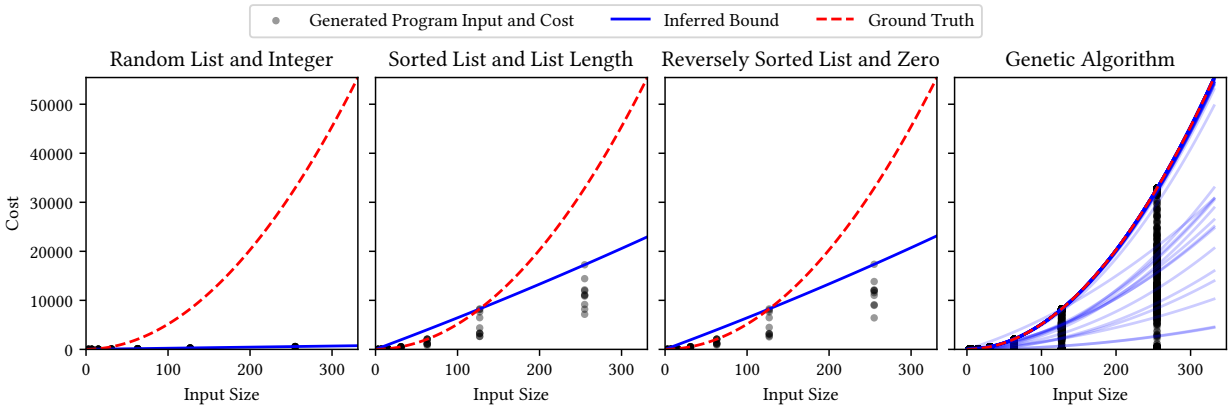


Figure C.9: Generated runtime-cost data and inferred cost bounds in QuickSelect.

Table C.10: Relative errors of program inputs' costs generated in QuickSelectStr.

Input Size	Percentile	Relative Errors of Generated Costs		
		Random-Input Generator	Genetic Algorithm	Random Enumeration
200	5^{th}	-0.9806	-0.9899	-0.8718
	50^{th}	-0.9783	-0.9895	-0.4849
	95^{th}	-0.9761	-0.0198	-0.0099
300	5^{th}	-0.9874	-0.9933	-0.9036
	50^{th}	-0.9854	-0.9931	-0.5817
	95^{th}	-0.9840	-0.2930	-0.0198
400	5^{th}	-0.9904	-0.9950	-0.9359
	50^{th}	-0.9893	-0.9947	-0.6753
	95^{th}	-0.9885	-0.1958	-0.1799

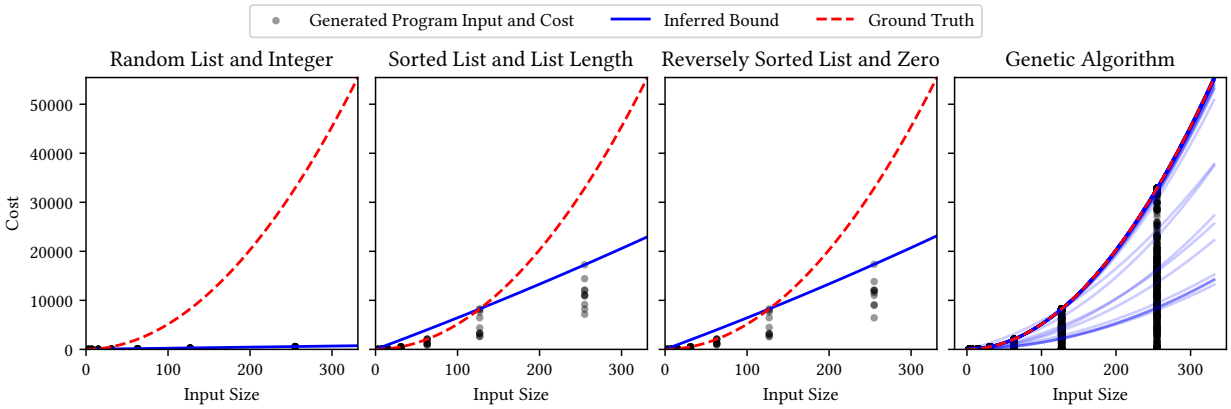


Figure C.10: Generated runtime-cost data and inferred cost bounds in QuickSelectStr.

Table C.11: Relative errors of program inputs' costs generated in LinearSearch.

Input Size	Percentile	Relative Errors of Generated Costs		
		Random-Input Generator	Genetic Algorithm	Random Enumeration
200	5^{th}	-0.5755	0.0000	0.0000
	50^{th}	0.0000	0.0000	0.0000
	95^{th}	0.0000	0.0000	0.0000
300	5^{th}	-0.7050	0.0000	0.0000
	50^{th}	0.0000	0.0000	0.0000
	95^{th}	0.0000	0.0000	0.0000
400	5^{th}	-0.7250	0.0000	0.0000
	50^{th}	0.0000	0.0000	0.0000
	95^{th}	0.0000	0.0000	0.0000

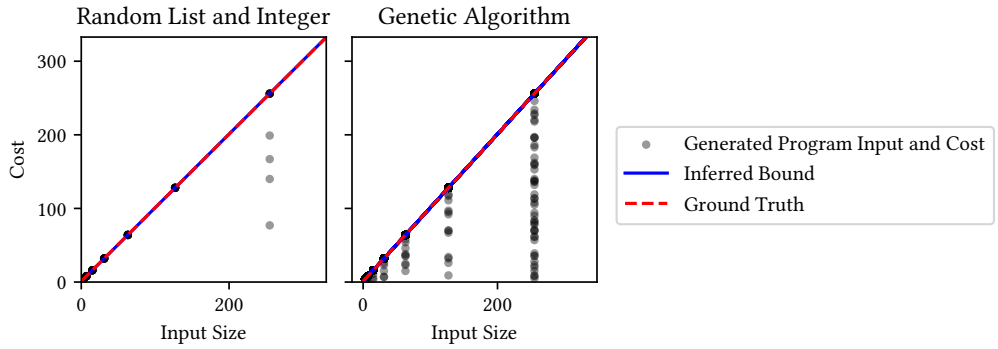


Figure C.11: Generated runtime-cost data and inferred cost bounds in LinearSearch.

Table C.12: Relative errors of program inputs' costs generated in QuickSortPairs.

Input Size	Percentile	Relative Errors of Generated Costs		
		Random-Input Generator	Genetic Algorithm	Random Enumeration
200	5^{th}	-0.9285	-0.0721	-0.0721
	50^{th}	-0.9228	-0.0418	-0.0418
	95^{th}	-0.9139	-0.0106	-0.0106
300	5^{th}	-0.9485	-0.0747	-0.0747
	50^{th}	-0.9447	-0.0418	-0.0418
	95^{th}	-0.9384	-0.0160	-0.0160
400	5^{th}	-0.9590	-0.0687	-0.0687
	50^{th}	-0.9560	-0.0471	-0.0471
	95^{th}	-0.9503	-0.0261	-0.0261

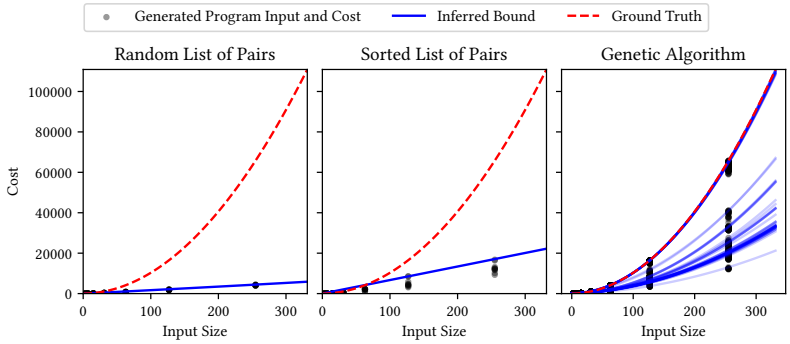


Figure C.12: Generated runtime-cost data and inferred cost bounds in QuickSortPairs.

Table C.13: Relative errors of program inputs' costs generated in QuickSortPairsStr.

Input Size	Percentile	Relative Errors of Generated Costs		
		Random-Input Generator	Genetic Algorithm	Random Enumeration
200	5^{th}	-0.9285	-0.5227	-0.5227
	50^{th}	-0.9228	-0.5030	-0.5030
	95^{th}	-0.9139	-0.4907	-0.4907
300	5^{th}	-0.9485	-0.5253	-0.5253
	50^{th}	-0.9447	-0.5072	-0.5072
	95^{th}	-0.9384	-0.4955	-0.4955
400	5^{th}	-0.9590	-0.5232	-0.5232
	50^{th}	-0.9560	-0.5121	-0.5121
	95^{th}	-0.9503	-0.5036	-0.5028

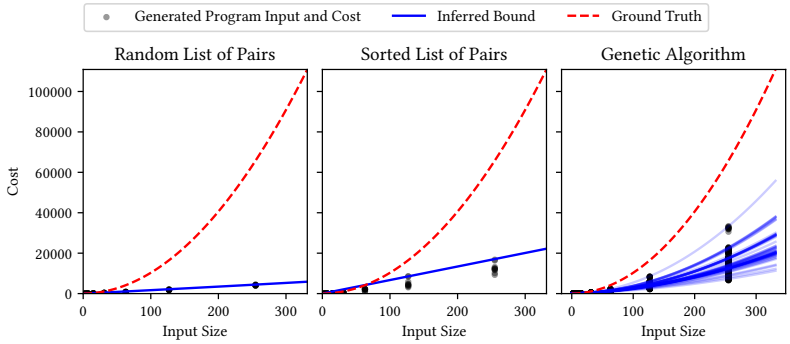


Figure C.13: Generated runtime-cost data and inferred cost bounds in QuickSortPairsStr.

Table C.14: Relative errors of program inputs' costs generated in SplitSort.

Input Size	Percentile	Relative Errors of Generated Costs		
		Random-Input Generator	Genetic Algorithm	Random Enumeration
200	5^{th}	-0.9945	-0.0879	-0.0879
	50^{th}	-0.9943	-0.0495	-0.0495
	95^{th}	-0.9939	-0.0138	-0.0138
300	5^{th}	-0.9961	-0.0826	-0.0826
	50^{th}	-0.9960	-0.0483	-0.0483
	95^{th}	-0.9958	-0.0211	-0.0211
400	5^{th}	-0.9969	-0.0756	-0.0756
	50^{th}	-0.9968	-0.0513	-0.0513
	95^{th}	-0.9966	-0.0284	-0.0284

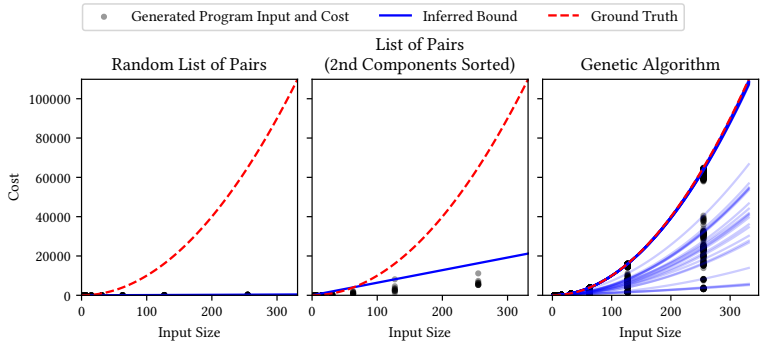


Figure C.14: Generated runtime-cost data and inferred cost bounds in SplitSort.

Table C.15: Relative errors of program inputs' costs generated in SplitSortStr.

Input Size	Percentile	Relative Errors of Generated Costs		
		Random-Input Generator	Genetic Algorithm	Random Enumeration
200	5^{th}	-0.9945	-0.5363	-0.5363
	50^{th}	-0.9943	-0.5168	-0.5168
	95^{th}	-0.9939	-0.5024	-0.5024
300	5^{th}	-0.9961	-0.5323	-0.5323
	50^{th}	-0.9960	-0.5177	-0.5177
	95^{th}	-0.9958	-0.5020	-0.5020
400	5^{th}	-0.9969	-0.5316	-0.5316
	50^{th}	-0.9968	-0.5207	-0.5207
	95^{th}	-0.9966	-0.5088	-0.5088

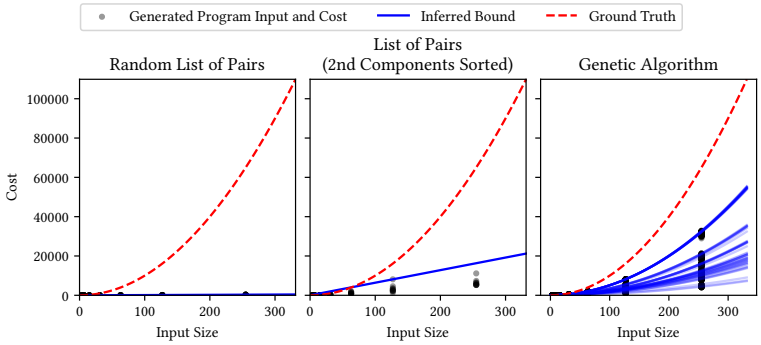


Figure C.15: Generated runtime-cost data and inferred cost bounds in SplitSortStr.

Table C.16: Relative errors of program inputs' costs generated in Queue.

Input Size	Percentile	Relative Errors of Generated Costs		
		Random-Input Generator	Genetic Algorithm	Random Enumeration
200	5^{th}	-0.1452	-0.0250	-0.0218
	50^{th}	-0.1152	-0.0134	-0.0117
	95^{th}	-0.0901	-0.0050	-0.0050
300	5^{th}	-0.1525	-0.0211	-0.0211
	50^{th}	-0.1201	-0.0133	-0.0133
	95^{th}	-0.0945	-0.0044	-0.0066
400	5^{th}	-0.1537	-0.0200	-0.0201
	50^{th}	-0.1293	-0.0125	-0.0121
	95^{th}	-0.0974	-0.0067	-0.0067

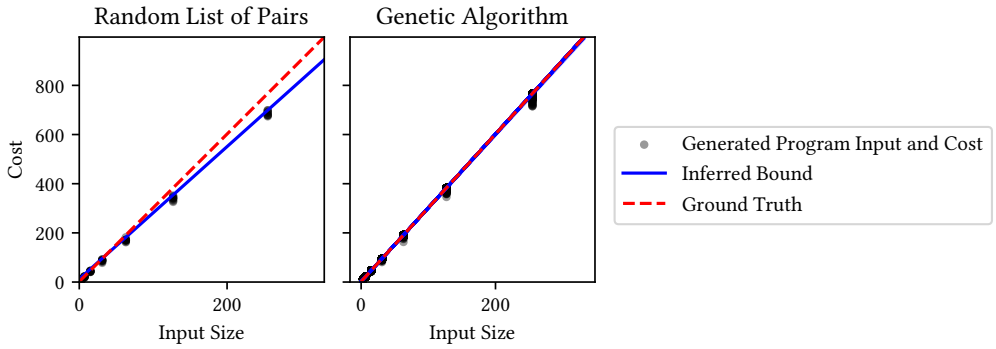


Figure C.16: Generated runtime-cost data and inferred cost bounds in Queue.

Table C.17: Relative errors of program inputs' costs generated in Compare.

Input Size	Percentile	Relative Errors of Generated Costs		
		Random-Input Generator	Genetic Algorithm	Random Enumeration
200	5^{th}	-0.9950	-0.9852	-0.9950
	50^{th}	-0.9950	-0.8850	-0.9900
	95^{th}	-0.9950	-0.3790	-0.9750
300	5^{th}	-0.9967	-0.9868	-0.9967
	50^{th}	-0.9967	-0.9267	-0.9967
	95^{th}	-0.9967	-0.7050	-0.9833
400	5^{th}	-0.9975	-0.9900	-0.9975
	50^{th}	-0.9975	-0.9450	-0.9975
	95^{th}	-0.9975	-0.7299	-0.9900

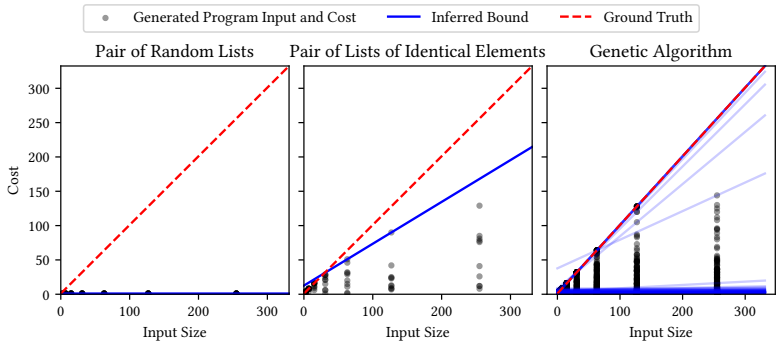


Figure C.17: Generated runtime-cost data and inferred cost bounds in Compare.

Table C.18: Relative errors of program inputs' costs generated in QuickSortLists.

Input Size	Percentile	Relative Errors of Generated Costs		
		Random-Input Generator	Genetic Algorithm	Random Enumeration
(30, 30)	5^{th}	-0.9713	-0.6491	-0.6543
	50^{th}	-0.9683	-0.4727	-0.4731
	95^{th}	-0.9639	-0.2819	-0.2944
(40, 40)	5^{th}	-0.9826	-0.7219	-0.7411
	50^{th}	-0.9804	-0.6036	-0.6022
	95^{th}	-0.9782	-0.4702	-0.4542
(50, 50)	5^{th}	-0.9878	-0.7661	-0.7759
	50^{th}	-0.9865	-0.6953	-0.6961
	95^{th}	-0.9848	-0.5693	-0.5599

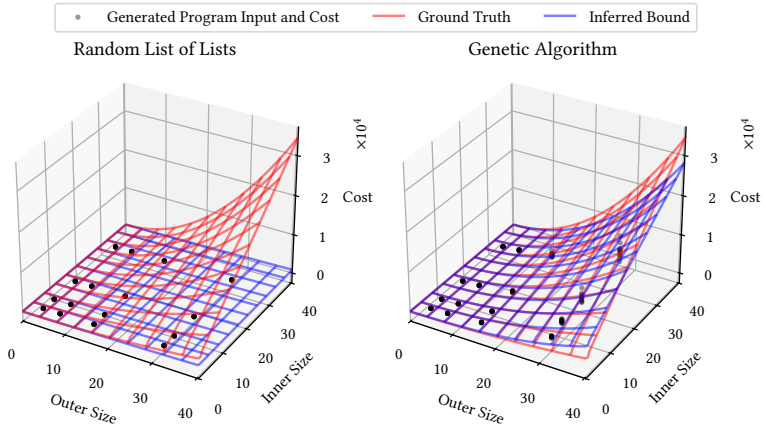


Figure C.18: Generated runtime-cost data and inferred cost bounds in QuickSortLists.

Table C.19: Relative errors of program inputs' costs generated in QuickSortListsStr.

Input Size	Percentile	Relative Errors of Generated Costs		
		Random-Input Generator	Genetic Algorithm	Random Enumeration
(30, 30)	5^{th}	-0.9713	-0.6613	-0.6630
	50^{th}	-0.9683	-0.4837	-0.5070
	95^{th}	-0.9639	-0.3072	-0.3189
(40, 40)	5^{th}	-0.9826	-0.7398	-0.7240
	50^{th}	-0.9804	-0.6033	-0.6007
	95^{th}	-0.9782	-0.4804	-0.4760
(50, 50)	5^{th}	-0.9878	-0.7864	-0.7806
	50^{th}	-0.9865	-0.6960	-0.7010
	95^{th}	-0.9848	-0.5852	-0.5512

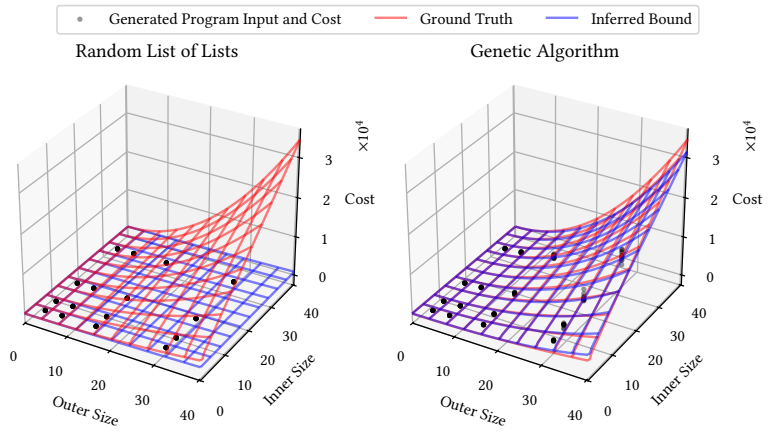


Figure C.19: Generated runtime-cost data and inferred cost bounds in QuickSortListsStr.

Table C.20: Relative errors of program inputs' costs generated in SortAll.

Input Size	Percentile	Relative Errors of Generated Costs		
		Random-Input Generator	Genetic Algorithm	Random Enumeration
(30, 30)	5^{th}	-0.6854	-0.0221	-0.0437
	50^{th}	-0.6768	-0.0152	-0.0304
	95^{th}	-0.6693	-0.0075	-0.0225
(40, 40)	5^{th}	-0.7401	-0.0231	-0.0385
	50^{th}	-0.7352	-0.0168	-0.0289
	95^{th}	-0.7288	-0.0106	-0.0217
(50, 50)	5^{th}	-0.7782	-0.0227	-0.0355
	50^{th}	-0.7746	-0.0171	-0.0286
	95^{th}	-0.7698	-0.0118	-0.0226

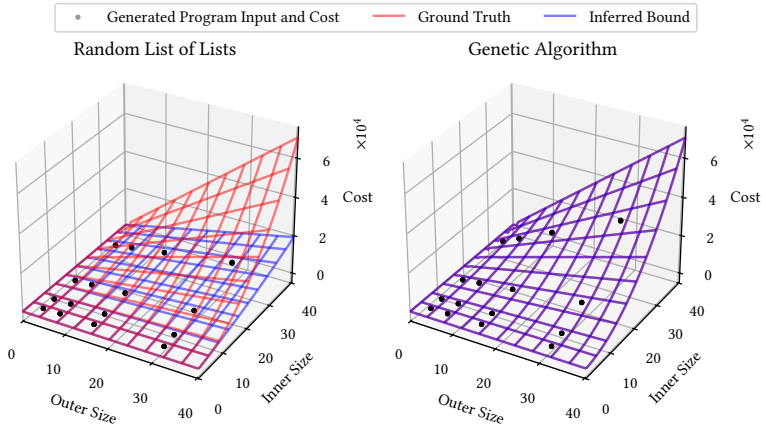


Figure C.20: Generated runtime-cost data and inferred cost bounds in SortAll.

Table C.21: Relative errors of program inputs' costs generated in SortAllStr.

Input Size	Percentile	Relative Errors of Generated Costs		
		Random-Input Generator	Genetic Algorithm	Random Enumeration
(30, 30)	5^{th}	-0.6861	-0.4490	-0.1780
	50^{th}	-0.6773	-0.4422	-0.1091
	95^{th}	-0.6697	-0.4353	-0.0652
(40, 40)	5^{th}	-0.7402	-0.4623	-0.1935
	50^{th}	-0.7356	-0.4567	-0.1513
	95^{th}	-0.7293	-0.4533	-0.1088
(50, 50)	5^{th}	-0.7786	-0.4703	-0.2486
	50^{th}	-0.7749	-0.4672	-0.1893
	95^{th}	-0.7700	-0.4629	-0.1492

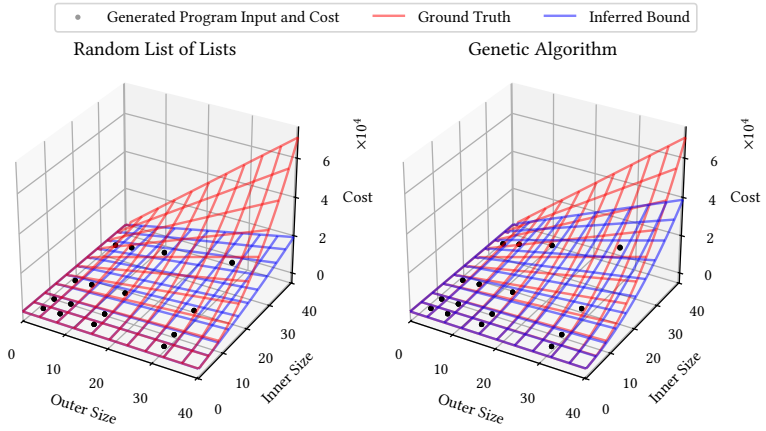


Figure C.21: Generated runtime-cost data and inferred cost bounds in SortAllStr.

Bibliography

[1] Abdallah A. Abdel Ghaly, Hanan M. Aly, and Rana N. Salah. Different estimation methods for constant stress accelerated life test under the family of the exponentiated distributions. *Quality and Reliability Engineering International*, 32(3):1095–1108, 2016. doi: doi.org/10.1002/qre.1818. 7.3.3

[2] Keith Abrams, Deborah Ashby, and Dough Errington. A bayesian approach to weibull survival models—application to a cancer clinical trial. *Lifetime Data Analysis*, pages 154–174, 1996. doi: doi.org/10.1007/BF00128573. 7.3.3

[3] Samson Abramsky and Achim Jung. *Domain theory*, pages 1–168. Oxford University Press, Inc., USA, 1995. ISBN 019853762X. 8.2.2, 8.2.2, 8.2.2, 4, 8.2.2, 8.2.2, 8.2.3

[4] Oriol Abril-Pla, Virgile Andreani, Colin Carroll, Larry Dong, Christopher J. Fonnesbeck, Maxim Kochurov, Ravin Kumar, Junpeng Lao, Christian C. Luhmann, Osvaldo A. Martin, Michael Osthege, Ricardo Vieira, Thomas Wiecki, and Robert Zinkov. Pymc: a modern, and comprehensive probabilistic programming framework in python. *PeerJ Computer Science*, 9, September 2023. doi: 10.7717/peerj-cs.1516. URL http://dx.doi.org/10.7717/peerj-cs.1516. 7.1.2

[5] Beniamino Accattoli. A Fresh Look at the λ-Calculus. In Herman Geuvers, editor, *4th International Conference on Formal Structures for Computation and Deduction (FSCD 2019)*, volume 131 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 1:1–1:20, Dagstuhl, Germany, 2019. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. ISBN 978-3-95977-107-8. doi: 10.4230/LIPIcs.FSCD.2019.1. URL https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.FSCD.2019.1. 5.1

[6] Beniamino Accattoli and Ugo Dal Lago. Beta reduction is invariant, indeed. In *Proceedings of the Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, CSL-LICS '14, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450328869. doi: 10.1145/2603088.2603105. URL https://doi.org/10.1145/2603088.2603105. 5.1

[7] Beniamino Accattoli, Andrea Condoluci, and Claudio Sacerdoti Coen. Strong call-by-value is reasonable, implosively. In *Proceedings of the 36th Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '21, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781665448956. doi: 10.1109/LICS52264.2021.9470630. URL https://doi.org/10.1109/LICS52264.2021.9470630. 5.1

[8] Beniamino Accattoli, Ugo Dal Lago, and Gabriele Vanoni. Reasonable space for the λ-calculus, logarithmically. In *Proceedings of the 37th Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '22, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450393515. doi: 10.1145/3531130.3533362. URL https://doi.org/10.1145/3531130.3533362. 5.1, 1

[9] Elvira Albert, Puri Arenas, Samir Genaim, German Puebla, and Damiano Zanardini. Cost Analysis of Java Bytecode. In *Programming Languages and Systems*, volume 4421 of *Lecture Notes in Computer Science*, pages 157–172, Berlin, 2007. Springer. ISBN 978-3-540-71316-6. doi: 10.1007/978-3-540-71316-6_12. 1.1, 5.2.1, 5.2.1, 6.3.3

[10] Elvira Albert, Puri Arenas, Samir Genaim, and Germán Puebla. Automatic inference of upper bounds for recurrence relations in cost analysis. In María Alpuente and Germán Vidal, editors, *Static Analysis*, pages 221–237, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. ISBN 978-3-540-69166-2. 5.2.1, 5.2.1

[11] Elvira Albert, Puri Arenas, Samir Genaim, German Puebla, and Damiano Zanardini. COSTA: Design and Implementation of a Cost and Termination Analyzer for Java Bytecode. In *Formal Methods for Components and Objects*, volume 5382 of *Lecture Notes in Computer Science*, pages 113–132, Berlin, 2008. Springer. ISBN 978-3-540-92188-2. doi: 10.1007/978-3-540-92188-2_5. 5.2.1, 5.2.1, 6.3.3

[12] Miltiadis Allamanis, Earl T. Barr, Soline Ducousso, and Zheng Gao. Typilus: neural type hints. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, pages 91–105, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450376136. doi: 10.1145/3385412.3385997. URL https://doi.org/10.1145/3385412.3385997. 7.7.3

[13] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. A general path-based representation for predicting program properties. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018, pages 404–419, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450356985. doi: 10.1145/3192366.3192412. URL https://doi.org/10.1145/3192366.3192412. 7.7.3

[14] Stephen Alstrup, Mikkel Thorup, Inge Li Gørtz, Theis Rauhe, and Uri Zwick. Union-find with constant time deletions. *ACM Trans. Algorithms*, 11(1), August 2014. ISSN 1549-6325. doi: 10.1145/2636922. URL https://doi.org/10.1145/2636922. 8.6

[15] Per K. Andersen and Michael Vaeth. Survival analysis. In *Wiley StatsRef: Statistics Reference Online*. John Wiley & Sons, Ltd, Hoboken, NJ, USA, 2015. doi: 10.1002/9781118445112.stat02177.pub2. 7.3.3

[16] Krzysztof R. Apt. Ten years of hoare's logic: A survey—part i. *ACM Trans. Program. Lang. Syst.*, 3(4):431–483, October 1981. ISSN 0164-0925. doi: 10.1145/357146.357150. URL https://doi.org/10.1145/357146.357150. 6.3.3

[17] Krzysztof R. Apt and Ernst-Rüdiger Olderog. Fifty years of hoare's logic. *Form. Asp. Comput.*, 31(6):751–807, December 2019. ISSN 0934-5043. doi: 10.1007/s00165-019-00501-3. URL https://doi.org/10.1007/s00165-019-00501-3. 6.3.3

[18] Andrea Asperti and Luca Roversi. Intuitionistic light affine logic. *ACM Trans. Comput. Logic*, 3(1):137–175, January 2002. ISSN 1529-3785. doi: 10.1145/504077.504081. URL https://doi.org/10.1145/504077.504081. 5.1

[19] Robert Atkey. Amortised resource analysis with separation logic. In Andrew D. Gordon, editor, *Programming Languages and Systems*, pages 85–103, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg. ISBN 978-3-642-11957-6. 1.1, 1.2.2, 1.2.3, 5.2.3

[20] Robert Atkey. Polynomial time and dependent types. *Proc. ACM Program. Lang.*, 8(POPL): 2288–2317, 2024. doi: 10.1145/3632918. URL https://doi.org/10.1145/3632918. 5.1

[21] Martin Avanzini and Ugo Dal Lago. Automating sized-type inference for complexity analysis. *Proc. ACM Program. Lang.*, 1(ICFP), August 2017. doi: 10.1145/3110287. 5.2.3, 10.2

[22] Martin Avanzini and Georg Moser. A combination framework for complexity. *Information and Computation*, 248:22–55, 2016. doi: 10.1016/j.ic.2015.12.007. 1.1, 5.2.1

[23] Martin Avanzini, Ugo Dal Lago, and Georg Moser. Analysing the complexity of functional programs: Higher-order meets first-order. In *Proc. 20th ACM SIGPLAN International Conference on Functional Programming*, pages 152–164, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3669-7. doi: 10.1145/2784731.2784753. 1.1, 5.2.1

[24] Martin Avanzini, Georg Moser, and Michael Schaper. Tct: Tyrolean complexity tool. In Marsha Chechik and Jean-François Raskin, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 407–423, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg. ISBN 978-3-662-49674-9. 5.2.1

[25] Jeremy Avigad. *Mathematical Logic and Computation*. Cambridge University Press, 2022. 6.3.3, 6.3.3

[26] Patrick Baillot and Virgile Mogbil. Soft lambda-calculus: A language for polynomial time computation. In Igor Walukiewicz, editor, *Foundations of Software Science and Computation Structures*, pages 27–41, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg. ISBN 978-3-540-24727-2. 5.1

[27] Gilles Barthe, Sandrine Blazy, Benjamin Grégoire, Rémi Hutin, Vincent Laporte, David Pichardie, and Alix Trieu. Formal verification of a constant-time preserving C compiler. *Proceedings of the ACM on Programming Languages*, 4(POPL):7:1–7:30, December 2019. doi: 10.1145/3371075. 1.1

[28] Stephen Bellantoni and Stephen Cook. A new recursion-theoretic characterization of the polytime functions (extended abstract). In *Proceedings of the Twenty-Fourth Annual ACM Symposium on Theory of Computing*, STOC '92, pages 283–293, New York, NY, USA, 1992. Association for Computing Machinery. ISBN 0897915119. doi: 10.1145/129712.129740. URL https://doi.org/10.1145/129712.129740. 5.1

[29] Yves Bertot and Castéran Pierre. *Interactive Theorem Proving and Program Development*. Springer, Berlin, 2004. doi: 10.1007/978-3-662-07964-5. 2.2, 5.2.3, 8.6

[30] Luca Beurer-Kellner, Marc Fischer, and Martin Vechev. Prompting is programming: A query language for large language models. *Proc. ACM Program. Lang.*, 7(PLDI), June 2023.

doi: 10.1145/3591300. URL https://doi.org/10.1145/3591300. 10.2

[31] Luca Beurer-Kellner, Marc Fischer, and Martin Vechev. Guiding llms the right way: fast, non-invasive constrained generation. In *Proceedings of the 41st International Conference on Machine Learning*, ICML'24. JMLR.org, 2024. 10.2

[32] Malgorzata Biernacka, Witold Charatonik, and Tomasz Drab. A derived reasonable abstract machine for strong call by value. In *Proceedings of the 23rd International Symposium on Principles and Practice of Declarative Programming*, PPDP '21, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450386890. doi: 10.1145/3479394.3479401. URL https://doi.org/10.1145/3479394.3479401. 5.1

[33] Eli Bingham, Jonathan P. Chen, Martin Jankowiak, Fritz Obermeyer, Neeraj Pradhan, Theofanis Karaletsos, Rohit Singh, Paul Szerlip, Paul Horsfall, and Noah D. Goodman. Pyro: deep universal probabilistic programming. *J. Mach. Learn. Res.*, 20(1):973–978, January 2019. ISSN 1532-4435. 7.1.2

[34] Guy Blelloch and John Greiner. Parallelism in sequential functional languages. In *Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture*, FPCA '95, pages 226–237, New York, NY, USA, 1995. Association for Computing Machinery. ISBN 0897917197. doi: 10.1145/224164.224210. URL https://doi.org/10.1145/224164.224210. 5.1

[35] Guy E. Blelloch and John Greiner. A provable time and space efficient implementation of nesl. In *Proceedings of the First ACM SIGPLAN International Conference on Functional Programming*, ICFP '96, pages 213–225, New York, NY, USA, 1996. Association for Computing Machinery. ISBN 0897917707. doi: 10.1145/232627.232650. URL https://doi.org/10.1145/232627.232650. 5.1

[36] Manuel Blum, Robert W. Floyd, Vaughan Pratt, Ronald L. Rivest, and Robert E. Tarjan. Time bounds for selection. *Journal of Computer and System Sciences*, 7(4):448–461, August 1973. ISSN 0022-0000. doi: 10.1016/S0022-0000(73)80033-9. 2.1, 2.1, 7.6.1

[37] Jason Breck, John Cyphert, Zachary Kincaid, and Thomas Reps. Templates and recurrences: better together. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, pages 688–702, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450376136. doi: 10.1145/3385412.3386035. URL https://doi.org/10.1145/3385412.3386035. 5.2.1

[38] Marc Brockschmidt, Fabian Emmes, Stephan Falke, Carsten Fuhs, and Jürgen Giesl. Alternating runtime and size complexity analysis of integer programs. In *Proc. 20th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2014)*, volume 8413 of *Lecture Notes in Computer Science*, pages 140–155, Berlin, 2014. Springer. doi: 10.1007/978-3-642-54862-8_10. 1.1, 5.2.1

[39] Marc Brockschmidt, Fabian Emmes, Stephan Falke, Carsten Fuhs, and Jürgen Giesl. Analyzing runtime and size complexity of integer programs. *ACM Trans. Program. Lang. Syst.*, 38(4), aug 2016. ISSN 0164-0925. doi: 10.1145/2866575. URL https://doi.org/10.1145/2866575. 5.2.1

[40] Steve Brooks, Andrew Gelman, Galin Jones, and Xiao-Li Meng. *Handbook of Markov*

Chain Monte Carlo. Chapman and Hall/CRC, May 2011. ISBN 9780429138508. doi: 10.1201/b10905. URL http://dx.doi.org/10.1201/b10905. 2.1, 7.1.2, 7.5.3

[41] Sébastien Bubeck. Convex optimization: Algorithms and complexity. *Found. Trends Mach. Learn.*, 8(3–4):231–357, November 2015. ISSN 1935-8237. doi: 10.1561/2200000050. URL https://doi.org/10.1561/2200000050. 7.5.2, 7.5.3

[42] Jacob Burnim, Sudeep Juvekar, and Koushik Sen. Wise: Automated test generation for worst-case complexity. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 463–473, USA, 2009. IEEE Computer Society. ISBN 9781424434534. doi: 10.1109/ICSE.2009.5070545. URL https://doi.org/10.1109/ICSE.2009.5070545. 1.1, 5.3.1, 9.1

[43] Quentin Carbonneaux, Jan Hoffmann, and Zhong Shao. Compositional certified resource bounds. In *Proc. 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 467–478, New York, NY, USA, June 2015. ACM. ISBN 978-1-4503-3468-6. doi: 10.1145/2737924.2737955. 5.2.1

[44] Bob Carpenter, Andrew Gelman, Matthew D. Hoffman, Daniel Lee, Ben Goodrich, Michael Betancourt, Marcus Brubaker, Jiqiang Guo, Peter Li, and Allen Riddell. Stan: A probabilistic programming language. *Journal of Statistical Software*, 76(1):1–32, 2017. doi: 10.18637/jss.v076.i01. URL https://www.jstatsoft.org/index.php/jss/article/view/v076i01. 1, 7.1.2, 7.1.2, 7.3.2, 7.5.3, 7.5.3, 7.6.2, 7.7.3, 8.5.2, 8.5.3

[45] George Casella. An introduction to empirical bayes data analysis. *The American Statistician*, 39(2):83–87, 1985. doi: 10.1080/00031305.1985.10479400. 7.3.1, 7.5.2

[46] Sunjay Cauligi, Gary Soeller, Brian Johannesmeyer, Fraser Brown, Riad S. Wahby, John Renner, Benjamin Grégoire, Gilles Barthe, Ranjit Jhala, and Deian Stefan. FaCT: A DSL for timing-sensitive computation. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, pages 174–189, New York, NY, USA, June 2019. Association for Computing Machinery. ISBN 978-1-4503-6712-7. doi: 10.1145/3314221.3314605. 1.1

[47] Frederic Cazals, Augustin Chevallier, and Sylvain Pion. Improved polytope volume calculations based on Hamiltonian Monte Carlo with boundary reflections and sweet arithmetics. *Journal of Computational Geometry*, 13(1):52–88, April 2022. ISSN 1920-180X. doi: 10.20382/jocg.v13i1a3. 1.2.2, 7.3.2, 7.5.3, 7.7.2, 8.8.3, 10.1, 10.2

[48] Apostolos Chalkis and Vissarion Fisikopoulos. Volesti: Volume Approximation and Sampling for Convex Polytopes in R. *The R Journal*, 13(2):642–660, 2021. ISSN 2073-4859. doi: 10.32614/RJ-2021-077. 7.3.2, 7.5.3, 7.6.2, 7.7.2

[49] Apostolos Chalkis, Vissarion Fisikopoulos, Marios Papachristou, and Elias P. Tsigaridas. Truncated log-concave sampling with reflective hamiltonian monte carlo. *CoRR*, abs/2102.13068, 2021. URL https://arxiv.org/abs/2102.13068. 1.2.2, 7.3.2, 7.5.3, 7.5.3, 7.5.4, 7.5.1, 7.7.2, 8.8.3, 10.1, 10.2

[50] Apostolos Chalkis, Vissarion Fisikopoulos, Marios Papachristou, and Elias Tsigaridas. Truncated Log-concave Sampling for Convex Bodies with Reflective Hamiltonian Monte Carlo. *ACM Transactions on Mathematical Software*, 49(2), June 2023. ISSN 0098-3500.

doi: 10.1145/3589505. 1.2.2, 2.1, 7.3.2, 7.5.3, 7.7.2, 8.8.3, 10.1, 10.2

[51] Pierre Chambon, Baptiste Roziere, Benoit Sagot, and Gabriel Synnaeve. Bigo(bench) – can llms generate code with controlled time and space complexity?, 2025. URL https://arxiv.org/abs/2503.15242. 5.2.2, 8.8.3

[52] Arthur Charguéraud and François Pottier. Verifying the Correctness and Amortized Complexity of a Union-Find Implementation in Separation Logic with Time Credits. *Journal of Automated Reasoning*, 62(3):331–365, March 2019. ISSN 1573-0670. doi: 10.1007/s10817-017-9431-7. 1.1, 1.2.2, 1.2.3, 2, 1.2.3, 2.2, 5.2.3, 5.2.3, 8.6, 8.6

[53] Krishnendu Chatterjee, Hongfei Fu, and Amir Kafshdar Goharshady. Non-polynomial Worst-Case Analysis of Recursive Programs. *ACM Transactions on Programming Languages and Systems*, 41(4), October 2019. ISSN 0164-0925. doi: 10.1145/3339984. 1.1, 5.2.1

[54] Ezgi Çiçek, Deepak Garg, and Umut Acar. Refinement types for incremental computational complexity. In Jan Vitek, editor, *Programming Languages and Systems*, pages 406–431, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg. ISBN 978-3-662-46669-8. 5.2.3

[55] Ezgi Çiçek, Gilles Barthe, Marco Gaboardi, Deepak Garg, and Jan Hoffmann. Relational cost analysis. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL '17, pages 316–329, New York, NY, USA, January 2017. Association for Computing Machinery. ISBN 978-1-4503-4660-3. doi: 10.1145/3009837.3009858. 4, 5.2.3

[56] Ezgi Çiçek, Mehdi Bouaziz, Sungkeun Cho, and Dino Distefano. Static Resource Analysis at Scale (Extended Abstract). In *Proc. 27th International Static Analysis Symposium*, volume 12389 of *Lecture Notes in Computer Science*, pages 3–6, Cham, 2021. Springer. ISBN 978-3-030-65474-0. doi: 10.1007/978-3-030-65474-0_1. 1.1

[57] Koen Claessen and John Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, ICFP '00, pages 268–279, New York, NY, USA, 2000. Association for Computing Machinery. ISBN 1581132026. doi: 10.1145/351240.351266. URL https://doi.org/10.1145/351240.351266. 5.3.2, 7.2.3

[58] Edmund Melson Clarke. Programming language constructs for which it is impossible to obtain good hoare axiom systems. *J. ACM*, 26(1):129–147, January 1979. ISSN 0004-5411. doi: 10.1145/322108.322121. URL https://doi.org/10.1145/322108.322121. 6.3.3

[59] Stephen A. Cook. Soundness and completeness of an axiom system for program verification. *SIAM J. Comput.*, 7(1):70–90, February 1978. ISSN 0097-5397. doi: 10.1137/0207005. URL https://doi.org/10.1137/0207005. 6.3.3

[60] Emilio Coppa, Camil Demetrescu, and Irene Finocchi. Input-sensitive profiling. In *Proc. 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 89–98, New York, NY, USA, June 2012. ACM. ISBN 978-1-4503-1205-9. doi: 10.1145/2254064.2254076. 1.1, 1.2.2, 1.2.2, 2.1, 5.2.2, 5.2.2, 6.3.3, 7.3, 7.3.2

[61] Thomas H. Cormen, Charles Eric Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, Cambridge, Massachusetts, fourth edition. edition, 2022/2022. ISBN 978-0-262-04630-5. 8.5.3

[62] Karl Crary and Stephnie Weirich. Resource bound certification. In *Proc. 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 184–198, New York, NY, USA, 2000. ACM. ISBN 1-58113-125-9. doi: 10.1145/325694.325716. 1.1, 5.2.3

[63] Marco F. Cusumano-Towner, Feras A. Saad, Alexander K. Lew, and Vikash K. Mansinghka. Gen: a general-purpose probabilistic programming system with programmable inference. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, pages 221–236, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450367127. doi: 10.1145/3314221.3314642. URL https://doi.org/10.1145/3314221.3314642. 7.1.2, 7.7.3

[64] Joseph W. Cutler, Daniel R. Licata, and Norman Danner. Denotational recurrence extraction for amortized analysis. *Proc. ACM Program. Lang.*, 4(ICFP), August 2020. doi: 10.1145/3408979. 1.1, 3.1, 4.1, 5.2.1

[65] U. Dal Lago. Implicit computation complexity in higher-order programming languages: A survey in memory of Martin Hofmann. *Mathematical Structures in Computer Science*, 32(6):760–776, 2022. doi: 10.1017/S0960129521000505. 5.1, 5.1

[66] Ugo Dal Lago. *A Short Introduction to Implicit Computational Complexity*, pages 89–109. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012. ISBN 978-3-642-31485-8. doi: 10.1007/978-3-642-31485-8_3. URL https://doi.org/10.1007/978-3-642-31485-8_3. 5.1

[67] Nils Anders Danielsson. Lightweight semiformal time complexity analysis for purely functional data structures. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '08, pages 133–144, New York, NY, USA, 2008. Association for Computing Machinery. ISBN 978-1-59593-689-9. doi: 10.1145/1328438.1328457. 1.1, 5.2.3

[68] Norman Danner, Daniel R. Licata, and Ramyaa Ramyaa. Denotational cost semantics for functional languages with inductive types. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, ICFP 2015, pages 140–151, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450336697. doi: 10.1145/2784731.2784749. URL https://doi.org/10.1145/2784731.2784749. 5.2.1

[69] Ankush Das, Jan Hoffmann, and Frank Pfenning. Work Analysis with Resource-Aware Session Types. In *Proc. 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*, pages 305–314, New York, NY, USA, July 2018. ACM. ISBN 978-1-4503-5583-4. doi: 10.1145/3209108.3209146. 5.2.1

[70] Ankush Das, Stephanie Balzer, Jan Hoffmann, Frank Pfenning, and Ishani Santurkar. Resource-Aware Session Types for Digital Contracts, 2019. 5.2.1

[71] Tom de Jong. Domain theory in constructive and predicative univalent foundations, 2023. 8.2.2, 8.2.2

[72] Leonardo De Moura and Nikolaj Bjørner. Z3: an efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 3540787992. 5.2.3, 8.7

[73] Camil Demetrescu, Irene Finocchi, and Giuseppe F. Italiano. Algorithm engineering, algorithmics column. *Bull. EATCS*, 79:48–63, 2003. 5.2.2

[74] Francisco Demontiê, Junio Cezar, Mariza Bigonha, Frederico Campos, and Fernando Magno Quintão Pereira. Automatic Inference of Loop Complexity Through Polynomial Interpolation. In Alberto Pardo and S. Doaitse Swierstra, editors, *Proc. 19th Brazilian Symposium on Programming Languages*, volume 9325 of *Lecture Notes in Computer Science*, pages 1–15, Cham, 2015. Springer. ISBN 978-3-319-24012-1. doi: 10.1007/978-3-319-24012-1_1. 1.1, 1.2.2, 2.1, 5.2.2

[75] P. Erdös and A. Rényi. On random graphs i. *Publicationes Mathematicae Debrecen*, 6: 290–297, 1959. doi: 10.5486/PMD.1959.6.3-4.12. 8.5.3

[76] Inc. Facebook. Infer Website - Cost: Runtime Complexity Analysis. https://fbinfer.com/docs/checker-cost, 2024. 1.1

[77] Azadeh Farzan and Zachary Kincaid. Compositional recurrence analysis. In *2015 Formal Methods in Computer-Aided Design (FMCAD)*, pages 57–64, 2015. doi: 10.1109/FMCAD.2015.7542253. 5.2.1

[78] Tor Erlend Fjelde, Kai Xu, David Widmann, Mohamed Tarek, Cameron Pfiffer, Martin Trapp, Seth D. Axen, Xianda Sun, Markus Hauru, Penelope Yong, Will Tebbutt, Zoubin Ghahramani, and Hong Ge. Turing.jl: a general-purpose probabilistic programming language. *ACM Trans. Probab. Mach. Learn.*, February 2025. doi: 10.1145/3711897. URL https://doi.org/10.1145/3711897. Just Accepted. 7.1.2

[79] Cormac Flanagan. Hybrid type checking. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '06, pages 245–256, New York, NY, USA, 2006. Association for Computing Machinery. ISBN 1595930272. doi: 10.1145/1111037.1111059. URL https://doi.org/10.1145/1111037.1111059. 7.7.3, 7.7.3

[80] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, PLDI '93, pages 237–247, New York, NY, USA, 1993. Association for Computing Machinery. ISBN 0897915984. doi: 10.1145/155090.155113. URL https://doi.org/10.1145/155090.155113. 1

[81] Antonio Flores-Montoya and Reiner Hähnle. Resource analysis of complex programs with cost equations. In Jacques Garrigue, editor, *Programming Languages and Systems*, pages 275–295, Cham, 2014. Springer International Publishing. ISBN 978-3-319-12736-1. 5.2.1

[82] John Forrest, Stefan Vigerske, Ted Ralphs, John Forrest, Lou Hafer, jpfasano, Haroldo Gambini Santos, Jan-Willem, Matthew Saltzman, Bjarni Kristjansson, h-i gassmann, Alan King, Arevall, Pierre Bonami, Ruan Luies, Samuel Brito, and to st. coin-

or/clp: Release releases/1.17.9, October 2023. URL https://doi.org/10.5281/zenodo.10041272. 4.1, 4.4, 5.2.1, 8.5.3

[83] Yannick Forster, Fabian Kunze, Gert Smolka, and Maximilian Wuttke. A Mechanised Proof of the Time Invariance Thesis for the Weak Call-By-Value λ-Calculus. In Liron Cohen and Cezary Kaliszyk, editors, *12th International Conference on Interactive Theorem Proving (ITP 2021)*, volume 193 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 19:1–19:20, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. ISBN 978-3-95977-188-7. doi: 10.4230/LIPIcs.ITP.2021.19. URL https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ITP.2021.19. 5.1

[84] Steven Fortune and James Wyllie. Parallelism in random access machines. In *Proceedings of the Tenth Annual ACM Symposium on Theory of Computing*, STOC '78, pages 114–118, New York, NY, USA, 1978. Association for Computing Machinery. ISBN 9781450374378. doi: 10.1145/800133.804339. URL https://doi.org/10.1145/800133.804339. 5.1

[85] Marco Gaboardi and Simona Ronchi Della Rocca. A soft type assignment system for λ-calculus. In *Proceedings of the 21st International Conference, and Proceedings of the 16th Annuall Conference on Computer Science Logic*, CSL'07/EACSL'07, pages 253–267, Berlin, Heidelberg, 2007. Springer-Verlag. ISBN 3540749144. 5.1

[86] Marco Gaboardi, Jean-Yves Marion, and Simona Ronchi Della Rocca. An implicit characterization of pspace. *ACM Trans. Comput. Logic*, 13(2), April 2012. ISSN 1529-3785. doi: 10.1145/2159531.2159540. URL https://doi.org/10.1145/2159531.2159540. 5.1

[87] David Gajser. Verifying time complexity of deterministic turing machines. *CoRR*, abs/1307.3648, 2013. URL http://arxiv.org/abs/1307.3648. (i), (ii), 5.1, 6.3.1, 6.3.1, 6.3.2, 6.3.1, 6.3.2, 6.3.1, 1, 6.3.2, 10.1

[88] David Gajser. Verifying time complexity of turing machines. *Theoretical Computer Science*, 600:86–97, 2015. ISSN 0304-3975. doi: https://doi.org/10.1016/j.tcs.2015.07.028. URL https://www.sciencedirect.com/science/article/pii/S0304397515006623. (i), (ii), 5.1, 5.1, 6, 6.2.2, 6.3.1, 6.3.1, 6.3.2, 6.3.1, 6.3.2, 6.3.1, 1, 6.3.2, 10.1

[89] Andrew Gelman, John Carlin, Hal Stern, David Dunson, Aki Vehtari, and Donald Rubin. *Bayesian Data Analysis, Third Edition, 3rd Edition*. Texts in statistical science. CRC Press, 3rd edition edition, 2013. ISBN 9781439898222. 7.1.1, 7.1.1, 1

[90] Jürgen Giesl, Nils Lommen, Marcel Hark, and Fabian Meyer. Improving Automatic Complexity Analysis of Integer Programs. In Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, and Einar Broch Johnsen, editors, *The Logic of Software. A Tasting Menu of Formal Methods: Essays Dedicated to Reiner Hähnle on the Occasion of His 60th Birthday*, volume 13360 of *Lecture Notes in Computer Science*, pages 193–228. Springer, Cham, 2022. ISBN 978-3-031-08166-8. doi: 10.1007/978-3-031-08166-8_10. 1.1, 5.2.1

[91] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50(1):1–101, 1987. ISSN 0304-3975. doi: https://doi.org/10.1016/0304-3975(87)90045-4. URL https://www.sciencedirect.com/science/article/pii/0304397587900454. 5.1

[92] Jean-Yves Girard. Light linear logic. *Information and Computation*, 143(2):175–204, 1998. ISSN 0890-5401. doi: https://doi.org/10.1006/inco.1998.2700. URL https://www.

sciencedirect.com/science/article/pii/S0890540198927006. 5.1

[93] Jean-Yves Girard, Andre Scedrov, and Philip J. Scott. Bounded linear logic: a modular approach to polynomial-time computability. *Theor. Comput. Sci.*, 97(1):1–66, April 1992. ISSN 0304-3975. doi: 10.1016/0304-3975(92)90386-T. URL https://doi.org/10.1016/0304-3975(92)90386-T. 5.1

[94] Simon F. Goldsmith, Alex S. Aiken, and Daniel S. Wilkerson. Measuring empirical computational complexity. In *Proc. the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, pages 395–404, New York, NY, USA, September 2007. ACM. ISBN 978-1-59593-811-4. doi: 10.1145/1287624.1287681. 1.1, 1.2.2, 1.2.2, 2.1, 5.2.2, 6.3.3, 7.3, 7.3.2, 9.1

[95] Harrison Goldstein, Joseph W. Cutler, Daniel Dickstein, Benjamin C. Pierce, and Andrew Head. Property-based testing in practice. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ICSE '24, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400702174. doi: 10.1145/3597503.3639581. URL https://doi.org/10.1145/3597503.3639581. 5.3.2

[96] Noah D. Goodman, Vikash K. Mansinghka, Daniel Roy, Keith Bonawitz, and Joshua B. Tenenbaum. Church: a language for generative models. In *Proceedings of the Twenty-Fourth Conference on Uncertainty in Artificial Intelligence*, UAI'08, pages 220–229, Arlington, Virginia, USA, 2008. AUAI Press. ISBN 0974903949. 7.1.2

[97] Bernd Grobauer. Cost recurrences for DML programs. In *Proc. 6th ACM SIGPLAN International Conference on Functional Programming*, pages 253–264, New York, NY, USA, 2001. ACM. ISBN 1-58113-415-0. doi: 10.1145/507635.507666. 1.1, 5.2.1

[98] Harrison Grodin, Yue Niu, Jonathan Sterling, and Robert Harper. Decalf: A directed, effectful cost-aware logical framework. *Proc. ACM Program. Lang.*, 8(POPL):273–301, 2024. doi: 10.1145/3632852. URL https://doi.org/10.1145/3632852. 1.1, 5.2.3

[99] Jessie Grosen, David M. Kahn, and Jan Hoffmann. Automatic Amortized Resource Analysis with Regular Recursive Types. In *Proc. 38th Annual ACM/IEEE Symposium on Logic in Computer Science*, pages 1–14, New York, NY, USA, June 2023. IEEE. doi: 10.1109/LICS56636.2023.10175720. 1.2.2, 5.2.1

[100] Armaël Guéneau, Arthur Charguéraud, and François Pottier. A Fistful of Dollars: Formalizing Asymptotic Complexity Claims via Deductive Program Verification. In Amal Ahmed, editor, *Programming Languages and Systems*, Lecture Notes in Computer Science, pages 533–560, Cham, 2018. Springer International Publishing. ISBN 978-3-319-89884-1. doi: 10.1007/978-3-319-89884-1_19. 1.1

[101] Bhargav S. Gulavani and Sumit Gulwani. A numerical abstract domain based on expression abstraction and max operator with application in timing analysis. In Aarti Gupta and Sharad Malik, editors, *Computer Aided Verification*, pages 370–384, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. ISBN 978-3-540-70545-1. 1.1, 5.2.1

[102] Sumit Gulwani, Krishna K. Mehra, and Trishul Chilimbi. SPEED: Precise and efficient static estimation of program computational complexity. In *Proc. 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 127–139,

New York, NY, USA, 2009. ACM. ISBN 978-1-60558-379-2. doi: 10.1145/1480881.1480898. 1.1, 5.2.1, 8.8.1

[103] Petr Hájek. Arithmetical hierarchy and complexity of computation. *Theoretical Computer Science*, 8(2):227–237, 1979. ISSN 0304-3975. doi: https://doi.org/10.1016/0304-3975(79)90046-X. URL https://www.sciencedirect.com/science/article/pii/030439757990046X. 5.1, 6, 6.2.2, 6.3.1

[104] Martin A. T. Handley, Niki Vazou, and Graham Hutton. Liquidate your assets: Reasoning about resource usage in liquid Haskell. *Proc. ACM Program. Lang.*, 4(POPL), December 2019. doi: 10.1145/3371092. 1.1, 5.2.3

[105] Robert Harper. *Practical foundations for programming languages*. Cambridge University Press, New York NY, 2nd ed. edition, 2016. ISBN 9781316576892. 8.1.2, 8.3.1

[106] Robert Harper. Pfpl supplement: Pcf by value. https://www.cs.cmu.edu/~rwh/pfpl/supplements/pcfv.pdf, 2019. 8.1.1, 8.1.2

[107] Vincent J. Hellendoorn, Christian Bird, Earl T. Barr, and Miltiadis Allamanis. Deep learning type inference. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2018, pages 152–162, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450355735. doi: 10.1145/3236024.3236051. URL https://doi.org/10.1145/3236024.3236051. 7.7.3

[108] Manuel V. Hermenegildo, Germán Puebla, Francisco Bueno, and Pedro López-García. Integrated program debugging, verification, and optimization using abstract interpretation (and the ciao system preprocessor). *Science of Computer Programming*, 58(1):115–140, 2005. ISSN 0167-6423. doi: https://doi.org/10.1016/j.scico.2005.02.006. URL https://www.sciencedirect.com/science/article/pii/S0167642305000468. Special Issue on the Static Analysis Symposium 2003. 5.2.1

[109] R. Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60, 1969. ISSN 00029947. URL http://www.jstor.org/stable/1995158. 4.4

[110] Jennifer A. Hoeting, David Madigan, Adrian E. Raftery, and Chris T. Volinsky. Bayesian model averaging: A tutorial. *Statistical Science*, 14(4):382–401, 1999. ISSN 08834237. URL http://www.jstor.org/stable/2676803. 2.2, 8.5.2

[111] Matthew D. Hoffman and Andrew Gelman. The no-u-turn sampler: adaptively setting path lengths in hamiltonian monte carlo. *J. Mach. Learn. Res.*, 15(1):1593–1623, jan 2014. ISSN 1532-4435. 7.1.2, 7.5.3, 7.5.3, 7.7.2, 7.7.3, 8.5.2, 8.5.3

[112] Jan Hoffmann. *Types with Potential: Polynomial Resource Bounds via Automatic Amortized Analysis*. PhD thesis, Ludwig-Maximilians-Universität München, October 2011. URL https://edoc.ub.uni-muenchen.de/13955/. (document), 1.3, 2.1, 2.1, 2.2, 3.1, 3.2, 3.2, 4, 4.3, 4.3, 4.4, 4.4, 4.4, 5.1, 6, 6.2, 6.2.2, 6.3.1, 6.3.2, 6.3.3, 7.4.1, 7.4.3, 7.5.1, 7.6.1, **Q2**, 8.1, 8.6, A.10

[113] Jan Hoffmann and Martin Hofmann. Amortized resource analysis with polynomial po-

tential. In Andrew D. Gordon, editor, *Programming Languages and Systems*, pages 287–306, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg. ISBN 978-3-642-11957-6. 1.1, 1.2.2, 1.3, 3.1, 4, 4.1, 4.3, 4.3.1, 4.3, 4.4, 5.1, 5.2.1, 6.2, 6.3.3

[114] Jan Hoffmann and Steffen Jost. Two decades of automatic amortized resource analysis. *Mathematical Structures in Computer Science*, 32(6):729–759, June 2022. ISSN 0960-1295, 1469-8072. doi: 10.1017/S0960129521000487. URL https://www.cambridge.org/core/journals/mathematical-structures-in-computer-science/article/two-decades-of-automatic-amortized-resource-analysis/9A47A8663CD8A7147E2F17865C368094. 1.2.2, 1, 5.2.1

[115] Jan Hoffmann and Zhong Shao. Automatic Static Cost Analysis for Parallel Programs. In Jan Vitek, editor, *Proc. 24th European Symposium on Programming Languages and Systems*, volume 9032 of *Lecture Notes in Computer Science*, pages 132–157, Berlin, 2015. Springer. ISBN 978-3-662-46669-8. doi: 10.1007/978-3-662-46669-8_6. 5.2.1

[116] Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. Multivariate amortized resource analysis. In *Proc. 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 357–370, New York, NY, USA, January 2011. ACM. doi: 10.1145/1926385.1926427. 1.1, 1.2.2, 1.3, 2.1, 2.2, 3.1, 3.2, 4, 4.1, 4.3, 4.4, 5.1, 5.2.1, 6.2, 6.3.3, 7.3.5, 7.4.3

[117] Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. Resource aware ML. In P. Madhusudan and Sanjit A. Seshia, editors, *Proc. 24 International Conference on Computer Aided Verification*, volume 7358 of *Lecture Notes in Computer Science*, pages 781–786, Berlin, 2012. Springer. doi: 10.1007/978-3-642-31424-7_64. 1.2.2, 1.2.2, 4, 2.1, 2.1, 2.2, 3.1, 4, 4.1, 4.1, 4.4, 4.4, 5.2.1, 7.4.1, 7.5, 7.5.1, 7.5.3, 8.1.1, 8.5.3, 9.4.2

[118] Jan Hoffmann, Ankush Das, and Shu-Chun Weng. Towards automatic resource bound analysis for OCaml. In *Proc. 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 359–373, New York, NY, USA, January 2017. ACM. doi: 10.1145/3009837.3009842. 1.2.2, 1.2.2, 2.1, 2.1, 2.2, 3.1, 4, 4.1, 4.1, 4.4, 4.4, 5.2.1, 7.4.1, 7.5, 7.5.1, 7.5.3, 8.1.1, 8.5.3, 9.4.2

[119] Martin Hofmann. A mixed modal/linear lambda calculus with applications to bellantoni-cook safe recursion. In *Selected Papers from The11th International Workshop on Computer Science Logic*, CSL '97, pages 275–294, Berlin, Heidelberg, 1997. Springer-Verlag. ISBN 3540645705. 5.1

[120] Martin Hofmann. Linear types and non size-increasing polynomial time computation. In *Proceedings of the 14th Annual IEEE Symposium on Logic in Computer Science*, LICS '99, page 464, USA, 1999. IEEE Computer Society. ISBN 0769501583. 5.1, 5.2.1

[121] Martin Hofmann. A type system for bounded space and functional in-place update. *Nordic J. of Computing*, 7(4):258–289, December 2000. ISSN 1236-6064. 5.1, 5.2.1

[122] Martin Hofmann and Steffen Jost. Static prediction of heap space usage for first-order functional programs. In *Proc. 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 185–197, New York, NY, USA, 2003. ACM. doi: 10.1145/604131.604148. 1.1, 1.2.2, 1, 4, 5.1, 5.2.1

[123] Martin Hofmann and Georg Moser. Amortised resource analysis and typed polynomial interpretations. In *Rewriting and Typed Lambda Calculi*, volume 8560 of *Lecture Notes in Computer Science*, pages 272–286, Heidelberg, 2014. Springer. doi: 10.1007/978-3-319-08918-8_19. 1.1, 5.2.1

[124] Martin Hofmann and Georg Moser. Analysis of logarithmic amortised complexity, July 2018. 1.1

[125] Martin Hofmann and Mariela Pavlova. Elimination of ghost variables in program logics. In Gilles Barthe and Cédric Fournet, editors, *Trustworthy Global Computing*, pages 1–20, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. ISBN 978-3-540-78663-4. 8.2, 8.8.1

[126] Martin Hofmann, Lorenz Leutgeb, David Obwaller, Georg Moser, and Florian Zuleger. Type-based analysis of logarithmic amortised complexity. *Mathematical Structures in Computer Science*, 32(6):794–826, June 2022. ISSN 0960-1295, 1469-8072. doi: 10.1017/S0960129521000232. 4, 5.1, 5.2.1

[127] Ling Huang, Jinzhu Jia, Bin Yu, Byung-Gon Chun, Petros Maniatis, and Mayur Naik. Predicting execution time of computer programs using sparse polynomial regression. In *Advances in Neural Information Processing Systems*, volume 23, pages 883–891, Red Hook, NY, USA, 2010. Curran Associates Inc. 1.1, 1.2.2, 1.2.2, 2.1, 5.2.2

[128] Xiuqi Huang, Shiyi Cao, Yuanning Gao, Xiaofeng Gao, and Guihai Chen. LightPro: Lightweight probabilistic workload prediction framework for database-as-a-service. In Claudio Agostino Ardagna, Nimanthi L. Atukorala, Boualem Benatallah, Athman Bouguettaya, Fabio Casati, Carl K. Chang, Rong N. Chang, Ernesto Damiani, Chirine Ghedira Guégan, Robert Ward, Fatos Xhafa, Xiaofei Xu, and Jia Zhang, editors, *IEEE International Conference on Web Services, ICWS 2022, Barcelona, Spain, July 10-16, 2022*, pages 160–169, New York, NY, USA, 2022. IEEE. doi: 10.1109/ICWS55610.2022.00036. 1.1

[129] David Hubbard, Benoit Rostykus, Yves Raimond, and Tony Jebara. Beta survival models. In Russell Greiner, Neeraj Kumar, Thomas Alexander Gerds, and Mihaela van der Schaar, editors, *Proceedings of AAAI Spring Symposium on Survival Prediction — Algorithms, Challenges, and Applications*, volume 146 of *Proceedings of Machine Learning Research*, pages 22–39. PMLR, March 2021. 7.3.3

[130] John Hughes, Lars Pareto, and Amr Sabry. Proving the correctness of reactive systems using sized types. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '96, pages 410–423, New York, NY, USA, 1996. Association for Computing Machinery. ISBN 0897917693. doi: 10.1145/237721.240882. URL https://doi.org/10.1145/237721.240882. 10.2

[131] Didier Ishimwe, KimHao Nguyen, and ThanhVu Nguyen. Dynaplex: Analyzing program complexity using dynamically inferred recurrence relations. *Proc. ACM Program. Lang.*, 5(OOPSLA), October 2021. doi: 10.1145/3485515. 1.1, 1.2.2, 5.2.2, 7.7.1, 9.1

[132] F. Jelinek, J. D. Lafferty, and R. L. Mercer. Basic methods of probabilistic context free grammars. In Pietro Laface and Renato De Mori, editors, *Speech Recognition and Understanding*, pages 345–360, Berlin, Heidelberg, 1992. Springer Berlin Heidelberg. ISBN

978-3-642-76626-8. 10.2

[133] David S. Johnson. A theoretician's guide to the experimental analysis of algorithms. In Michael H. Goldwasser, David S. Johnson, and Catherine C. McGeoch, editors, *Data Structures, Near Neighbor Searches, and Methodology: Fifth and Sixth DIMACS Implementation Challenges*, volume 59 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 215–250, Piscataway, NJ, USA, 1999. DIMACS/AMS. doi: 10.1090/dimacs/059/11. 5.2.2

[134] Steffen Jost, Kevin Hammond, Hans-Wolfgang Loidl, and Martin Hofmann. Static determination of quantitative resource usage for higher-order programs. *Proc. 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 45(1):223–236, January 2010. ISSN 0362-1340. doi: 10.1145/1706299.1706327. 1.2.2, 5.2.1

[135] David Kahn. *Leveraging Linearity to Improve Automatic Amortized Resource Analysis*. PhD thesis, Carnegie Mellon University, 8 2024. URL https://kilthub.cmu.edu/articles/thesis/Leveraging_Linearity_to_Improve_Automatic_Amortized_Resource_Analysis/26380885. 3.1, 3.2, 4.4, 5.2.1

[136] David M. Kahn and Jan Hoffmann. Exponential Automatic Amortized Resource Analysis. In Jean Goubault-Larrecq and Barbara König, editors, *Proc. 23rd International Conference on Foundations of Software Science and Computation Structures*, volume 12077 of *Lecture Notes in Computer Science*, pages 359–380, Cham, 2020. Springer. ISBN 978-3-030-45231-5. doi: 10.1007/978-3-030-45231-5_19. 1.1, 5.2.1

[137] David M. Kahn and Jan Hoffmann. Automatic amortized resource analysis with the quantum physicist's method. *Proceedings of the ACM on Programming Languages*, 5 (ICFP):76:1–76:29, August 2021. doi: 10.1145/3473581. URL https://doi.org/10.1145/3473581. 3.1

[138] Haim Kaplan, Nira Shafrir, and Robert E. Tarjan. Union-find with deletions. In *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '02, pages 19–28, USA, 2002. Society for Industrial and Applied Mathematics. ISBN 089871513X. 8.6

[139] G. A. Kavvos, Edward Morehouse, Daniel R. Licata, and Norman Danner. Recurrence extraction for functional programs through call-by-push-value. *Proc. ACM Program. Lang.*, 4(POPL), December 2019. doi: 10.1145/3371083. 1.1, 5.2.1

[140] Zachary Kincaid, Jason Breck, Ashkan Forouhi Boroujeni, and Thomas Reps. Compositional recurrence analysis revisited. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pages 248–262, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450349888. doi: 10.1145/3062341.3062373. URL https://doi.org/10.1145/3062341.3062373. 5.2.1

[141] Zachary Kincaid, John Cyphert, Jason Breck, and Thomas Reps. Non-linear reasoning for invariant synthesis. *Proc. ACM Program. Lang.*, 2(POPL), December 2017. doi: 10.1145/3158142. 1.1, 5.2.1

[142] Stephen Cole Kleene. General recursive functions of natural numbers. *Mathematische annalen*, 112(1):727–742, 1936. 5.1

[143] Tristan Knoth, Di Wang, Nadia Polikarpova, and Jan Hoffmann. Resource-guided program synthesis. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, pages 253–268, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450367127. doi: 10.1145/3314221.3314602. URL https://doi.org/10.1145/3314221.3314602. 5.2.1, 5.2.3

[144] Tristan Knoth, Di Wang, Adam Reynolds, Jan Hoffmann, and Nadia Polikarpova. Liquid resource types. *Proceedings of the ACM on Programming Languages*, 4(ICFP):106:1–106:29, August 2020. doi: 10.1145/3408988. 1.1, 5.2.1, 5.2.3

[145] Kenneth Knowles and Cormac Flanagan. Hybrid type checking. *ACM Trans. Program. Lang. Syst.*, 32(2), February 2010. ISSN 0164-0925. doi: 10.1145/1667048.1667051. URL https://doi.org/10.1145/1667048.1667051. 7.7.3, 7.7.3

[146] John R. Koza. Genetic programming: a paradigm for genetically breeding populations of computer programs to solve problems. Technical report, Stanford, CA, USA, 1990. 1, 5

[147] John R. Koza. *Genetic programming : on the programming of computers by means of natural selection*. Complex adaptive systems. MIT Press, Cambridge, Mass, 1992. ISBN 0262111705. 1, 5

[148] Alexander Kraskov, Harald Stögbauer, and Peter Grassberger. Estimating mutual information. *Phys. Rev. E*, 69:066138, Jun 2004. doi: 10.1103/PhysRevE.69.066138. URL https://link.aps.org/doi/10.1103/PhysRevE.69.066138. 8.5.2

[149] Yves Lafont. Soft linear logic and polynomial time. *Theoretical Computer Science*, 318 (1):163–180, 2004. ISSN 0304-3975. doi: https://doi.org/10.1016/j.tcs.2003.10.018. URL https://www.sciencedirect.com/science/article/pii/S0304397503005231. Implicit Computational Complexity. 5.1

[150] Ugo Dal Lago and Marco Gaboardi. Linear dependent types and relative completeness. In *Proc. 26th IEEE Symposium on Logic in Computer Science*, pages 133–142, New York, NY, USA, 2011. IEEE. doi: 10.1109/LICS.2011.22. 1.2.2, 1.2.3, 5.2.3

[151] Leonidas Lampropoulos, Diane Gallois-Wong, Cătălin Hriţcu, John Hughes, Benjamin C. Pierce, and Li-yao Xia. Beginner's luck: a language for property-based generators. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL '17, pages 114–129, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450346603. doi: 10.1145/3009837.3009868. URL https://doi.org/10.1145/3009837.3009868. 5.3.2

[152] Xuan-Bach D. Le, Corina Pasareanu, Rohan Padhye, David Lo, Willem Visser, and Koushik Sen. Saffron: Adaptive grammar-based fuzzing for worst-case analysis. *SIGSOFT Softw. Eng. Notes*, 44(4):14, December 2019. ISSN 0163-5948. doi: 10.1145/3364452.3364455. URL https://doi.org/10.1145/3364452.3364455. 5.3.1, 9.1

[153] Allan M. M. Leal. autodiff, a modern, fast and expressive C++ library for automatic differentiation. `https://autodiff.github.io`, 2018. URL https://autodiff.github.io. 7.5.3, 7.6.2

[154] Daniel Leivant. Ramified recurrence and computational complexity iii: Higher type

recurrence and elementary complexity. *Annals of Pure and Applied Logic*, 96(1):209–229, 1999. ISSN 0168-0072. doi: https://doi.org/10.1016/S0168-0072(98)00040-2. URL https://www.sciencedirect.com/science/article/pii/S0168007298000402. 5.1

[155] Caroline Lemieux, Rohan Padhye, Koushik Sen, and Dawn Song. PerfFuzz: Automatically generating pathological inputs. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2018, pages 254–265, New York, NY, USA, July 2018. Association for Computing Machinery. ISBN 978-1-4503-5699-2. doi: 10.1145/3213846.3213874. 1.1, 5.3.1, 9.1, 9.3.2

[156] Lorenz Leutgeb, Georg Moser, and Florian Zuleger. ATLAS: Automated Amortised Complexity Analysis of Self-adjusting Data Structures. In *Proc. 33rd International Conference on Computer Aided Verification*, volume 12760 of *Lecture Notes in Computer Science*, pages 99–122, Berlin, July 2021. Springer-Verlag. doi: 10.1007/978-3-030-81688-9_5. 1.1, 4, 5.2.1

[157] Lorenz Leutgeb, Georg Moser, and Florian Zuleger. Automated Expected Amortised Cost Analysis of Probabilistic Data Structures. In Sharon Shoham and Yakir Vizel, editors, *Proc. 34th International Conference on Computer Aided Verification*, volume 13372 of *Lecture Notes in Computer Science*, pages 70–91, Cham, 2022. Springer. doi: 10.1007/978-3-031-13188-2_4. 5.2.1

[158] Benjamin Lichtman and Jan Hoffmann. Arrays and References in Resource Aware ML. In Dale Miller, editor, *Proc. 2nd International Conference on Formal Structures for Computation and Deduction*, volume 84 of *Leibniz International Proceedings in Informatics*, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi: 10.4230/LIPIcs.FSCD.2017.26. 1.2.2, 5.2.1

[159] P. Lopez-Garcia, L. Darmawan, M. Klemen, U. Liqat, F. Bueno, and M. V. Hermenegildo. Interval-based resource usage verification by translation into horn clauses and an application to energy consumption. *Theory and Practice of Logic Programming*, 18(2):167–223, 2018. doi: 10.1017/S1471068418000042. 5.2.1

[160] Kasper Luckow, Rody Kersten, and Corina Păsăreanu. Symbolic Complexity Analysis Using Context-Preserving Histories. In *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 58–68, March 2017. doi: 10.1109/ICST.2017.13. 1.1, 5.3.1, 9.1

[161] George Markowsky. Chain-complete posets and directed sets with applications. *Algebra Universalis*, 6:53–68, 12 1976. doi: 10.1007/BF02485815. 4

[162] C.J.H. McDiarmid and R.B. Hayward. Large Deviations for Quicksort. *J. Algorithms*, 21 (3):476–507, November 1996. ISSN 0196-6774. doi: 10.1006/jagm.1996.0055. 9.1

[163] Colin McDiarmid. Quicksort and Large Deviations. In Antonín Kučera, Thomas A. Henzinger, Jaroslav Nešetřil, Tomáš Vojnar, and David Antoš, editors, *Mathematical and Engineering Methods in Computer Science*, pages 43–52, Berlin, Heidelberg, 2013. Springer. ISBN 978-3-642-36046-6. doi: 10.1007/978-3-642-36046-6_5. 9.1

[164] Colin McDiarmid and Ryan Hayward. Strong concentration for Quicksort. In *Proceedings of the Third Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '92, pages 414–421, USA, September 1992. Society for Industrial and Applied Mathematics. ISBN 978-0-

89791-466-6. 9.1

[165] Catherine McGeoch, Peter Sanders, Rudolf Fleischer, Paul R. Cohen, and Doina Precup. Using Finite Experiments to Study Asymptotic Performance. In Rudolf Fleischer, Bernard Moret, and Erik Meineche Schmidt, editors, *Experimental Algorithmics: From Algorithm Design to Robust and Efficient Software*, Lecture Notes in Computer Science, pages 93–126. Springer, Berlin, 2002. ISBN 978-3-540-36383-5. doi: 10.1007/3-540-36383-1_5. 5.2.2

[166] Catherine C. McGeoch. *A Guide to Experimental Algorithmics*. Cambridge University Press, 2012. 5.2.2

[167] Catherine Cole Mcgeoch. *Experimental analysis of algorithms*. PhD thesis, Carnegie Mellon University, USA, 1986. AAI8802935. 5.2.2

[168] Glen Mével, Jacques-Henri Jourdan, and François Pottier. Time Credits and Time Receipts in Iris. In Luís Caires, editor, *Proc. 28th European Symposium on Programming*, volume 11423 of *Lecture Notes in Computer Science*, pages 3–29, Cham, 2019. Springer. ISBN 978-3-030-17184-1. doi: 10.1007/978-3-030-17184-1_1. 1.1, 2, 2.2, 3.1, 5.2.3, 5.2.3, 6.3.3, 8.6

[169] Robin Milner. Models of lcf. Technical report, Stanford, CA, USA, 1973. 8.2.2

[170] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, 1978. ISSN 0022-0000. doi: https://doi.org/10.1016/0022-0000(78)90014-4. URL https://www.sciencedirect.com/science/article/pii/0022000078900144. 4.4

[171] Hadi Mohasel Afshar and Justin Domke. Reflection, Refraction, and Hamiltonian Monte Carlo. In *Advances in Neural Information Processing Systems*, volume 28, pages 3007–3015, Cambridge, MA, USA, 2015. MIT Press. 1.2.2, 2.1, 7.3.2, 7.5.3, 7.7.2, 8.8.3, 10.1, 10.2

[172] Alexandre Moine, Arthur Charguéraud, and François Pottier. A High-Level Separation Logic for Heap Space under Garbage Collection. *Proc. ACM Program. Lang.*, 7(POPL), January 2023. doi: 10.1145/3571218. 1.1, 5.2.3

[173] Bernard M. E. Moret. Towards a discipline of experimental algorithmics. In Michael H. Goldwasser, David S. Johnson, and Catherine C. McGeoch, editors, *Data Structures, Near Neighbor Searches, and Methodology: Fifth and Sixth DIMACS Implementation Challenges*, volume 59 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 197–213, Piscataway, NJ, USA, 1999. DIMACS/AMS. doi: 10.1090/dimacs/059/10. 5.2.2

[174] Georg Moser and Manuel Schneckenreither. Automated amortised resource analysis for term rewrite systems. *Science of Computer Programming*, 185, January 2020. ISSN 0167-6423. doi: 10.1016/j.scico.2019.102306. 1.1, 5.2.1

[175] Niels Mündler, Jingxuan He, Hao Wang, Koushik Sen, Dawn Song, and Martin Vechev. Type-constrained code generation with language models. *Proc. ACM Program. Lang.*, 9 (PLDI), June 2025. doi: 10.1145/3729274. URL https://doi.org/10.1145/3729274. 10.2

[176] Van Chan Ngo, Mario Dehesa-Azuara, Matthew Fredrikson, and Jan Hoffmann. Verifying and Synthesizing Constant-Resource Implementations with Types. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 710–728, New York, NY, USA, May 2017.

IEEE Computer Society. doi: 10.1109/SP.2017.53. 1.1, (i)

[177] Van Chan Ngo, Quentin Carbonneaux, and Jan Hoffmann. Bounded expectations: Resource analysis for probabilistic programs. In *Proc. 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 496–512, New York, NY, USA, June 2018. ACM. doi: 10.1145/3192366.3192394. 5.2.1

[178] Tobias Nipkow and Hauke Brinkop. Amortized Complexity Verified. *Journal of Automated Reasoning*, 62(3):367–391, March 2019. doi: 10.1007/s10817-018-9459-3. 1.2.2, 1.2.3, 5.2.3

[179] Yue Niu. Cost-sensitive programming, verification, and semantics, 2 2025. URL https://kilthub.cmu.edu/articles/thesis/Cost-sensitive_programming_verification_and_semantics/28352006. 8.2.2, 8.2.2

[180] Yue Niu, Jonathan Sterling, Harrison Grodin, and Robert Harper. A cost-aware logical framework. *Proc. ACM Program. Lang.*, 6(POPL):1–31, January 2022. doi: 10.1145/3498670. 1.1, 1.2.2, 1.2.3, 3.1, 5.2.3, 5.2.3, 6.3.3, 8.6, 8.8.1

[181] Yannic Noller, Rody Kersten, and Corina S. Păsăreanu. Badger: complexity analysis with fuzzing and symbolic execution. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2018, pages 322–332, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450356992. doi: 10.1145/3213846.3213868. URL https://doi.org/10.1145/3213846.3213868. 5.3.1, 9.1

[182] Ulf Norell. Dependently typed programming in agda. In *Proceedings of the 4th International Workshop on Types in Language Design and Implementation*, TLDI '09, pages 1–2, New York, NY, USA, 2009. Association for Computing Machinery. ISBN 9781605584201. doi: 10.1145/1481861.1481862. URL https://doi.org/10.1145/1481861.1481862. 5.2.3, 8.6

[183] Irene Vlassi Pandi, Earl T. Barr, Andrew D. Gordon, and Charles Sutton. Opttyper: Probabilistic type inference by optimising logical and natural constraints, 2021. URL https://arxiv.org/abs/2004.00348. 7.7.3

[184] Yun Peng, Chaozheng Wang, Wenxuan Wang, Cuiyun Gao, and Michael R. Lyu. Generative type inference for python. In *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering*, ASE '23, pages 988–999. IEEE Press, 2024. ISBN 9798350329964. doi: 10.1109/ASE56229.2023.00031. URL https://doi.org/10.1109/ASE56229.2023.00031. 7.7.3

[185] Theofilos Petsios, Jason Zhao, Angelos D. Keromytis, and Suman Jana. SlowFuzz: Automated Domain-Independent Detection of Algorithmic Complexity Vulnerabilities. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, pages 2155–2168, New York, NY, USA, October 2017. Association for Computing Machinery. ISBN 978-1-4503-4946-8. doi: 10.1145/3133956.3134073. 1.1, 5.3.1, 9.1, 9.3.2

[186] Long Pham and Jan Hoffmann. Typable Fragments of Polynomial Automatic Amortized Resource Analysis. In *DROPS-IDN/v2/Document/10.4230/LIPIcs.CSL.2021.34*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. doi: 10.4230/LIPIcs.CSL.2021.34. 1, 6.4

[187] Long Pham, Feras Saad, and Jan Hoffmann. Hybrid resource-aware ml, 2023. 7.5

[188] Long Pham, Feras A. Saad, and Jan Hoffmann. Robust resource bounds with static analysis and bayesian inference. *Proc. ACM Program. Lang.*, 8(PLDI), June 2024. doi: 10.1145/3656380. URL https://doi.org/10.1145/3656380. (document), 2, 5.2.1, 7.3, **Q2**, 8.1, 8.5.3

[189] Long Pham, Yue Niu, Nathan Glover, Feras A. Saad, and Jan Hoffmann. Integrating Resource Analyses via Resource Decomposition, March 2025. 3

[190] Andrew M. Pitts, Glynn Winskel, Marcelo Fiore, and Meven Lennon-Bertrand. Lecture notes on denotational semantics part ii of the computer science tripos 2023/24. https://www.cl.cam.ac.uk/teaching/2324/DenotSem/notes.pdf, 2023. 8.2.2, 8.2.2, 8.2.2, 8.2.3

[191] Gordon D. Plotkin. Lcf considered as a programming language. *Theoretical Computer Science*, 5(3):223–255, 1977. doi: 10.1016/0304-3975(77)90044-5. 8.1, 8.1.1, 8.2.2

[192] Gabriel Poesia, Alex Polozov, Vu Le, Ashish Tiwari, Gustavo Soares, Christopher Meek, and Sumit Gulwani. Synchromesh: Reliable code generation from pre-trained language models. In *International Conference on Learning Representations*, 2022. URL https://openreview.net/forum?id=KmtVD97J43e. 10.2

[193] Veselin Raychev, Martin Vechev, and Andreas Krause. Predicting program properties from "big code". In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pages 111–124, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450333009. doi: 10.1145/2676726.2677009. URL https://doi.org/10.1145/2676726.2677009. 7.7.3

[194] Stefano Rizzelli, Judith Rousseau, and Sonia Petrone. Empirical Bayes in Bayesian learning: Understanding a common practice, February 2024. 7.5.2

[195] Daniele Rogora, Antonio Carzaniga, Amer Diwan, Matthias Hauswirth, and Robert Soulé. Analyzing system performance with probabilistic performance annotations. In *Proc. 15th European Conference on Computer Systems*, pages 1–14, New York, NY, USA, April 2020. ACM. doi: 10.1145/3342195.3387554. 1.1, 1.2.2, 2.1, 5.2.2

[196] Nilabja Roy, Abhishek Dubey, and Aniruddha Gokhale. Efficient Autoscaling in the Cloud Using Predictive Models for Workload Forecasting. In *2011 IEEE 4th International Conference on Cloud Computing*, pages 500–507, New York, NY, USA, July 2011. IEEE. doi: 10.1109/CLOUD.2011.42. 1.1

[197] Louis Rustenholz, Maximiliano Klemen, Miguel Á. Carreira-Perpiñán, and Pedro López-García. A machine learning-based approach for solving recurrence relations and its use in cost analysis of logic programs. *Theory and Practice of Logic Programming*, 24(6):1163–1207, 2024. doi: 10.1017/S1471068424000413. 5.2.2, 7.7.1

[198] Louis Rustenholz, Pedro López-García, José F. Morales, and Manuel V. Hermenegildo. An order theory framework of recurrence equations for static cost analysis – dynamic inference of non-linear inequality invariants. In *Static Analysis: 31st International Symposium, SAS 2024, Pasadena, CA, USA, October 20–22, 2024, Proceedings*, pages

352–385, Berlin, Heidelberg, 2024. Springer-Verlag. ISBN 978-3-031-74775-5. doi: 10.1007/978-3-031-74776-2_14. URL https://doi.org/10.1007/978-3-031-74776-2_14. 5.2.2, 7.7.1

[199] Amr Sabry and Matthias Felleisen. Reasoning about programs in continuation-passing style. In *Proceedings of the 1992 ACM Conference on LISP and Functional Programming*, LFP '92, pages 288–298, New York, NY, USA, 1992. Association for Computing Machinery. ISBN 0897914813. doi: 10.1145/141471.141563. URL https://doi.org/10.1145/141471.141563. 1

[200] Peter Sanders and Rudolf Fleischer. Asymptotic Complexity from Experiments? A Case Study for Randomized Algorithms. In Stefan Näher and Dorothea Wagner, editors, *International Workshop on Algorithm Engineering*, volume 1982 of *Lecture Notes in Computer Science*, pages 135–146, Berlin, 2001. Springer. doi: 10.1007/3-540-44691-5_12. 5.2.2

[201] Alejandro Serrano, Pedro López-García, and Manuel V Hermenegildo. Resource usage analysis of logic programs via abstract interpretation using sized types. *Theory and Practice of Logic Programming*, 14(4-5):739–754, 2014. 5.2.1

[202] Alejandro Serrano Mena, Pedro López García, Francisco Bueno Carrillo, and Manuel V. Hermenegildo. Sized type analysis for logic programs (technical communication). *Theory and Practice of Logic Programming*, 13(4-5 (S):1–15, July 2013. ISSN 1471-0684. URL http://journals.cambridge.org/action/displayIssue?jid=TLP&volumeId=13&seriesId=0&issueId=4-5. Unpublished. 5.2.1

[203] Zhiming Shen, Sethuraman Subbiah, Xiaohui Gu, and John Wilkes. CloudScale: Elastic resource scaling for multi-tenant cloud systems. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, SOCC '11, pages 1–14, New York, NY, USA, October 2011. Association for Computing Machinery. ISBN 978-1-4503-0976-9. doi: 10.1145/2038916.2038921. 1.1

[204] Jeremy Siek and Walid Taha. Gradual typing for functional languages. 01 2006. 7.7.3, 7.7.3

[205] Hugo Simões, Pedro Vasconcelos, Mário Florido, Steffen Jost, and Kevin Hammond. Automatic amortised analysis of dynamic memory allocation for lazy functional programs. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming*, ICFP '12, pages 165–176, New York, NY, USA, 2012. Association for Computing Machinery. ISBN 9781450310543. doi: 10.1145/2364527.2364575. URL https://doi.org/10.1145/2364527.2364575. 5.2.1

[206] Moritz Sinn, Florian Zuleger, and Helmut Veith. A Simple and Scalable Static Analysis for Bound Analysis and Amortized Complexity Analysis. In Armin Biere and Roderick Bloem, editors, *Proc. 26th International Conference on Computer Aided Verification*, volume 8559 of *Lecture Notes in Computer Science*, pages 745–761, Cham, 2014. Springer. doi: 10.1007/978-3-319-08867-9_50. 1.1, 5.2.1

[207] Michael. Sipser. *Introduction to the theory of computation*. Cengage Learning, Boston, MA, 3rd ed. edition, 2013. ISBN 113318779X. 6.1.1, 6.1.2, 6.1

[208] Daniel D. Sleator and Robert E. Tarjan. Self-adjusting binary trees. In *Proc. 15th Annual*

ACM Symposium on Theory of Computing, pages 235–245, New York, NY, USA, December 1983. ACM. ISBN 978-0-89791-099-6. doi: 10.1145/800061.808752. 4.1, 5.2.1, 8.5.3

[209] Daniel D. Sleator and Robert E. Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28(2):202–208, February 1985. ISSN 0001-0782. doi: 10.1145/2786.2793. 4.1, 5.2.1

[210] Cees Slot and Peter van Emde Boas. The problem of space invariance for sequential machines. *Information and Computation*, 77(2):93–122, 1988. ISSN 0890-5401. doi: https://doi.org/10.1016/0890-5401(88)90052-1. URL https://www.sciencedirect.com/science/article/pii/0890540188900521. 5.1, 1

[211] M. B. Smyth and G. D. Plotkin. The category-theoretic solution of recursive domain equations. *SIAM Journal on Computing*, 11(4):761–783, 1982. doi: 10.1137/0211062. URL https://doi.org/10.1137/0211062. 8.2.3

[212] Thomas Streicher. *Domain-Theoretic Foundations of Functional Programming*. WORLD SCIENTIFIC, 2006. doi: 10.1142/6284. URL https://www.worldscientific.com/doi/abs/10.1142/6284. 8.2.2, 8.2.2, 8.2.2, 8.2.2, 8.2.3

[213] Robert E. Tarjan. Amortized computational complexity. *SIAM Journal on Matrix Analysis and Applications*, 6(2):306–13, April 1985. doi: 10.1137/0606031. 5.2.1

[214] Robert E. Tarjan and Jan van Leeuwen. Worst-case analysis of set union algorithms. *J. ACM*, 31(2):245–281, March 1984. ISSN 0004-5411. doi: 10.1145/62.2160. URL https://doi.org/10.1145/62.2160. 8.6

[215] Robert Endre Tarjan. Efficiency of a good but not linear set union algorithm. *J. ACM*, 22(2):215–225, April 1975. ISSN 0004-5411. doi: 10.1145/321879.321884. URL https://doi.org/10.1145/321879.321884. 8.6

[216] Jan-Willem van de Meent, Brooks Paige, Hongseok Yang, and Frank Wood. An introduction to probabilistic programming, 2021. URL https://arxiv.org/abs/1809.10756. 7.1.1, 1

[217] Pedro Vasconcelos, Steffen Jost, Mário Florido, and Kevin Hammond. Type-based allocation analysis for co-recursion in lazy functional languages. In Jan Vitek, editor, *Programming Languages and Systems*, pages 787–811, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg. ISBN 978-3-662-46669-8. 5.2.1

[218] Pedro B. Vasconcelos. *Space Cost Analysis Using Sized Types*. PhD thesis, University of St. Andrews, 2008. 1.1, 3.1, 5.2.3, 10.2

[219] Di Wang and Jan Hoffmann. Type-guided worst-case input generation. *Proceedings of the ACM on Programming Languages*, 3(POPL):13:1–13:30, January 2019. doi: 10.1145/3290326. 1.1, 5.3.1, 9.1, 9.4.1, 9.4.1, 9.4.1

[220] Di Wang, David M. Kahn, and Jan Hoffmann. Raising expectations: Automating expected cost analysis with types. *Proc. ACM Program. Lang.*, 4(ICFP), August 2020. doi: 10.1145/3408992. 5.2.1, 10.2

[221] Jinyi Wang, Yican Sun, Hongfei Fu, Mingzhang Huang, Amir Kafshdar Goharshady, and Krishnendu Chatterjee. Concentration-bound analysis for probabilistic programs and

probabilistic recurrence relations, 2020. URL https://arxiv.org/abs/2008.00425. 9.1

[222] Peng Wang, Di Wang, and Adam Chlipala. Timl: a functional language for practical complexity analysis with invariants. *Proc. ACM Program. Lang.*, 1(OOPSLA), oct 2017. doi: 10.1145/3133903. URL https://doi.org/10.1145/3133903. 1.1, 1.2.2, 1.2.3, 3, 2.2, 5.2.3, 5.2.3, 8.6, 8.7, B.56, B.57

[223] Ben Wegbreit. Mechanical program analysis. *Communications of the ACM*, 18(9):528–539, September 1975. ISSN 0001-0782. doi: 10.1145/361002.361016. 1.1, 4.1, 5.2.1

[224] Jiayi Wei, Jia Chen, Yu Feng, Kostas Ferles, and Isil Dillig. Singularity: pattern fuzzing for worst case complexity. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2018, pages 213–223, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450355735. doi: 10.1145/3236024.3236039. URL https://doi.org/10.1145/3236024.3236039. 5.3.1, 9.1, 9.2, 9.3.2, 9.3.2

[225] Jiayi Wei, Maruth Goyal, Greg Durrett, and Isil Dillig. Lambdanet: Probabilistic type inference using graph neural networks, 2020. URL https://arxiv.org/abs/2005.02161. 7.7.3

[226] Cheng Wen, Haijun Wang, Yuekang Li, Shengchao Qin, Yang Liu, Zhiwu Xu, Hongxu Chen, Xiaofei Xie, Geguang Pu, and Ting Liu. Memlock: memory usage guided fuzzing. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, pages 765–777, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450371216. doi: 10.1145/3377811.3380396. URL https://doi.org/10.1145/3377811.3380396. 1.1, 5.3.1, 9.1

[227] Brandon T. Willard and Rémi Louf. Efficient guided generation for large language models, 2023. URL https://arxiv.org/abs/2307.09702. 10.2

[228] Glynn Winskel. *The formal semantics of programming languages: an introduction*. MIT Press, Cambridge, MA, USA, 1993. ISBN 0262231697. 8.2.3, 8.2.3, 8.2.3

[229] Frank Wood, Jan Willem Meent, and Vikash Mansinghka. A New Approach to Probabilistic Programming Inference. In Samuel Kaski and Jukka Corander, editors, *Proceedings of the Seventeenth International Conference on Artificial Intelligence and Statistics*, volume 33 of *Proceedings of Machine Learning Research*, pages 1024–1032, Reykjavik, Iceland, 22–25 Apr 2014. PMLR. URL https://proceedings.mlr.press/v33/wood14.html. 7.1.2

[230] Hongjun Wu and Di Wang. Worst-case analysis is maximum-a-posteriori estimation, 2023. URL https://arxiv.org/abs/2310.09774. 5.3.1

[231] Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '99, pages 214–227, New York, NY, USA, 1999. Association for Computing Machinery. ISBN 1581130953. doi: 10.1145/292540.292560. URL https://doi.org/10.1145/292540.292560. 5.2.1

[232] Akihisa Yamada and Jérémy Dubut. Formalizing Results on Directed Sets in Isabelle/HOL (Proof Pearl). In Adam Naumowicz and René Thiemann, editors, *14th International*

Conference on Interactive Theorem Proving (ITP 2023), volume 268 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 34:1–34:13, Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. ISBN 978-3-95977-284-6. doi: 10.4230/LIPIcs.ITP.2023.34. URL https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ITP.2023.34. 4

[233] Michal Zalewski. American fuzzy lop (afl), 2017. URL https://lcamtuf.coredump.cx/afl/. 9.3.2

[234] Dmitrijs Zaparanuks and Matthias Hauswirth. Algorithmic profiling. In *Proc. 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 67–76, New York, NY, USA, 2012. ACM. doi: 10.1145/2254064.2254074. 1.1, 1.2.2, 1.2.2, 2.1, 5.2.2, 6.3.3, 7.3, 7.3.2, 9.1

[235] Florian Zuleger, Sumit Gulwani, Moritz Sinn, and Helmut Veith. Bound Analysis of Imperative Programs with the Size-Change Abstraction. In Eran Yahav, editor, *Proc. 18th International Static Analysis Symposium*, volume 6887 of *Lecture Notes in Computer Science*, pages 280–297, Berlin, 2011. Springer. doi: 10.1007/978-3-642-23702-7_22. 1.1, 5.2.1