

A (CO)ALGEBRAIC APPROACH TO PROGRAMMING AND VERIFYING COMPUTER NETWORKS

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Steffen Juilf Smolka

December 2019

© 2019 Steffen Juilf Smolka
ALL RIGHTS RESERVED

A (CO)ALGEBRAIC APPROACH TO PROGRAMMING AND VERIFYING COMPUTER
NETWORKS

Steffen Juilf Smolka, Ph.D.

Cornell University 2019

As computer networks have grown into some of the most complex and critical computing systems today, the means of configuring them have not kept up: they remain manual, low-level, and ad-hoc. This makes network operations expensive and network outages due to misconfigurations commonplace. The thesis of this dissertation is that high-level programming languages and formal methods can make network configuration dramatically easier and more reliable.

The dissertation consists of three parts. In the first part, we develop algorithms for compiling a network programming language with high-level abstractions to low-level network configurations, and introduce a symbolic data structure that makes compilation efficient in practice. In the second part, we develop foundations for a probabilistic network programming language using measure and domain theory, showing that continuity can be exploited to approximate (statistics of) packet distributions algorithmically. Based on this foundation and the theory of Markov chains, we then design a network verification tool that can reason about fault-tolerance and other probabilistic properties, scaling to data-center-size networks. In the third part, we introduce a general-purpose (co)algebraic framework for designing and reasoning about programming languages, and show that it permits an almost linear-time decision procedure for program equivalence. We hope that the framework will serve as a foundation for efficient verification tools, for networks and beyond, in the future.

Biographical Sketch

Steffen Smolka grew up in Saarbrücken, Germany, where he graduated from Gymnasium am Schloss in 2010. He spent 10th grade as a foreign exchange student in Ashland, Wisconsin, developing a passion for cultural exchange. Steffen went on to study computer science at Technical University Munich in 2010, graduating with a Bachelor of Science with a minor in mathematics in 2013. Combining his passions for cultural exchange and math and sciences, Steffen enrolled as a doctoral student in computer science at Cornell University in 2013, graduating in 2019. During this period, he spent summers with Dimitrios Vytiniotis at Microsoft Research in Cambridge, UK; at Barefoot Networks and Stanford University in Palo Alto, California; with Alexandra Silva at University College London; and with the Google Cloud team in Sunnyvale, California.

To my parents, for teaching me the important things in life.
And to my brother, for being my role model and friend from day one.

Acknowledgements

The road towards a Ph.D. was wonderful, but undeniably long and at times tough. I owe a grate deal to the people who supported me along the way.

First and foremost, I am grateful to my adviser and mentor Nate Foster. Nate and I share a passion for bringing theory to bear on real-world problems. This has been a major theme throughout my Ph.D. Nate also introduced me to NetKAT (in our very first research meeting), a second major theme throughout my Ph.D. Big challenges come with occasional struggles and self-doubts; it was invaluable to have an adviser who supported, nurtured, and pushed me, but most of all firmly believed in me. I entered countless meetings with a sense of defeat after having hit a roadblock, but left full of energy and motivation after talking to Nate.

Dexter Kozen served on my committee and became somewhat of an inofficial second adviser. Dexter has an incredible sense for technical elegance; I will forever strive to inherit it. I am grateful for always finding an open door when looking to discuss ideas or understand unfamiliar math; and for always getting a quick reply to my countless emails, sent at all times of the day, asking questions about various corners of the mathematical universe.

I credit Bobby Kleinberg, the third member of my committee, for teaching my favorite class at Cornell: *Analysis of Algorithms*. I also blame him for (unwittingly) instilling occasional doubts in me about picking the right concentration. I am grateful for his feedback and advice along the way.

I was fortunate to work with outstanding collaborators over the years; without their contributions, the papers in this dissertation would not have been written: Spiridon Eliopoulos, Nate Foster, Arjun Guha, Justin Hsu, David Kahn, Tobias Kappé, Praveen Kumar, Dexter Kozen, and Alexandra Silva.

Dimitrios Vytiniotis hosted me for a summer at Microsoft Research in Cambridge, UK during my early days as a graduate student. Thank you for an incredibly fun, intense, and intellectually stimulating time, and many hours of hands-on work together.

I spent another wonderful summer with Alexandra Silva at UCL in London. Alexandra has been a collaborator, role model, and by now great friend. I am very grateful for multiple invitations to the Bellairs workshop in Barbados, where Alexandra and I worked on end on automata and decision procedures for various KATs. The workshops also allowed me to get to know many brilliant researchers on a more personal level, making me feel a lot more at home in the world of academia. Thank you to the Barbados Crew (and Prakash Panangaden in particular)!

Working with Justin Hsu during his postdoc was one of my favorite times at Cornell (and culminated in a proof with a seven-line subscript!). Thank you for the advice, discussions about \mathbb{C} ün coin, and for convincing me to finally give IPAs a real shot.

My time in Ithaca would have lacked color if it were not for the great friends I made along the way. Thank you to my first housemate Rahim Gulamaliyev, for getting me started in Collegetown; to Thodoris Lykouris, for braking my undergrad habit of quitting problem sets after solving 60%; to Rahmtin Rotabi, for being there for me when I needed a friend; to Mischa Olson and Aditya Vaidyanathan, for playing countless hours of Loopin' Louie with me; to Zoya Segelbacher, for teaching me yoga; and to Mystery Machine, the team who drinks together and wins together.

I am especially grateful to my housemates at Lake Street, who have been like a family to me. Thank you to Jonathan DiLorenzo, for enthusiastically supporting my

crazy, impulsive ideas and for discussing my research with me; to Molly Feldman, for being my favorite office mate and making the best Eggplant Parm; to Daniel Freund, for being my German friend abroad; and to Amir Montazeri, for sharing my passion for Gennaro Contaldo. Thanks also to the former Lake Street inhabitants Sam Hopkins, Eoin O'Mahony, and Mark Reitblatt.

Marisa, you were there from day one. I will be forever grateful for your love and support, and for the beautiful times we spent together.

Finally, I thank my family; I would not be where I am without them. My dad deserves special thanks for feeding me with captivating functional-programming puzzles long before I ever imagined studying computer science; and my mum deserves special thanks for teaching me that the truly important things in life are not a matter of science.

Funding. My research was funded by the National Science Foundation under grants CNS-1111698, CCF-1421752, and CCF-1637532; by the Office of Naval Research under grant N00014-15-1-2177; and by a gift from InfoSys.

This dissertation was typeset in Bitsream Charter using \LaTeX 2 ϵ .

Table of Contents

1	Introduction	1
1.1	Overview of Approach	3
1.1.1	Approach 1: High-level Programming Languages	3
1.1.2	Approach 2: Automated Verification	5
1.1.3	Combined Approach	6
1.2	Contributions	7
1.3	Acknowledgments	7
I	Deterministic Networks	9
2	Compilation	11
2.1	Introduction	12
2.2	Overview	15
2.3	Local Compilation	22
2.4	Global Compilation	31
2.4.1	NetKAT Automata	33
2.4.2	Local Program Generation	37
2.5	Virtual Compilation	40
2.6	Evaluation	47
2.7	Related Work	54
2.8	Conclusion	55
II	Probabilistic Networks	57
3	Semantic Foundations	59
3.1	Introduction	59
3.2	Overview	63
3.3	Preliminaries	69
3.4	ProbNetKAT	75
3.5	Cantor Meets Scott	82
3.6	A DCPO on Markov Kernels	92
3.7	Continuity and Semantics of Iteration	93

3.8	Approximation	96
3.9	Implementation and Case Studies	98
3.10	Related Work	103
3.11	Conclusion	105
4	Scalable Verification	107
4.1	Introduction	108
4.2	Overview	110
4.3	ProbNetKAT Syntax and Semantics	115
4.4	Computing Stochastic Matrices	119
4.5	Implementation	125
4.5.1	Native Backend	126
4.5.2	PRISM Backend	128
4.6	Evaluation	129
4.7	Case Study: Data Center Fault-Tolerance	135
4.8	Related Work	140
4.9	Conclusion	142
III	A Family of Programming Languages	143
5	Guarded Kleene Algebra with Tests	145
5.1	Introduction	145
5.2	Overview: An Abstract Programming Language	148
5.2.1	Syntax	149
5.2.2	Semantics: Language Model	149
5.2.3	Relational Model	152
5.2.4	Probabilistic Model	154
5.3	Axiomatization	157
5.3.1	Some Simple Axioms	157
5.3.2	A Fundamental Theorem	160
5.3.3	Derivable Facts	162
5.3.4	A Limited Form of Completeness	164
5.4	Automaton Model and Kleene Theorem	167
5.4.1	Automata and Languages	168
5.4.2	Expressions to Automata: a Thompson Construction	169
5.4.3	Automata to Expressions: Solving Linear Systems	172
5.5	Decision Procedure	176
5.5.1	Normal Coalgebras	177
5.5.2	Bisimilarity for Normal Coalgebras	179
5.5.3	Deciding Equivalence	184
5.6	Completeness for the Language Model	186
5.6.1	Systems of Left-Affine Equations	187
5.6.2	General Completeness	189

5.7	Related Work	190
5.8	Conclusions and Future Directions	192
IV	Conclusion	195
6	Conclusion	197
6.1	Thoughts on Practical Impact	197
6.2	Future Directions	198
	Bibliography	201
V	Appendix	223
A	Appendix to Chapter 3	225
A.1	$(\mathcal{M}(2^H), \sqsubseteq)$ is not a Semilattice	225
A.2	Non-Algebraicity	226
A.3	Cantor Meets Scott	227
A.4	A DCPO on Markov Kernels	227
A.5	Continuity of Kernels and Program Operators and a Least-Fixpoint Characterization of Iteration	231
A.5.1	Products and Integration	231
A.5.2	Continuous Operations on Measures	237
A.5.3	Continuous Kernels	239
A.5.4	Continuous Operations on Kernels	243
A.5.5	Iteration as Least Fixpoint	246
A.6	Approximation and Discrete Measures	251
B	Appendix to Chapter 4	255
B.1	ProbNetKAT Denotational Semantics for History-free Fragment	255
B.1.1	The CPO $(\mathcal{D}(2^{P^k}), \sqsubseteq)$	258
B.2	Omitted Proofs	259
B.3	Background on Datacenter Topologies	266
C	Appendix to Chapter 5	269
C.1	Omitted Proofs	269
C.2	Generalized Guarded Union	299

Chapter 1

Introduction

“In the beginning, there was chaos.”

—Hesiod, Theogony

As computer networks have grown into some of the most complex and critical computing systems today, the means of configuring them have not kept up: they remain embarrassingly manual, low-level, and ad-hoc. This makes network operations expensive and network outages due to misconfigurations commonplace. Answering even the most basic questions about network behavior, such as, *“Why did my packet not get delivered?”*, requires tedious manual reasoning and investigating, or is right out impossible. At a high level, there are three main contributors to this issue.

The visibility problem. Modern networks are essentially black boxes, permitting very limited visibility into the processing of individual packets. While software developers can step through the execution of their code line by line with debugging tools, there are no comparable tools for network engineers. Even determining basic information, such as the path that a packet takes through a network, can be challenging. Network engineers must resort to low-level tools such as PING and TRACEROUTE, validating and invalidating hypotheses about network behavior through packet probes. Given the

limited observability—e.g., one may observe *that* a packet was dropped, but this leaves open *why* and *where* it was dropped—this process is time-consuming, error-prone, and often inconclusive.

The state space problem. The visibility problem is exacerbated by the vast size of the state space that network administrators must reason about. For example, there are 2^{160} different IPv4 headers, influencing how a packet gets processed by the network. Thus, even when the behavior of individual packets is understood, it is challenging to establish more general properties such as, “*All TCP packets from the internet can reach the web server*”, or, “*No packet from point A can reach point B*”. In particular, the number of packets is too large to reduce, through naive enumeration, a universal or existential property to multiple single-packet properties.

The semantic gap problem. Configuring a network requires translating network-wide, high-level objectives—*i.e.*, the network operator’s intent—to device-local, hardware-level configurations. For example, the intent “*Hosts A and B can communicate*” may necessitate assigning IP addresses and configuring routing tables across multiple devices. Thus, configuring a network requires bridging a large semantic gap: from global to local, and from high-level to low-level. Today this is typically done by hand, which is slow, tedious, and error-prone. This semantic gap also makes it hard to reason about global network behavior, since it arises—possibly through complex interactions—from the local behavior of multiple devices and their low-level configurations.

The thesis of this dissertation is that high-level programming languages and formal methods can address these problems, making network configuration dramatically easier and more reliable.

1.1 Overview of Approach

We propose two largely orthogonal approaches to making the configuration of networks easier and more reliable. The approaches synergize when applied in conjunction, but may also be applied independently. The first approach focuses on closing the semantics gap, whereas the second approach focuses on solving the visibility and state space problems.

1.1.1 Approach 1: High-level Programming Languages

We propose using domain-specific, high-level programming languages to configure networks. This has a number of advantages over the low-level approach to configuration that is pervasive today; in particular, many of the well-known advantages from general-purpose programming apply also to networking.

Our primary motivation is that high-level languages can offer abstractions that are significantly closer to the mental models of network operators than the hardware-level interfaces exposed by network devices. Thus, the semantic gap between a network operator's intent and a program encoding this intent is reduced, and can be bridged by the operator with greater efficiency and accuracy, making configuration changes less costly and error-prone. Some secondary advantages include:

- **Modularity:** High-level languages can offer mechanisms that support the separation of concerns: a program can be structured as a collection of largely independent components that are then composed to form the overall configuration. This reduces complexity, facilitates code reuse, and allows developing separate components in parallel.
- **Portability:** High-level languages can abstract hardware-level details, making it possible to use the same program to configure, *e.g.*, hardware from different vendors. They can also offer mechanisms to abstract the physical network topology,

making it possible to evolve the network configuration largely independently from the network topology.

- **Safety:** High-level languages can rule out large classes of misconfigurations simply by making it impossible to express these undesirable configurations in the language. Mechanisms such as type-systems can be employed to rule out additional errors.

By reducing the semantic gap and introducing natural abstractions, high-level languages not only make it easier to *program* the network, but also to *reason* about the network's behavior, since reasoning can now take place at the language-level instead of the hardware-level.

Suitable network programming languages have already been proposed in the literature [6, 43, 116, 120, 174]. However, the approach depends crucially on compilation algorithms that can implement the high-level abstractions exposed by such languages. To be practical, the algorithms must meet additional requirements:

- **Efficiency:** Network devices tend to require frequent reconfigurations. Thus compilation must scale to real-world networks with thousands of switches in seconds.
- **Frugality:** Network devices have tight resource limitations. Thus compilation must produce code that is economical in its use of hardware resources.
- **Soundness:** The compilation algorithms should bridge the semantic gap reliably without introducing errors. At a minimum, one should be able to state formal correctness properties; ideally, the implementation should be formally verified.

We develop a compiler meeting these requirements in Chapter 2, thus making a key contribution towards making the high-level language approach to network configuration viable.

1.1.2 Approach 2: Automated Verification

Our second proposal aims at addressing the visibility and state space problems, which make it hard to observe and reason about network behavior.

It is natural to address the visibility problem—the fact that today’s networking devices are essentially black boxes—by proposing a different hardware design with improved visibility, for example through the logging of processing steps. In fact, such an approach has become viable [80] thanks to recent hardware advances [122]. However, visibility alone is arguably not enough; state-of-the-art switches forward over a billion packets per second [122], meaning that network misbehavior can cause serious harm before it has been observed, analyzed, and debugged, even with perfect visibility. What is needed is foresight, not just sight: bugs should be ruled out before they ever manifest.

We propose using automated verification tools to rule out network misconfigurations. The approach consists of two parts. First, the verification tools builds a mathematical model of the network (based on its configuration). Second, the verification tool analyzes the model and establishes desired properties (or reports violations of these properties). The mathematical network model solves the visibility problem: the model can be used to predict network behavior accurately. Verification solves the state space problem: formal proofs ensure that the properties of interest hold for *all* packets. Under the hood, this relies on symbolic techniques that can reason about the large state space efficiently.

Network verification has by now become an active research area with many recent advances [14, 45, 77, 79, 136, 178]. However, prior work ignores probabilistic network features such as random link failures or randomized and fault-tolerant routing schemes, assuming instead that network behavior is deterministic; and it can only establish qualitative properties.

In Chapter 3, we develop the denotational semantics of a probabilistic network programming and verification language, a suitable framework for reasoning about proba-

bilistic network behavior. This addresses the visibility problem for probabilistic networks. In Chapter 4, we build on this foundation and develop a probabilistic verification tool that can reason about quantitative properties such as fault-tolerance, scaling to real-world networks with thousands of nodes. This addresses the state space problem.

1.1.3 Combined Approach

While the two approaches—configuring networks using high-level languages, and ruling out bugs using automated verification tools—can be applied independently, significant synergies can be obtained by combining them. NetKAT [6] is a system that follows this combined approach: the NetKAT language serves both as a programming language (Chapter 2) and as a verification language [6, 45]. Verification properties are encoded as equivalences between programs; such equivalences can then be established manually using NetKAT’s equational axioms, or algorithmically using NetKAT’s decision procedure.

Such an approach requires the codesign of the language and its associated reasoning framework. In the case of NetKAT, this was accomplished by building on Kleene Algebra with Tests (KAT), a (co)algebraic reasoning and programming framework whose metatheory has been carefully and extensively studied in the literature. KAT enjoys a sound and complete equational axiomatization, and a decision procedure for program equivalence based on automata and language models for KAT.

In Chapter 5, we study a variation on KAT called Guarded Kleene Algebra with Tests (GKAT) that addresses two issues arising when using KAT as a foundation for network programming languages. First, KAT is not a suitable foundation for probabilistic languages (as considered in Chapters 3 and 4) due to well-known issues that arise when combining non-deterministic and probabilistic choice. Second, KAT’s decision procedure is PSPACE hard. We show that GKAT can model probabilistic languages and permits an almost linear-time decision procedure. We hope that GKAT will serve as an efficient

foundation for future language/verification codesigns, for networks and beyond.

1.2 Contributions

In summary, this dissertation makes the following contributions:

- We propose novel algorithms for compiling a high-level network programming language to low-level forwarding tables that can be installed on programmable switches, and introduce a symbolic data structure that makes compilation efficient in practice (Chapter 2).
- We develop the denotational semantics of a probabilistic network programming and verification language, show that it is amenable to monotone approximation, and prototype a simple verification tool with formal convergence guarantees based on this foundation (Chapter 3).
- We build a scalable verification tool for probabilistic networks that can reason about probabilistic properties such as fault-tolerance, scaling to large, real-world networks (Chapter 4).
- We develop a (co)algebraic verification framework with a linear-time decision procedure and a sound and complete axiomatization, and show that it soundly models relational and probabilistic programming languages (Chapter 5).

1.3 Acknowledgments

This dissertation includes contributions by several coauthors and is based on the following publications and preprints:

- **Chapter 2:** Steffen Smolka, Spiros Eliopoulos, Nate Foster, and Arjun Guha. *A Fast Compiler for NetKAT*. In ICFP 2015. <https://doi.org/10.1145/2784731.2784761>.

- **Chapter 3 and Appendix A:** Steffen Smolka, Praveen Kumar, Nate Foster, Dexter Kozen, and Alexandra Silva. *Cantor Meets Scott: Semantic Foundations for Probabilistic Networks*. In POPL 2017. <https://doi.org/10.1145/3009837.3009843>.
- **Chapter 4 and Appendix B:** Steffen Smolka, Praveen Kumar, David M. Kahn, Nate Foster, Justin Hsu, Dexter Kozen, and Alexandra Silva. *Scalable Verification of Probabilistic Networks*. In PLDI 2019. <https://doi.org/10.1145/3314221.3314639>.
- **Chapter 5 and Appendix C:** Steffen Smolka, Nate Foster, Justin Hsu, Tobias Kappé, Dexter Kozen, and Alexandra Silva. *Guarded Kleene Algebra with Tests: Verification of Uninterpreted Programs in Nearly Linear Time*. In arXiv:1907.05920. <https://arxiv.org/abs/1907.05920>.

The paper “A Fast Compiler for NetKAT” was named a 2016 *SIGPLAN Research Highlight*.

Part I

Deterministic Networks

Chapter 2

Compilation

“Modularity based on abstraction is the way things get done.”

—Barbara Liskov

High-level programming languages play a key role in a growing number of networking platforms, streamlining application development and enabling precise formal reasoning about network behavior. Unfortunately, current compilers only handle “local” programs that specify behavior in terms of hop-by-hop forwarding behavior, or modest extensions such as simple paths. To encode richer “global” behaviors, programmers must add extra state—something that is tricky to get right and makes programs harder to write and maintain. Making matters worse, existing compilers can take tens of minutes to generate the forwarding state for the network, even on relatively small inputs. This forces programmers to waste time working around performance issues or even revert to using hardware-level APIs.

This chapter presents a new compiler for the NetKAT language that handles rich features including regular paths and virtual networks, and yet is several orders of magnitude faster than previous compilers. The compiler uses symbolic automata to calculate the extra state needed to implement “global” programs, and an intermediate representation based on binary decision diagrams to dramatically improve performance. We describe

the design and implementation of three essential compiler stages: from virtual programs (which specify behavior in terms of virtual topologies) to global programs (which specify network-wide behavior in terms of physical topologies), from global programs to local programs (which specify behavior in terms of single-switch behavior), and from local programs to hardware-level forwarding tables. We present results from experiments on real-world benchmarks that quantify performance in terms of compilation time and forwarding table size.

2.1 Introduction

High-level languages are playing a key role in a growing number of networking platforms being developed in academia and industry. There are many examples: VMware uses nlog, a declarative language based on Datalog, to implement network virtualization [85]; SDX uses Pyretic to combine programs provided by different participants at Internet exchange points [58, 116]; PANE uses NetCore to allow end-hosts to participate in network management decisions [40, 115]; Flowlog offers tierless abstractions based on Datalog [120]; Maple allows packet-processing functions to be specified directly in Haskell or Java [174]; OpenDaylight’s group-based policies describe the state of the network in terms of application-level connectivity requirements [140]; and ONOS provides an “intent framework” that encodes constraints on end-to-end paths [139].

The details of these languages differ, but they all offer abstractions that enable thinking about the behavior of a network in terms of high-level constructs such as packet-processing functions rather than low-level switch configurations. To bridge the gap between these abstractions and the underlying hardware, the compilers for these languages map source programs into forwarding rules that can be installed in the hardware tables maintained by software-defined networking (SDN) switches.

Unfortunately, most compilers for SDN languages only handle “local” programs

in which the intended behavior of the network is specified in terms of hop-by-hop processing on individual switches. A few support richer features such as end-to-end paths and network virtualization [85, 139, 174], but to the best of our knowledge, no prior work has presented a complete description of the algorithms one would use to generate the forwarding state needed to implement these features. For example, although NetKAT includes primitives that can be used to succinctly specify global behaviors including regular paths, the existing compiler only handles a local fragment [6]. This means that programmers can only use a restricted subset that is strictly less expressive than the full language and must manually manage the state needed to implement network-wide paths, virtual networks, and other similar features.

Another limitation of current compilers is that they are based on algorithms that perform poorly at scale. For example, the NetCore, NetKAT, PANE, and Pyretic compilers use a simple translation to forwarding tables, where primitive constructs are mapped directly to small tables and other constructs are mapped to algebraic operators on forwarding tables. This approach quickly becomes impractical as the size of the generated tables can grow exponentially with the size of the program! This is a problem for platforms that rely on high-level languages to express control application logic, as a slow compiler can hinder the ability of the platform to effectively monitor and react to changing network state.

Indeed, to work around the performance issues in the current Pyretic compiler, the developers of SDX [58] extended the language in several ways, including adding a new low-cost composition operator that implements the disjoint union of packet-processing functions. The idea was that the implementation of the disjoint union operator could use a linear algorithm that simply concatenates the forwarding tables for each function rather than using the usual quadratic algorithm that does an all-pairs intersection between the entries in each table. However, even with this and other optimizations, the Pyretic

compiler still took tens of minutes to generate the forwarding state for inputs of modest size.

Our approach. This chapter presents a new compiler pipeline for NetKAT that handles local programs executing on a single switch, global programs that utilize the full expressive power of the language, and even programs written against virtual topologies. The algorithms that make up this pipeline are orders of magnitude faster than previous approaches—e.g., our system takes two seconds to compile the largest SDX benchmarks, versus several minutes in Pyretic, and other benchmarks demonstrate that our compiler is able to handle large inputs far beyond the scope of its competitors.

These results stem from a few key insights. First, to compile local programs, we exploit a novel intermediate representation based on binary decision diagrams (BDDs). This representation avoids the combinatorial explosion inherent in approaches based on forwarding tables and allows our compiler to leverage well-known techniques for representing and transforming BDDs. Second, to compile global programs, we use a generalization of symbolic automata [137] to handle the difficult task of generating the state needed to correctly implement features such as regular forwarding paths. Third, to compile virtual programs, we exploit the additional expressiveness provided by the global compiler to translate programs on a virtual topology into programs on the underlying physical topology.

We have built a full working implementation of our compiler in OCaml, and designed optimizations that reduce compilation time and the size of the generated forwarding tables. These optimizations are based on general insights related to BDDs (sharing common structures, rewriting naive recursive algorithms using dynamic programming, using heuristic field orderings, etc.) as well as domain-specific insights specific to SDN (algebraic optimization of NetKAT programs, per-switch specialization, etc.). To evaluate the performance of our compiler, we present results from experiments run on a variety

of benchmarks. These experiments demonstrate that our compiler provides improved performance, scales to networks with tens of thousands of switches, and easily handles complex features such as virtualization.

Overall, this chapter makes the following contributions:

- We present the first complete compiler pipeline for NetKAT that translates local, global, and virtual programs into forwarding tables for SDN switches.
- We develop a generalization of BDDs and show how to implement a local SDN compiler using this data structure as an intermediate representation.
- We describe compilation algorithms for virtual and global programs based on graph algorithms and symbolic automata.
- We discuss an implementation in OCaml and develop optimizations that reduce running time and the size of the generated forwarding tables.
- We conduct experiments that show dramatic improvements over other compilers on a collection of benchmarks and case studies.

The next section briefly reviews the NetKAT language and discusses some challenges related to compiling SDN programs, to set the stage for the results described in the following sections.

2.2 Overview

NetKAT is a domain-specific language for specifying and reasoning about networks [6, 45]. It offers primitives for matching and modifying packet headers, as well combinators such as union and sequential composition that merge smaller programs into larger ones. NetKAT is based on a solid mathematical foundation, Kleene Algebra with Tests (KAT) [90], and comes equipped with an equational reasoning system that can be used to automatically verify many properties of programs [45].

Syntax

Naturals	$n ::= 0 \mid 1 \mid 2 \mid \dots$	
Fields	$f ::= f_1 \mid \dots \mid f_k$	
Packets	$\pi ::= \{f_1 = n_1, \dots, f_k = n_k\}$	
Histories	$h ::= \langle \pi \rangle \mid \pi :: h$	
Predicates	$t, u ::= \text{skip}$	<i>Identity</i>
	drop	<i>Drop</i>
	$f = n$	<i>Test</i>
	$t + u$	<i>Disjunction</i>
	$t \cdot u$	<i>Conjunction</i>
	$\neg t$	<i>Negation</i>
Programs	$p, q ::= t$	<i>Filter</i>
	$f \leftarrow n$	<i>Modification</i>
	$p + q$	<i>Union</i>
	$p \cdot q$	<i>Sequencing</i>
	p^*	<i>Iteration</i>
	dup	<i>Duplication</i>

Semantics

$\llbracket p \rrbracket \in \mathbf{H} \rightarrow 2^{\mathbf{H}}$
$\llbracket \text{skip} \rrbracket(h) := \{h\}$
$\llbracket \text{drop} \rrbracket(h) := \{\}$
$\llbracket f = n \rrbracket(\pi :: h) := \begin{cases} \{\pi :: h\} & \text{if } \pi.f = n \\ \{\} & \text{if } \pi.f \neq n \end{cases}$
$\llbracket \neg t \rrbracket(h) := \{h\} \setminus \llbracket t \rrbracket(h)$
$\llbracket f \leftarrow n \rrbracket(\pi :: h) := \{\pi[f := n] :: h\}$
$\llbracket p + q \rrbracket(h) := \llbracket p \rrbracket(h) \cup \llbracket q \rrbracket(h)$
$\llbracket p \cdot q \rrbracket(h) := (\llbracket p \rrbracket \bullet \llbracket q \rrbracket)(h)$
$\llbracket p^* \rrbracket(h) := \bigcup_i F^i(h)$
where $F^0(h) := \{h\}$
and $F^{i+1}(h) := (\llbracket p \rrbracket \bullet F^i)(h)$
$\llbracket \text{dup} \rrbracket(\pi :: h) := \{\pi :: \pi :: h\}$

Figure 2.1: NetKAT syntax and semantics.

NetKAT enables programmers to think in terms of functions on packets histories, where a packet (π) is a record of fields and a history (h) is a non-empty list of packets. This is a dramatic departure from hardware-level APIs such as OpenFlow, which require thinking about low-level details such as forwarding table rules, matches, priorities, actions, timeouts, etc. NetKAT fields f include standard packet headers such as Ethernet source and destination addresses, VLAN tags, *etc.*, as well as special fields to indicate the port (pt) and switch (sw) where the packet is located in the network. For brevity, we use `src` and `dst` fields in examples, though our compiler implements all of the standard fields supported by OpenFlow [113].

NetKAT syntax and semantics. Formally, NetKAT is defined by the syntax and semantics given in Figure 2.1. Predicates t describe logical predicates on packets and include primitive tests $f = n$, which check whether field f is equal to n , as well as the standard collection of boolean operators. This exposition focuses on tests that match fields exactly,

although our implementation supports generalized tests, such as IP prefix matches. Programs p can be understood as packet-processing functions that consume a packet history and produce a set of packet histories. Filters t drop packets that do not satisfy t ; modifications $f \leftarrow n$ update the f field to n ; unions $p + q$ copy the input packet and process one copy using p , the other copy using q , and take the union of the results; sequences $p \cdot q$ process the input packet using p and then feed each output of p into q (the \bullet operator is Kleisli composition); iterations p^* behave like the union of p composed with itself zero or more times; and dups extend the trajectory recorded in the packet history by one hop.

Topology encoding. Readers who are familiar with Frenetic [43], Pyretic [116], or NetCore [115], will be familiar with the basic details of this functional packet-processing model. However, unlike these languages, NetKAT can also model the behavior of the entire network, including its topology. For example, a (unidirectional) link from port pt_1 on switch sw_1 to port pt_2 on switch sw_2 , can be encoded in NetKAT as follows:

$$\text{dup} \cdot \text{sw} = sw_1 \cdot \text{pt} = pt_1 \cdot \text{sw} \leftarrow sw_2 \cdot \text{pt} \leftarrow pt_2 \cdot \text{dup}$$

Applying this pattern, the entire topology can be encoded as a union of links. Throughout this chapter, we will use the shorthand $[sw_1:pt_1] \rightarrow [sw_2:pt_2]$ to indicate links, and assume that dup and modifications to the switch field occur only in links.

Local programs. Since NetKAT can encode both the network topology and the behavior of switches, a NetKAT program describes the end-to-end behavior of a network. One simple way to write NetKAT programs is to define predicates that describe where packets enter (in) and exit (out) the network, and interleave steps of processing on switches (p) and topology (t):

$$in \cdot (p \cdot t)^* \cdot p \cdot out$$

To execute the program, only p needs to be specified—the physical topology implements in , t , and out . Because no switch modifications or dups occur in p , it can be directly compiled to a collection of forwarding tables, one for each switch. Provided the physical topology is faithful to the encoding specified by in , t , and out , a network of switches populated with these forwarding tables will behave like the above program. We call such a switch program p a *local* program because it describes the behavior of the network in terms of hop-by-hop forwarding steps on individual switches.

Global programs. Because NetKAT is based on Kleene algebra, it includes regular expressions, which are a natural and expressive formalism for describing paths through a network. Ideally, programmers would be able to use regular expressions to construct forwarding paths directly, without having to worry about how those paths were implemented. For example, a programmer might write the following to forward packets from port 1 on switch sw_1 to port 1 on switch sw_2 , and from port 2 on sw_1 to port 2 on sw_2 , assuming a link connecting the two switches on port 3:

$$\begin{aligned} & \text{pt}=1 \cdot \text{pt}\leftarrow 3 \cdot [sw_1:3] \rightarrow [sw_2:3] \cdot \text{pt}\leftarrow 1 \\ & + \text{pt}=2 \cdot \text{pt}\leftarrow 3 \cdot [sw_1:3] \rightarrow [sw_2:3] \cdot \text{pt}\leftarrow 2 \end{aligned}$$

Note that this is *not* a local program, since is not written in the general form given above and instead combines switch processing and topology processing using a particular combination of union and sequential composition to describe a pair of overlapping forwarding paths. To express the same behavior as a local NetKAT program or in a language such as Pyretic, we would have to somehow write a single program that specifies the processing that should be done at each intermediate step. The challenge is that when sw_2 receives a packet from sw_1 , it needs to determine if that packet originated at port 1 or 2 of sw_1 , but this can't be done without extra information. For example, the compiler could add a tag to packets at sw_1 to track the original ingress and use this information to determine the processing at sw_2 . In general, the expressiveness of *global*

programs creates challenges for the compiler, which must generate explicit code to create and manipulate tags. These challenges have not been met in previous work on NetKAT or other SDN languages.

Virtual programs. Going a step further, NetKAT can also be used to specify behavior in terms of virtual topologies. To see why this is a useful abstraction, suppose that we wish to implement point-to-point connectivity between a given pair of hosts in a network with dozens of switches. One could write a global program that explicitly forwards along the path between these hosts. But this would be tedious for the programmer, since they would have to enumerate all of the intermediate switches along the path. A better approach is to express the program in terms of a virtual “big switch” topology whose ports are directly connected to the hosts, and where the relationship between ports in the virtual and physical networks is specified by an explicit mapping—*e.g.*, the top of Figure 2.2 depicts a big switch virtual topology. The desired functionality could then be specified using a simple local program that forwards in both directions between ports on the single virtual switch:

$$p := (pt=1 \cdot pt \leftarrow 2) + (pt=2 \cdot pt \leftarrow 1)$$

This one-switch virtual program is evidently much easier to write than a program that has to reference dozens of switches. In addition, the program is robust to changes in the underlying network. If the operator adds new switches to the network or removes switches for maintenance, the program remains valid and does not need to be rewritten. In fact, this program could be ported to a completely different physical network too, provided it is able to implement the same virtual topology.

Another feature of virtualization is that the physical-virtual mapping can limit access to certain switches, ports, and even packets that match certain predicates, providing a simple form of language-based isolation [59]. In this example, suppose the physical

network has hundreds of connected hosts. Yet, since the virtual-physical mapping only exposes two ports, the abstraction guarantees that the virtual program is isolated from the hosts connected to the other ports. Moreover, we can run several isolated virtual networks on the same physical network, *e.g.*, to provide different services to different customers in multi-tenant datacenters [85].

Of course, while virtual programs are a powerful abstraction, they create additional challenges for the compiler since it must generate physical paths that implement forwarding between virtual ports and also instrument programs with extra bookkeeping information to keep track of the locations of virtual packets traversing the physical network. Although virtualization has been extensively studied in the networking community [5, 23, 85, 116], no previous work fully describes how to compile virtual programs.

Compilation pipeline. This chapter presents new algorithms for compiling NetKAT that address the key challenges related to expressiveness and performance just discussed. Figure 2.2 depicts the overall architecture of our compiler, which is structured as a pipeline with several smaller stages: (i) a *virtual compiler* that takes as input a virtual program v , a virtual topology, and a mapping that specifies the relationship between the virtual and physical topology, and emits a global program that uses a fabric to transit between virtual ports using physical paths; (ii) a *global compiler* that takes an arbitrary NetKAT program g as input and emits a local program that has been instrumented with extra state to keep track of the execution of the global program; and a (iii) *local compiler* that takes a local program p as input and generates OpenFlow forwarding tables, using a generalization of binary decision diagrams as an intermediate representation. Overall, our compiler automatically generates the extra state needed to implement virtual and global programs, with performance that is dramatically faster than current SDN compilers.

These three stages are designed to work well together—*e.g.*, the fabric constructed

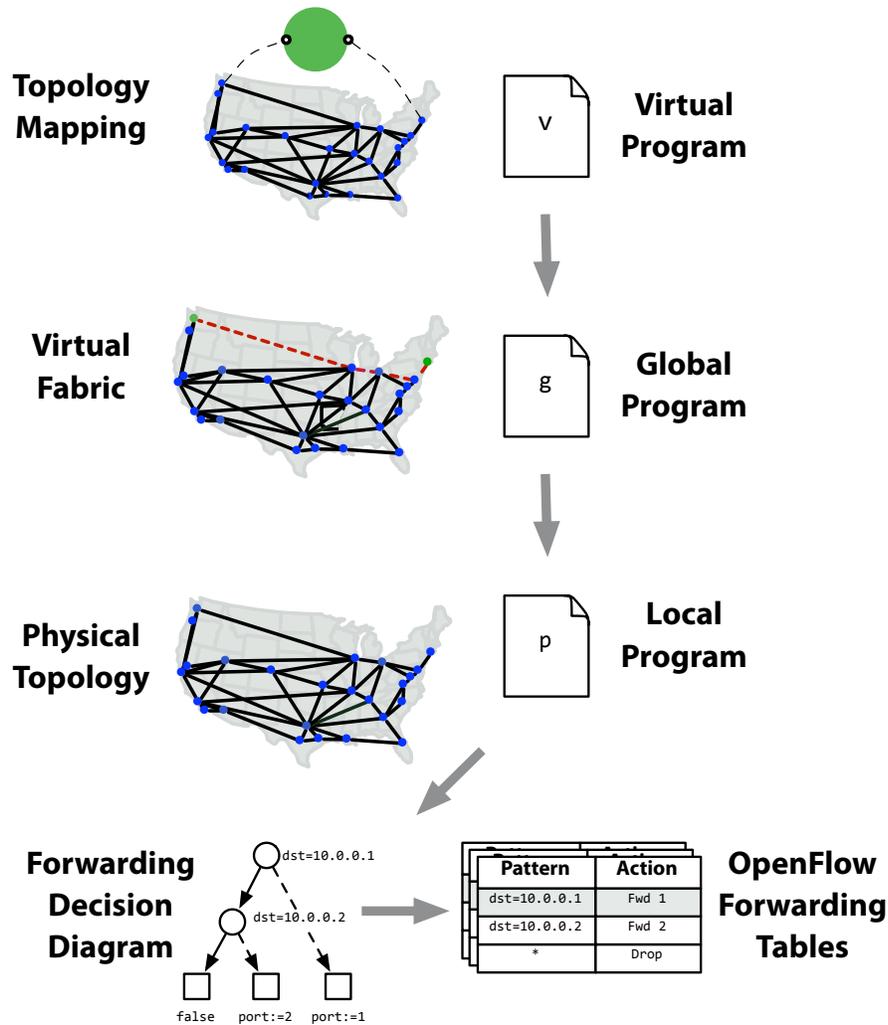


Figure 2.2: NetKAT compiler pipeline.

by the virtual compiler is expressed in terms of regular paths, which are translated to local programs by the global compiler, and the local and global compilers both use FDDs as an intermediate representation. However, the individual compiler stages can also be used independently. For example, the global compiler provides a general mechanism for compiling forwarding paths specified using regular expressions to SDN switches. We have also been working with the developers of Pyretic to improve performance by retargeting its backend to use our local compiler.

The next few sections present these stages in detail, starting with local compilation

and building up to global and virtual compilation.

2.3 Local Compilation

The foundation of our compiler pipeline is a translation that maps local NetKAT programs to OpenFlow forwarding tables. Recall that a local program describes the hop-by-hop behavior of individual switches—i.e. it does not contain `dup` or `switch` modifications.

Compilation via forwarding tables. A simple approach to compiling local programs is to define a translation that maps primitive constructs to forwarding tables and operators such as union and sequential composition to functions that implement the analogous operations on tables. For example, the current NetKAT compiler translates the modification `pt←2` to a forwarding table with a single rule that sets the port of all packets to 2 (Figure 2.3 (a)), while it translates the predicate `dst=A` to a flow table with two rules: the first matches packets where `dst=A` and leaves them unchanged and the second matches all other packets and drops them (Figure 2.3 (b)).

To compile the sequential composition of these programs, the compiler combines each row in the first table with the entire second table, retaining rules that could apply to packets produced by the row (Figure 2.3 (c)). In the example, the second table has a single rule that sends all packets to port 2. The first rule of the first table matches packets with destination A, thus the second table is transformed to only send packets with destination A to port 2. However, the second rule of the first table drops all packets, therefore no packets ever reach the second table from this rule.

To compile a union, the compiler computes the pairwise intersection of all patterns to account for packets that may match both tables. For example, in Figure 2.3 (d), the two sub-programs forward traffic to hosts A and B based on the `dst` header. These two sub-programs do not overlap with each other, which is why the table in the figure

Pattern	Action
*	pt←2

$pol_A := pt \leftarrow 2$

(a) An atomic modification

Pattern	Action
dst=A	skip
*	drop

$pol_B := dst = A$

(b) An atomic predicate

Pattern	Action
dst=A	pt←2
*	drop

$pol_B \cdot pol_A$

(c) Forwarding to a single host

Pattern	Action
dst=A	pt←1
dst=B	pt←2
*	drop

$pol_D := dst = A \cdot pt \leftarrow 1 + dst = B \cdot pt \leftarrow 2$

(d) Forwarding traffic to two hosts

Pattern	Action
dst=A	pt←3
proto=ssh	pt←3
*	drop

$pol_E := \left(\begin{array}{l} \text{proto=ssh} \\ + \text{dst=A} \end{array} \right) \cdot pt \leftarrow 3$

(e) Monitoring SSH traffic and traffic to host A

Figure 2.3: Compiling using forwarding tables.

appears simple. However, in general, the two programs may overlap. Consider compiling the union of the forwarding program, in Figure 2.3 (d) and the monitoring program in Figure 2.3 (e). The monitoring program sends SSH packets and packets with $dst=A$ to port 3. The intersection will need to consider all interactions between pairs of rules—an $\mathcal{O}(n^2)$ operation. Since a NetKAT program may be built out of several nested programs and compilation is quadratic at each step, we can easily get a tower of squares or exponential behavior.

Approaches based on flow tables are attractive for their simplicity, but they suffer several serious limitations. One issue is that tables are not an efficient way to represent packet-processing functions since each rule in a table can only encode positive tests on packet headers. In general, the compiler must emit sequences of prioritized rules to

encode operators such as negation or union. Moreover, the algorithms that implement these operators are worst-case quadratic, which can cause the compiler to become a bottleneck on large inputs. Another issue is that there are generally many equivalent ways to encode the same packet-processing function as a forwarding table. This means that a straightforward computation of fixed-points, as is needed to implement Kleene star, is not guaranteed to terminate.

Binary decision diagrams. To avoid these issues, our compiler is based on a novel representation of packet-forwarding functions using a generalization of *binary decision diagrams* (BDDs) [2, 21]. To briefly review, a BDD is a data structure that encodes a boolean function as a directed acyclic graph. The interior nodes encode boolean variables and have two outgoing edges: a true edge drawn as a solid line, and a false edge drawn as a dashed line. The leaf nodes encode constant values true or false. Given an assignment to the variables, we can evaluate the expression by following the appropriate edges in the graph. An *ordered* BDD imposes a total order in which the variables are visited. In general, the choice of variable-order can have a dramatic effect on the size of a BDD and hence on the run-time of BDD-manipulating operations. Picking an optimal variable-order is NP-hard, but efficient heuristics often work well in practice. A *reduced* BDD has no isomorphic subgraphs and every interior node has two distinct successors. A BDD can be reduced by repeatedly applying these two transformations:

- If two subgraphs are isomorphic, delete one by connecting its incoming edges to the isomorphic nodes in the other, thereby *sharing* a single copy of the subgraph.
- If both outgoing edges of an interior node lead to the same successor, eliminate the interior node by connecting its incoming edges directly to the common successor node.

Logically, an interior node can be thought of as representing an IF-THEN-ELSE expression.¹ For example, the expression:

$$(a ? (c ? 1 : (d ? 1 : 0)) : (b ? (c ? 1 : (d ? 1 : 0)) : 0))$$

represents a BDD for the boolean expression $(a \vee b) \wedge (c \vee d)$. This notation makes the logical structure of the BDD clear while abstracting away from the sharing in the underlying graph representation and is convenient for defining BDD-manipulating algorithms.

In principle, we could use BDDs to directly encode NetKAT programs as follows. We would treat packet headers as flat, n -bit vectors and encode NetKAT predicates as n -variable BDDs. Since NetKAT programs produce sets of packets, we could represent them in a relational style using BDDs with $2n$ variables. However, there are two issues with this representation:

- Typical NetKAT programs modify only a few headers and leave the rest unchanged. The BDD that represents such a program would have to encode the identity relation between most of its input-output variables. Encoding the identity relation with BDDs requires a linear amount of space, so even trivial programs, such as the identity program, would require large BDDs.
- The final step of compilation needs to produce a prioritized flow table. It is not clear how to efficiently translate BDDs that represent NetKAT programs as relations into tables that represent packet-processing functions. For example, a table of length one is sufficient to represent the identity program, but to generate this table from the BDD sketched above, several paths would have to be compressed into a single rule.

Forwarding Decision Diagrams. To encode NetKAT programs as decision diagrams, we introduce a modest generalization of BDDs called *forwarding decision diagrams*

¹We write conditionals as $(a ? b : c)$, in the style of the C ternary operator.

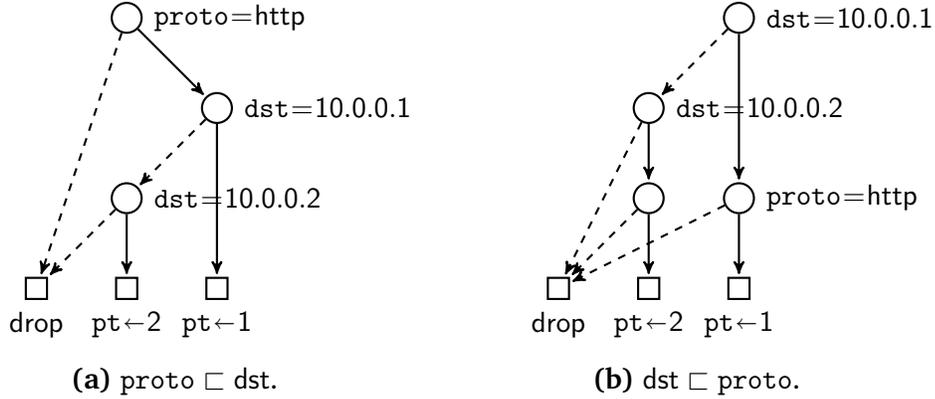


Figure 2.4: Two ordered FDDs for the same program.

(FDDs). An FDD differs from BDDs in two ways. First, interior nodes match header fields instead of individual bits, which means we need far fewer variables compared to a BDD to represent the same program. Our FDD implementation requires 12 variables (because OpenFlow supports 12 headers), but these headers span over 200 bits. Second, leaf nodes in an FDD directly encode packet modifications instead of boolean values. Hence, FDDs do not encode programs in a relational style.

Figures 2.4a and 2.4b show FDDs for a program that forwards HTTP packets to hosts 10.0.0.1 and 10.0.0.2 at ports 1 and 2 respectively. The diagrams have interior nodes that match on headers and leaf nodes corresponding to the actions used in the program.

To generalize ordered BDDs to FDDs, we assume orderings on fields and values, both written \sqsubset , and lift them to tests $f=n$ lexicographically:

$$f_1=n_1 \sqsubset f_2=n_2 := (f_1 \sqsubset f_2) \vee (f_1 = f_2 \wedge n_1 \sqsubset n_2)$$

We require that tests be arranged in ascending order from the root. For reduced FDDs, we stipulate that they must have no isomorphic subgraphs and that each interior node must have two unique successors, as with BDDs, and we also require that the FDD must not contain redundant tests and modifications. For example, if the test $\text{dst}=10.0.0.1$ is true, then $\text{dst}=10.0.0.2$ must be false. Accordingly, an FDD should not perform the latter test if the former succeeds. Similarly, because NetKAT’s union operator $(p + q)$ is associative,

Syntax

Booleans $b ::= \top \mid \perp$
 Contexts $\Gamma ::= \cdot \mid \Gamma, (f, n) : b$
 Actions $a ::= \{f_1 \leftarrow n_1, \dots, f_k \leftarrow n_k\}$
 Diagrams $d ::= \{a_1, \dots, a_k\}$ *constant*
 $\mid (f = n ? d_1 : d_2)$ *conditional*

Semantics

$\llbracket \{f_1 \leftarrow n_1, \dots, f_k \leftarrow n_k\} \rrbracket (\pi :: h) :=$
 $\quad \{\pi[f_1 := n_1] \cdots [f_k := n_k] :: h\}$
 $\llbracket \{a_1, \dots, a_k\} \rrbracket (\pi :: h) :=$
 $\quad \llbracket a_1 \rrbracket (\pi :: h) \cup \dots \cup \llbracket a_k \rrbracket (\pi :: h)$
 $\llbracket (f = n ? d_1 : d_2) \rrbracket (\pi :: h) :=$
 $\quad \begin{cases} \llbracket d_1 \rrbracket (\pi :: h) & \text{if } \pi.f = n \\ \llbracket d_2 \rrbracket (\pi :: h) & \text{if } \pi.f \neq n \end{cases}$

Well Formedness

$\boxed{\Gamma \sqsubset (f, n)} \quad \frac{}{\cdot \sqsubset (f, n)} \text{NIL}$
 $\frac{f' \sqsubset f}{\Gamma, (f', n') : b' \sqsubset (f, n)} \text{LT}$
 $\frac{f' = f \quad n' \sqsubset n}{\Gamma, (f', n') : \perp \sqsubset (f, n)} \text{EQ}$
 $\boxed{\Gamma \vdash d} \quad \frac{}{\Gamma \vdash \{a_1, \dots, a_k\}} \text{CONSTANT}$
 $\frac{\Gamma \sqsubset (f, n) \quad \Gamma, (f, n) : \top \vdash d_1 \quad \Gamma, (f, n) : \perp \vdash d_2}{\Gamma \vdash (f = n ? d_1 : d_2)} \text{CONDITIONAL}$

Figure 2.5: Forwarding decision diagrams: syntax, semantics, and well formedness.

commutative, and idempotent, to broadcast packets to both ports 1 and 2 we could either write $pt \leftarrow 1 + pt \leftarrow 2$ or $pt \leftarrow 2 + pt \leftarrow 1$. Likewise, repeated modifications to the same header are equivalent to just the final modification, and modifications to different headers commute. Hence, updating the `dst` header to 10.0.0.1 and then immediately re-updating it to 10.0.0.2 is the same as updating it to 10.0.0.2. In our implementation, we enforce the conditions for ordered, reduced FDDs by representing actions as sets of sets of modifications, and by using smart constructors that eliminate isomorphic subgraphs and contradictory tests.

Figure 2.5 summarizes the syntax, semantics, and well-formedness conditions for FDDs formally. Syntactically, an FDD d is either a constant diagram specified by a set of actions $\{a_1, \dots, a_k\}$, where an action a is a finite map $\{f_1 \leftarrow n_1, \dots, f_k \leftarrow n_k\}$ from fields to values such that each field occurs at most once; or a conditional diagram $(f = n ? d_1 : d_2)$ specified by a test $f = n$ and two sub-diagrams. Semantically, an action a denotes a

$$\begin{array}{ll}
\mathcal{L}[\text{drop}] := \{\} & \mathcal{L}[f \leftarrow n] := \{\{f \leftarrow n\}\} \\
\mathcal{L}[\text{skip}] := \{\{\}\} & \mathcal{L}[f = n] := (f = n ? \{\{\}\} : \{\}) \\
\mathcal{L}[\neg p] := \neg \mathcal{L}[p] & \mathcal{L}[p_1 + p_2] := \mathcal{L}[p_1] \oplus \mathcal{L}[p_2] \\
\mathcal{L}[p^*] := \mathcal{L}[p]^{\circledast} & \mathcal{L}[p_1 \cdot p_2] := \mathcal{L}[p_1] \odot \mathcal{L}[p_2]
\end{array}$$

Figure 2.6: Local compilation to FDDs.

sequence of modifications, a constant diagram $\{a_1, \dots, a_k\}$ denotes the union of the individual actions, and a conditional diagram $(f = n ? d_1 : d_2)$ tests if the packet satisfies the test and evaluates the true branch (d_1) or false branch (d_2) accordingly. The well-formedness judgments $\Gamma \sqsubset (f, n)$ and $\Gamma \vdash d$ ensure that tests appear in ascending order and do not contradict previous tests to the same field. The context Γ keeps track of previous tests and boolean outcomes.

Local compiler. Now we are ready to present the local compiler itself, which goes in two stages. The first stage translates NetKAT source programs into FDDs, using the simple recursive translation given in Figures 2.7 and 2.6; the second stage converts FDDs to forwarding tables.

The NetKAT primitives skip, drop, and $f \leftarrow n$ all compile to simple constant FDDs. Note that the empty action set $\{\}$ drops all packets while the singleton action set $\{\{\}\}$ containing the identity action $\{\}$ copies packets verbatim. NetKAT tests $f = n$ compile to a conditional whose branches are the constant diagrams for skip and drop respectively. NetKAT union, sequence, negation, and star all recursively compile their sub-programs and combine the results using corresponding operations on FDDs, which are given in Figure 2.7.

The FDD union operator ($d_1 \oplus d_2$) walks down the structure of d_1 and d_2 and takes the union of the action sets at the leaves. However, the definition is a bit involved as some care is needed to preserve well-formedness. In particular, when combining multiple

$d_1 \oplus d_2$ (omitting symmetric cases)

$$\{a_{11}, \dots, a_{1m}\} \oplus \{a_{21}, \dots, a_{2n}\} := \{a_{11}, \dots, a_{1m}\} \cup \{a_{21}, \dots, a_{2n}\}$$

$$(f=n ? d_{11} : d_{12}) \oplus \{a_{21}, \dots, a_{2n}\} := (f=n ? d_{11} \oplus \{a_{21}, \dots, a_{2n}\} : d_{12} \oplus \{a_{21}, \dots, a_{2n}\})$$

$$(f_1=n_1 ? d_{11} : d_{12}) \oplus (f_2=n_2 ? d_{21} : d_{22}) :=$$

$$\begin{cases} (f_1=n_1 ? d_{11} \oplus d_{21} : d_{12} \oplus d_{22}) & \text{if } f_1 = f_2 \wedge n_1 = n_2 \\ (f_1=n_1 ? d_{11} \oplus d_{22} : d_{12} \oplus (f_2=n_2 ? d_{21} : d_{22})) & \text{if } f_1 = f_2 \wedge n_1 \sqsubset n_2 \\ (f_1=n_1 ? d_{11} \oplus (f_2=n_2 ? d_{21} : d_{22}) : d_{12} \oplus (f_2=n_2 ? d_{21} : d_{22})) & \text{if } f_1 \sqsubset f_2 \end{cases}$$

$d|_{f=n}$

$$\{a_1, \dots, a_k\}|_{f=n} := (f=n ? \{a_1, \dots, a_k\} : \{\})$$

$$(f_1=n_1 ? d_{11} : d_{12})|_{f=n} := \begin{cases} (f=n ? d_{11} : \{\}) & \text{if } f = f_1 \wedge n = n_1 \\ (d_{12})|_{f=n} & \text{if } f = f_1 \wedge n \neq n_1 \\ (f=n ? (f_1=n_1 ? d_{11} : d_{12}) : \{\}) & \text{if } f \sqsubset f_1 \\ (f_1=n_1 ? (d_{11})|_{f=n} : (d_{12})|_{f=n}) & \text{if } f_1 \sqsubset f \end{cases}$$

$d_1 \odot d_2$

$$a \odot \{a_1, \dots, a_k\} := \{a \odot a_1, \dots, a \odot a_k\}$$

$$a \odot (f=n ? d_1 : d_2) := \begin{cases} a \odot d_1 & \text{if } f \leftarrow n \in a \\ a \odot d_2 & \text{if } f \leftarrow n' \in a \wedge n' \neq n \\ (f=n ? a \odot d_1 : a \odot d_2) & \text{otherwise} \end{cases}$$

$$\{a_1, \dots, a_k\} \odot d := (a_1 \odot d) \oplus \dots \oplus (a_k \odot d)$$

$$(f=n ? d_{11} : d_{12}) \odot d_2 := (d_{11} \odot d_2)|_{f=n} \oplus (d_{12} \odot d_2)|_{f \neq n}$$

$\neg d$

$$\neg\{\} := \{\{\}\}$$

$$\neg\{a_1, \dots, a_k\} := \{\} \text{ for } k \geq 1$$

$$\neg(f=n ? d_1 : d_2) := (f=n ? \neg d_1 : \neg d_2)$$

d^*

$$d^* := \text{fix}(\lambda d_0. \{\{\}\} \oplus d \odot d_0)$$

Figure 2.7: Auxiliary definitions for local compilation to FDDs.

conditional diagrams into one, one must ensure that the ordering on tests is respected and that the final diagram does not contain contradictions. Readers familiar with BDDs may notice that this function is simply the standard “apply” operation (instantiated with union at the leaves). The sequential composition operator $(d_1 \odot d_2)$ merges two packet-processing functions into a single function. It uses auxiliary operations $d|_{f=n}$ and $d|_{f \neq n}$ to restrict a diagram d by a positive or negative test respectively. We elide the sequence operator on atomic actions (which behaves like a right-biased merge of finite maps) and the negative restriction operator (which is similar to positive restriction, but not identical due to contradictory tests) to save space. The first few cases of the sequence operator handle situations where a single action on the left is composed with a diagram on the right. When the diagram on the right is a conditional, $(f=n ? d_1 : d_2)$, we partially evaluate the test using the modifications contained in the action on the left. For example, if the left-action contains the modification $f \leftarrow n$, we know that the test will be true, whereas if the left-action modifies the field to another value, we know the test will be false. The case that handles sequential composition of a conditional diagram on the left is also interesting. It uses restriction and union to implement the composition, reordering and removing contradictory tests as needed to ensure well formedness. The negation $\neg d$ operator is defined in the obvious way. Note that because negation can only be applied to predicates, the leaves of the diagram d are either $\{\}$ or $\{\{\}\}$. Finally, the FDD Kleene star operator d^* is defined using a straightforward fixed-point computation. The well-formedness conditions on FDDs ensures that a fixed point exists.

The soundness of local compilation from NetKAT programs to FDDs is captured by the following theorem:

Theorem 2.3.1 (Local Soundness). *If $\mathcal{L}[[p]] = d$ then $[[p]](h) = [[d]](h)$.*

Proof. Straightforward induction on p . □

The second stage of local compilation converts FDDs to forwarding tables. By design,

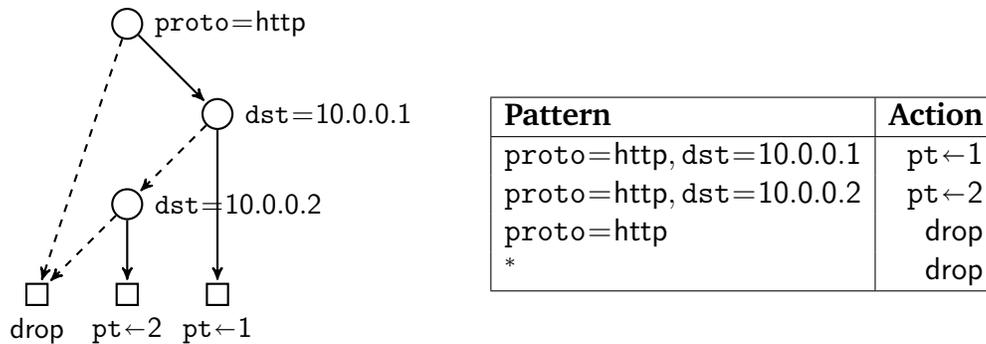


Figure 2.8: Forwarding table generation example.

this transformation is mostly straightforward: we generate a forwarding rule for every path from the root to a leaf, using the conjunction of tests along the path as the pattern and the actions at the leaf. For example, the FDD in Figure 2.8 has four paths from the root to the leaves so the resulting forwarding table has four rules. The left-most path is the highest-priority rule and the right-most path is the lowest-priority rule. Traversing paths from left to right has the effect of traversing true-branches before their associated false-branches. This makes sense, since the only way to encode a negative predicate is to partially shadow a negative-rule with a positive-rule. For example, the last rule in the figure cannot encode the test $\text{proto} \neq \text{http}$. However, since that rule is preceded by a pattern that tests $\text{proto} = \text{http}$, we can reason that the `proto` field is not HTTP in the last rule. If performed naively, this strategy could create a lot of extra forwarding rules—e.g., the table in Figure 2.8 has two drop rules, even though one of them completely shadows the other. In section 2.6, we discuss optimizations that eliminate redundant rules, exploiting the FDD representation.

2.4 Global Compilation

Thus far, we have seen how to compile local NetKAT programs into forwarding tables using FDDs. Now we turn to the global compiler, which translates global programs into

equivalent local programs.

In general, the translation from global to local programs requires introducing extra state, since global programs may use regular expressions to describe end-to-end forwarding paths—e.g., recall the example of a global program with two overlapping paths from Section 2.2. Put another way, because a local program does not contain dup, the compiler can analyze the entire program and generate an equivalent forwarding table that executes on a single switch, whereas the control flow of a global program must be made explicit so execution can be distributed across multiple switches. More formally, a local program encodes a function from packets to sets of packets, whereas a global program encodes a function from packets to sets of packet-histories.

To generate the extra state needed to encode the control flow of a global, distributed execution into a local program, the global compiler translates programs into finite state automata. To a first approximation, the automaton can be thought of as the one for the regular expression embedded in the global program, and the instrumented local program can be thought of as encoding the states and transitions of that automaton in a special header field. The actual construction is a bit more complex for several reasons. First, we cannot instrument the topology in the same way that we instrument switch terms. Second, we have to be careful not to introduce extra states that may lead to duplicate packet histories being generated. Third, NetKAT programs have more structure than ordinary regular expressions, since they denote functions on packet histories rather than sets of strings, so a more complicated notion of automaton—a symbolic NetKAT automaton—is needed.

At a high-level, the global compiler proceeds in several steps:

- It compiles the input program to an equivalent symbolic automaton. All valid paths through the automaton alternate between switch-processing states and topology-processing states, which enables executing them as local programs.

- It introduces a *program counter* by instrumenting the automaton to keep track of the current automaton state in the pc field.
- It determinizes the NetKAT automaton using an analogue of the subset construction for finite automata.
- It uses heuristic optimizations to reduce the number of states.
- It merges all switch-processing states into a single switch state and all topology-processing states into a single topology state.

The final result is a single local program that can be compiled using the local compiler. This program is equivalent to the original global program, modulo the pc field, which records the automaton state.

2.4.1 NetKAT Automata

In prior work, some of the authors introduced NetKAT automata and proved the analogue of Kleene’s theorem: programs and automata have the same expressive power [45]. This allows us to use automata as an intermediate representation for arbitrary NetKAT programs. This section reviews NetKAT automata, which are used in the global compiler, and then presents a function that constructs an automaton from an arbitrary NetKAT program.

Definition 2.4.1 (NetKAT Automaton). A NetKAT automaton is a tuple $(S, s_0, \varepsilon, \delta)$, where:

- S is a finite set of states,
- $s_0 \in S$ is the start state,
- $\varepsilon : S \rightarrow \text{Pk} \rightarrow 2^{\text{Pk}}$ is the observation function, and
- $\delta : S \rightarrow \text{Pk} \rightarrow 2^{\text{Pk} \times S}$ is the continuation function. □

A NetKAT automaton is said to be *deterministic* if δ maps each packet to a unique next state at every state, or more formally if

$$|\{s' : S \mid (pk', s') \in \delta s pk\}| \leq 1$$

for all states s and packets pk and pk' .

The inputs to NetKAT automata are guarded strings drawn from the set $\text{Pk} \cdot (\text{Pk} \cdot \text{dup})^* \cdot \text{Pk}$. That is, the inputs have the form

$$\pi_{in} \cdot \pi_1 \cdot \text{dup} \cdot \pi_2 \cdot \text{dup} \cdot \dots \cdot \pi_n \cdot \text{dup} \cdot \pi_{out}$$

where $n \geq 0$. Intuitively, such strings represent packet-histories through a network: π_{in} is the input state of a packet, π_{out} is the output state, and the π_i are the intermediate states of the packet that are recorded as it travels through the network.

To process such a string, an automaton in state s can either *accept* the trace if $n = 0$ and $\pi_{out} \in \varepsilon s \pi_{in}$, or it can consume one packet and dup from the start of the string and transition to state s' if $n > 0$ and $(pk_1, s') \in \delta s \pi_{in}$. In the latter case, the automaton yields a residual trace:

$$\pi_1 \cdot \pi_2 \cdot \text{dup} \cdot \dots \cdot \pi_n \cdot \text{dup} \cdot \pi_{out}$$

Note that the “output” π_1 of state s becomes the “input” to the successor state s' . More formally, acceptance is defined as:

$$\begin{aligned} \text{accept } s (\pi_{in} \cdot \pi_{out}) & \quad :\iff \quad \pi_{out} \in \varepsilon s \pi_{in} \\ \text{accept } s (\pi_{in} \cdot \pi_1 \cdot \text{dup} \cdot w) & \quad :\iff \quad \bigvee_{(\pi_1, s') \in \delta s \pi_{in}} \text{accept } s' (\pi_1 \cdot w) \end{aligned}$$

Next, we define a function that builds an automaton $A(p)$ from an arbitrary NetKAT program p such that

$$(\pi_{out} :: \pi_n :: \dots :: \langle \pi_1 \rangle) \in \llbracket p \rrbracket \langle pk_{in} \rangle \quad \iff \quad \text{accept}_{A(p)} s_0 (\pi_{in} \cdot \pi_1 \cdot \text{dup} \cdot \dots \cdot \pi_{out})$$

p	$\mathcal{E}[[p]] \in \text{Pol}$	$\mathcal{D}[[p]] \in 2^{\text{Pol} \times L \times \text{Pol}}$
t	t	\emptyset
$f \leftarrow n$	$f \leftarrow n$	\emptyset
dup^ℓ	drop	$\{\langle \text{skip}, \ell, \text{skip} \rangle\}$
$q + r$	$\mathcal{E}[[q]] + \mathcal{E}[[r]]$	$\mathcal{D}[[q]] \cup \mathcal{D}[[r]]$
$q \cdot r$	$\mathcal{E}[[q]] \cdot \mathcal{E}[[r]]$	$\mathcal{D}[[q]] \cdot r \cup \mathcal{E}[[q]] \cdot \mathcal{D}[[r]]$
q^*	$\mathcal{E}[[q]]^*$	$\mathcal{E}[[q^*]] \cdot \mathcal{D}[[q]] \cdot q^*$

Figure 2.9: Auxiliary definitions for NetKAT automata construction.

The construction is based on Antimirov partial derivatives for regular expressions [8]. We fix a set of labels L , and annotate each occurrence of dup in the source program p with a unique label $\ell \in L$. We then define a pair of functions:

$$\mathcal{E}[[\cdot]] : \text{Pol} \rightarrow \text{Pol} \qquad \mathcal{D}[[\cdot]] : \text{Pol} \rightarrow 2^{\text{Pol} \times L \times \text{Pol}}$$

Intuitively, $\mathcal{E}[[p]]$ can be thought of as extracting the local components from p (and will be used to construct ε), while $\mathcal{D}[[p]]$ extracts the global components (and will be used to construct δ). A triple $\langle d, \ell, k \rangle \in \mathcal{D}[[p]]$ represents the derivative of p with respect to dup^ℓ . That is, d is the dup -free component of p up to dup^ℓ , and k is the residual program (or *continuation*) of p after dup^ℓ .

We calculate $\mathcal{E}[[p]]$ and $\mathcal{D}[[p]]$ simultaneously using a simple recursive algorithm defined in Figure 2.9. The definition makes use of the following abbreviations,

$$\begin{aligned} \mathcal{D}[[p]] \cdot q &:= \{\langle d, \ell, k \cdot q \rangle \mid \langle d, \ell, k \rangle \in \mathcal{D}[[p]]\} \\ q \cdot \mathcal{D}[[p]] &:= \{\langle q \cdot d, \ell, k \rangle \mid \langle d, \ell, k \rangle \in \mathcal{D}[[p]]\} \end{aligned}$$

which lift sequencing to sets of triples in the obvious way.

The next lemma characterizes $\mathcal{E}[[p]]$ and $\mathcal{D}[[p]]$, using the following notation to reconstruct programs from sets of triples:

$$\sum \mathcal{D}[[p]] := \sum_{\langle d, \ell, k \rangle \in \mathcal{D}[[p]]} d \cdot \text{dup} \cdot k$$

Lemma 2.4.2 (Characterization of $\mathcal{E}[\cdot]$ and $\mathcal{D}[\cdot]$). *For all programs p , we have the following:*

- (a) $p \equiv \mathcal{E}[p] + \sum \mathcal{D}[p]$.
- (b) $\mathcal{E}[p]$ is a local program.
- (c) For all $\langle d, \ell, k \rangle \in \mathcal{D}[p]$, d is a local program.
- (d) For all labels ℓ in p , there exist unique programs d and k such that $\langle d, \ell, k \rangle \in \mathcal{D}[p]$.

Proof. By structural induction on p . Claims (b – d) are trivial. Claim (a) can be proved purely equationally using only the NetKAT axioms and the KAT-DENESTING rule from [6]. \square

Lemma 2.4.2 (d) allows us to write k_ℓ to refer to the unique continuation of dup^ℓ . By convention, we let k_0 denote the “initial continuation,” namely p .

Definition 2.4.3 (Program Automaton). The NetKAT automaton $A(p)$ for a program p is defined as $(S, s_0, \varepsilon, \delta)$ where

- S is the set of labels occurring in p , plus the initial label 0.
- $s_0 := 0$
- $\varepsilon \ell \pi := \{\pi' \mid \langle pk' \rangle \in \llbracket \mathcal{E}[k_\ell] \rrbracket \langle pk \rangle\}$
- $\delta \ell \pi := \{(\pi', \ell') \mid \langle d, \ell', k \rangle \in \mathcal{D}[k_\ell] \wedge \langle pk' \rangle \in \llbracket d \rrbracket \langle pk \rangle\}$ \square

Theorem 2.4.4 (Program Automaton Soundness). *For all programs p , packets π and histories h , we have*

$$h \in \llbracket p \rrbracket \langle \pi_{in} \rangle \iff \text{accept}_{s_0} (\pi_{in} \cdot \pi_1 \cdot \text{dup} \cdot \dots \cdot \pi_n \cdot \text{dup} \cdot \pi_{out})$$

where $h = pk_{out} :: pk_n :: \dots :: \langle \pi_1 \rangle$.

Proof. We first strengthen the claim, replacing $\langle pk_{in} \rangle$ with an arbitrary history $pk_{in} :: h'$, s_0 with an arbitrary label $\ell \in S$, and p with k_ℓ . We then proceed by induction on the length of the history, using Lemma 2.4.2 for the base case and induction step. \square

2.4.2 Local Program Generation

With a NetKAT automaton $A(p)$ for the global program p in hand, we are now ready to construct a local program. The main idea is to make the state of the global automaton explicit in the local program by introducing a new header field pc (represented concretely using VLANs, MPLS tags, or any other unused header field) that keeps track of the state as the packet traverses the network. This encoding enables simulating the automaton for the global program using a single local program (along with the physical topology). We also discuss determinization and optimization, which are important for correctness and performance.

Program counter. The first step in local program generation is to encode the state of the automaton into its observation and transition functions using the pc field. To do this, we use the same structures as are used by the local compiler, FDDs. Recall that the observation function ε maps input packets to output packets according to $\mathcal{E}[[k_\ell]]$, which is a dup-free NetKAT program. Hence, we can encode the observation function for a given state ℓ as a conditional FDD that tests whether pc is ℓ and either behaves like the FDD for $\mathcal{E}[[k_\ell]]$ or drop. We can encode the continuation function δ as an FDD in a similar fashion, although we also have to set the pc to each successor state s' . This symbolic representation of automata using FDDs allows us to efficiently manipulate automata despite the large size of their “input alphabet”, namely $|P_k \times P_k|$. In our implementation we introduce the pc field and FDDs on the fly as automata are constructed, rather than adding them as a post-processing step, as is described here for ease of exposition.

Determinization. The next step in local program generation is to determinize the NetKAT automaton. This step turns out to be critical for correctness—it eliminates extra outputs that would be produced if we attempted to directly implement a nondeterministic

NetKAT automaton. To see why, consider a program of the form $p + p$. Intuitively, because union is an idempotent operation, we expect that this program will behave the same as just a single copy of p . However, this will not be the case when p contains a `dup`: each occurrence of `dup` will be annotated with a different label. Therefore, when we instrument the program to track automaton states, it will create two packets that are identical except for the `pc` field, instead of one packet as required by the semantics. The solution to this problem is simply to determinize the automaton before converting it to a local program. Determinization ensures that every packet trace induces a unique path through the automaton and prevents duplicate packets from being produced. Using FDDs to represent the automaton symbolically is crucial for this step: it allows us to implement a NetKAT analogue of the subset construction efficiently.

Optimization. One practical issue with building automata using the algorithms described so far is that they can use a large number of states—one for each occurrence of `dup` in the program—and determinization can increase the number of states by an exponential factor. Although these automata are not wrong, attempting to compile them can lead to practical problems since extra states will trigger a proliferation of forwarding rules that must be installed on switches. Because switches today often have limited amounts of memory—often only a few thousand forwarding rules—reducing the number of states is an important optimization. An obvious idea is to optimize the automaton using (generalizations of) textbook minimization algorithms. Unfortunately this would be prohibitively expensive since deciding whether two states are equal is a costly operation in the case of NetKAT automata. Instead, we adopt a simple heuristic that works well in practice and simply merge states that are identical. In particular, by representing the observation and transition functions as FDDs, which are hash consed, testing equality is cheap—simple pointer comparisons.

Local Program Extraction. The final step is to extract a local program from the automaton. Recall from Section 2.2 that, by definition, links are enclosed by dups on either side, and links are the only NetKAT terms that contain dups or modify the switch field. It follows that every global program gives rise to a bipartite NetKAT automaton in which all accepting paths alternate between “switch states” (which do not modify the switch field) and “link states” (which forward across links and do modify the switch field), beginning with a switch state. Intuitively, the local program we want to extract is simply the union of the ε and δ FDDs of all switch states (recall Lemma 2.4.2 (a)), with the link states implemented by the physical network. Note however, that the physical network will neither match on the pc nor advance the pc to the next state (while the link states in our automaton do). To fix the latter, we observe that any link state has a unique successor state. We can thus simply advance the pc by two states instead of one at every switch state, anticipating the missing pc modification in link states. To address the former, we employ the equivalence

$$[sw_1:pt_1] \rightarrow [sw_2:pt_2] \equiv sw=1 \cdot pt=1 \cdot t \cdot sw=2 \cdot pt=2$$

It allows us to replace links with the entire topology if we modify switch states to match on the appropriate source and destination locations immediately before and after transitioning across a link. After modifying the ε and δ FDDs accordingly and taking the union of all switch states as described above, the resulting FDD can be passed to the local compiler to generate forwarding tables.

The tables will correctly implement the global program provided the physical topology (in, t, out) satisfies the following:

- $p \equiv in \cdot p \cdot out$, i.e. the global program specifies end-to-end forwarding paths
- t implements at least the links used in p .
- $t \cdot in \equiv drop \equiv out \cdot t$, i.e. the in and out predicates should not include locations that are internal to the network.

2.5 Virtual Compilation

The third and final stage of our compiler pipeline translates virtual programs to physical programs. Recall that a virtual program is one that is defined over a virtual topology. Network virtualization can make programs easier to write by abstracting complex physical topologies to simpler topologies and also makes programs portable across different physical topologies. It can even be used to multiplex several virtual networks onto a single physical network—*e.g.*, in multi-tenant datacenters [85].

To compile a virtual program, the compiler needs to know the mapping between virtual switches, ports, and links and their counterparts at the physical level. The programmer supplies a virtual program v , a virtual topology t , sets of ingress and egress locations for t , and a relation \mathcal{R} between virtual and physical ports. The relation \mathcal{R} must map each physical ingress to a virtual ingress, and conversely for egresses, but is otherwise unconstrained—*e.g.*, it need not be injective or even a function.² The constraints on ingresses and egresses ensures that each packet entering the physical network lifts uniquely to a packet in the virtual network, and similarly for packets exiting the virtual network. During execution of the virtual program, each packet can be thought of as having two locations, one in the virtual network and one in the physical network; \mathcal{R} defines which pairs of locations are consistent with each other. For simplicity, we assume the virtual program is a local program. If it is not, the programmer can use the global compiler to put it into local form.

Overview. To execute a virtual program on a physical network, possibly with a different underlying topology, the compiler must (i) instrument the program to keep track of packet locations in the virtual topology and (ii) implement forwarding between locations

²Actually, we can relax this condition slightly and allow physical ingresses to map to zero or one virtual ingresses—if a physical ingress has no corresponding representative in the virtual network, then packets arriving at that ingress will not be admitted to the virtual network.

that are adjacent in the virtual topology using physical paths. To achieve this, the virtual compiler proceeds as follows:

1. It instruments the program to use the virtual switch (vsw) and virtual port (vpt) fields to keep track of the location of the packet in the virtual topology.
2. It constructs a *fabric*: a NetKAT program that updates the physical location of a packet when its virtual location changes and vice versa, after each step of processing to restore consistency with respect to the virtual-physical relation, \mathcal{R} .
3. It assembles the final program by combining v with the fabric, eliminating the vsw and vpt fields, and compiling the result using the global compiler.

Most of the complexity arises in the second step because there may be many valid fabrics (or there may be none). However, this step is independent of the virtual program. The fabric can be computed once and for all and then be reused as the program changes. Fabrics can be generated in several ways—*e.g.*, to minimize a cost function such as path length or latency, maximize disjointness, etc.

Instrumentation. To keep track of a packet’s location in the virtual network, we introduce new packet fields vsw and vpt for the virtual switch and the virtual port, respectively. We replace all occurrences of the sw or pt field in the program v and the virtual topology t with vsw and vpt respectively using a simple textual substitution. Packets entering the physical network must be lifted to the virtual network. Hence, we replace in with a program that matches on all physical ingress locations \mathbb{I} and initializes vsw and vpt in accordance with \mathcal{R} :

$$in' := \sum_{\substack{(sw,pt) \in \mathbb{I} \\ (vsw,vpt) \mathcal{R} (sw,pt)}} sw = sw \cdot pt = pt \cdot vsw \leftarrow vsw \cdot vpt \leftarrow vpt$$

Recall that we require \mathcal{R} to relate each location in \mathbb{I} to at most one virtual ingress, so the program lifts each packet to at most one ingress location in the virtual network. The

$$\begin{array}{c}
\mathcal{V}\text{-POL} \\
\frac{(vsw, vpt, \mathbb{I}) \rightarrow_v (vsw, vpt', \mathbb{O})}{\left[\begin{array}{c} (vsw, vpt, \mathbb{I}) \\ (sw, pt, \mathbb{I}) \end{array} \right] \rightarrow \left[\begin{array}{c} (vsw, vpt', \mathbb{O}) \\ (sw, pt, \mathbb{I}) \end{array} \right]} \\
\mathcal{F}\text{-OUT} \\
\frac{\begin{array}{c} (sw, pt, \mathbb{I}) \rightarrow_p^+ (sw', pt', \mathbb{O}) \\ (vsw, vpt) \mathcal{R} (sw', pt') \end{array}}{\left[\begin{array}{c} (vsw, vpt, \mathbb{O}) \\ (sw, pt, \mathbb{I}) \end{array} \right] \rightarrow \left[\begin{array}{c} (vsw, vpt, \mathbb{O}) \\ (sw', pt', \mathbb{O}) \end{array} \right]} \\
\mathcal{F}\text{-LOOP-IN} \\
\frac{(vsw, vpt) \mathcal{R} (sw, pt)}{\left[\begin{array}{c} (vsw, vpt, \mathbb{I}) \\ (sw, pt, \mathbb{O}) \end{array} \right] \rightarrow \left[\begin{array}{c} (vsw, vpt, \mathbb{I}) \\ (sw, \text{Loop } pt, \mathbb{I}) \end{array} \right]} \\
\mathcal{V}\text{-TOPO} \\
\frac{(vsw, vpt, \mathbb{O}) \rightarrow_v (vsw', vpt', \mathbb{I})}{\left[\begin{array}{c} (vsw, vpt, \mathbb{O}) \\ (sw, pt, \mathbb{O}) \end{array} \right] \rightarrow \left[\begin{array}{c} (vsw', vpt', \mathbb{I}) \\ (sw, pt, \mathbb{O}) \end{array} \right]} \\
\mathcal{F}\text{-IN} \\
\frac{\begin{array}{c} (sw, pt, \mathbb{O}) \rightarrow_p^+ (sw', pt', \mathbb{I}) \\ (vsw, vpt) \mathcal{R} (sw', pt') \end{array}}{\left[\begin{array}{c} (vsw, vpt, \mathbb{I}) \\ (sw, pt, \mathbb{O}) \end{array} \right] \rightarrow \left[\begin{array}{c} (vsw, vpt, \mathbb{I}) \\ (sw', pt', \mathbb{I}) \end{array} \right]} \\
\mathcal{F}\text{-LOOP-OUT} \\
\frac{\begin{array}{c} (sw, pt, \mathbb{O}) \rightarrow_p^* (sw', pt', \mathbb{O}) \\ (vsw, vpt) \mathcal{R} (sw', pt') \end{array}}{\left[\begin{array}{c} (vsw, vpt, \mathbb{O}) \\ (sw, \text{Loop } pt, \mathbb{I}) \end{array} \right] \rightarrow \left[\begin{array}{c} (vsw, vpt, \mathbb{O}) \\ (sw', pt', \mathbb{O}) \end{array} \right]}
\end{array}$$

Figure 2.10: Fabric game graph edges.

<p>Reachable Nodes</p> $ \frac{\begin{array}{c} (sw, pt) \in \mathbb{I} \\ (vsw, vpt) \mathcal{R} (sw, pt) \end{array}}{\left[\begin{array}{c} (vsw, vpt, \mathbb{I}) \\ (sw, pt, \mathbb{I}) \end{array} \right] \in V} \text{ING} $ $ \frac{u \in V \quad u \rightarrow v}{v \in V} \text{TRANS} $	<p>Fatal Nodes</p> $ \frac{v = \left[\begin{array}{c} (vsw, vpt, d_1) \\ (sw, pt, d_2) \end{array} \right] \quad \begin{array}{c} d_1 \neq d_2 \\ \forall u. v \rightarrow u \implies u \text{ is fatal} \end{array}}{v \text{ is fatal}} \mathcal{F}\text{-FATAL} $ $ \frac{v = \left[\begin{array}{c} (vsw, vpt, d_1) \\ (sw, pt, d_2) \end{array} \right] \quad \begin{array}{c} d_1 = d_2 \\ \exists u. v \rightarrow u \wedge u \text{ is fatal} \end{array}}{v \text{ is fatal}} \mathcal{V}\text{-FATAL} $
--	--

Figure 2.11: Reachable and fatal nodes.

vsw and vpt fields are only used to track locations during the early stages of virtual compilation. They are completely eliminated in the final assembly. Hence, we will not need to introduce additional tags to implement the resulting physical program.

Fabric construction. Each packet can be thought of as having two locations: one in the virtual topology and one in the underlying physical topology. After executing in' , the locations are consistent according to the virtual-physical relation \mathcal{R} . However,

consistency can be broken after each step of processing using the virtual program v or virtual topology t . To restore consistency, we construct a *fabric* comprising programs f_{in} and f_{out} from the virtual and physical topologies and \mathcal{R} , and insert it into the program:

$$q := in' \cdot (v \cdot f_{out}) \cdot (t \cdot f_{in} \cdot v \cdot f_{out})^* \cdot out$$

In this program, v and t alternate with f_{out} and f_{in} in processing packets, thereby breaking and restoring consistency repeatedly. Intuitively, it is the job of the fabric to keep the virtual and physical locations in sync.

This process can be viewed as a two-player game between a virtual player \mathcal{V} (embodied by v and t) and a fabric player \mathcal{F} (embodied by f_{out} and f_{in}). The players take turns moving a packet across the virtual and the physical topology, respectively. Player \mathcal{V} wins if the fabric player \mathcal{F} fails to restore consistency after a finite number of steps; player \mathcal{F} wins otherwise. Constructing a fabric now amounts to finding a winning strategy for \mathcal{F} .

We start by building the game graph $G = (V, E)$ modeling all possible ways that consistency can be broken by \mathcal{V} or restored by \mathcal{F} . Nodes are pairs of virtual and physical locations, $[l_v, l_p]$, where a location is a 3-tuple comprising a switch, a port, and a direction that indicates if the packet is entering the port (I) or leaving the port (O). The rules in Figure 2.10 determine the edges of the game graph:

- The edge $[l_v, l_p] \rightarrow [l'_v, l_p]$ exists if \mathcal{V} can move packets from l_v to l'_v . There are two ways to do so: either \mathcal{V} moves packets across a virtual switch (\mathcal{V} -POL) or across a virtual link (\mathcal{V} -TOPO). In the inference rules, we write \rightarrow_v to denote a single hop in the virtual topology:

$$(vsw, vpt, d) \rightarrow_v (vsw', vpt', d')$$

If $d = \text{I}$ and $d' = \text{O}$ then the hop is across one switch, but if $d = \text{O}$ and $d' = \text{I}$ then the hop is across a link.

- The edge $[l_v, l_p] \rightarrow [l_v, l'_p]$ exists if \mathcal{F} can move packets from l_p to l'_p . When \mathcal{F} makes a move, it must restore physical-virtual consistency (the \mathcal{R} relation in the premise

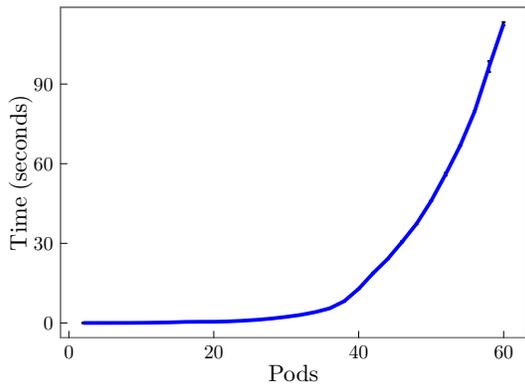
of \mathcal{F} -POL and \mathcal{F} -TOPO). To do so, it may need to take several hops through the physical network (written as \rightarrow_p^+).

- In addition, \mathcal{F} may leave a packet at their current location, if the location is already consistent (\mathcal{F} -LOOP-IN and \mathcal{F} -LOOP-OUT). Note that these force a packet located at physical location $(sw, pt, 0)$ to leave through port pt eventually. Intuitively, once the fabric has committed to emitting the packet through a given port, it can only delay but not withdraw that commitment.

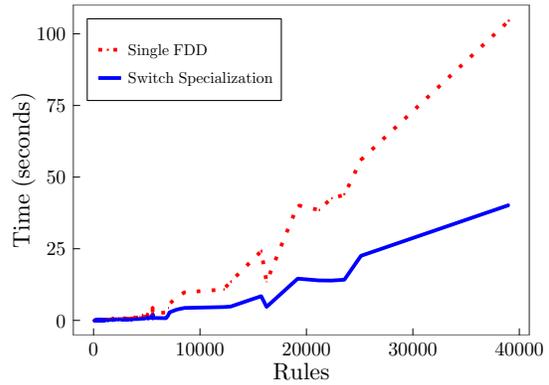
Although these rules determine the complete game graph, all packets enter the network at an ingress location (determined by the in' predicate). Therefore, we can restrict our attention to only those nodes that are reachable from the ingress (reachable nodes in Figure 2.11). In the resulting graph $G = (V, E)$, every path represents a possible trajectory that a packet processed by q may take through the virtual and physical topology.

In addition to removing unreachable nodes, we must remove *fatal nodes*, which are the nodes where \mathcal{F} is unable to restore consistency and thus loses the game. \mathcal{F} -FATAL says that any state from which \mathcal{F} is unable to move to a non-fatal state is fatal. In particular, this includes states in which \mathcal{F} cannot move to any other state at all. \mathcal{V} -FATAL says that any state in which \mathcal{V} can move to a fatal state is fatal. Intuitively, we define such states to be fatal since we want the fabric to work for any virtual program the programmer may write. Fatal states can be removed using a simple backwards traversal of the graph starting from nodes without outgoing edges. This process may remove ingress nodes if they turn out to be fatal. This happens if and only if there exists no fabric that can always restore consistency for arbitrary virtual programs. Of course, this case can only arise if the physical topology is not bidirectional.

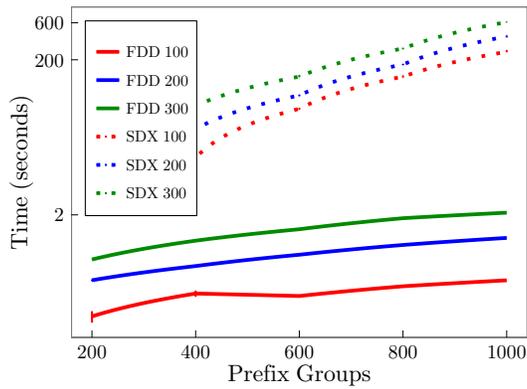
Fabric selection. If all ingress nodes withstand pruning, the resulting graph encodes exactly the set of all winning strategies for \mathcal{F} , *i.e.* the set of all possible fabrics. A



(a) Routing on k -pod fat-trees.



(b) Destination-based routing on topology zoo.



(c) Time needed to compile SDX benchmarks.

Figure 2.12: Experimental results: compilation time.

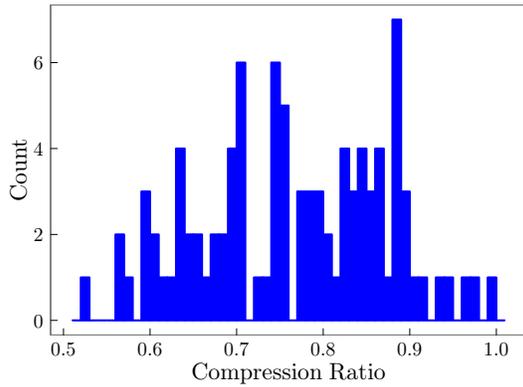
fabric is a subgraph of G that contains the ingress, is closed under all possible moves by the virtual program, and contains exactly one edge out of every state in which \mathcal{F} has to restore consistency. The \mathcal{F} -edges must be labeled with concrete paths through the physical topology, as there may exist several paths implementing the necessary multi-step transportation from the source node to the target node.

In general, there may be many fabrics possible and the choice of different \mathcal{F} -edges correspond to fabrics with different characteristics, such as minimizing hop counts,

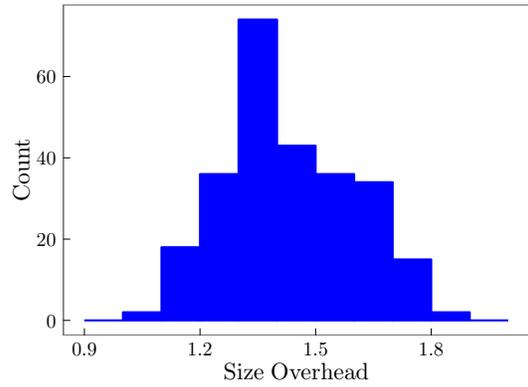
maximizing disjoint paths, and so on. Our compiler implements several simple strategies. For example, given a metric φ on paths (such as hop count), our greedy strategy starts at the ingresses and adds a node whenever it is reachable through an edge e rooted at a node u already selected, and e is (i) any \mathcal{V} -player edge or (ii) the \mathcal{F} -player edge with path π minimizing φ among all edges and their paths rooted at u .

After a fabric is selected, it is straightforward to encode it as a NetKAT term. Every \mathcal{F} -edge $[l_v, l_p] \rightarrow [l_v, l'_p]$ in the graph is encoded as a NetKAT term that matches on the locations l_v and l_p , forwards along the corresponding physical path from l_p to l'_p , and then resets the virtual location to l_v . Resetting the virtual location is semantically redundant but will make it easy to eliminate the `vsw` and `vpt` fields. We then take f_{in} to be the union of all \mathcal{F} -IN-edges, and f_{out} to be the union of all \mathcal{F} -OUT-edges. NetKAT's global abstractions play a key role, providing the building blocks for composing multiple overlapping paths into a unified fabric.

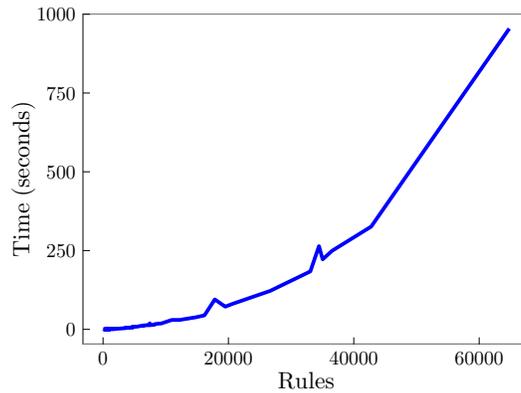
End-to-end Compilation. After the programs in' , f_{in} , and f_{out} , are calculated from \mathcal{R} , we assemble the physical program q , defined above. However, one last potential problem remains: although the virtual compiler adds instrumentation to update the physical switch and port fields, the program still matches and updates the virtual switch (`vsw`) and virtual port (`vpt`). However, note that by construction of q , any match on the `vsw` or `vpt` field is preceded by a modification of those fields on the same physical switch. Therefore, all matches are automatically eliminated during FDD generation, and only modifications of the `vsw` and `vpt` fields remain. These can be safely erased before generating flow tables. Intuitively, the program counter inserted into q by the global compiler plays double-duty to track both the physical location and the virtual location of a packet. Hence, we only need a single tag to compile virtual programs!



(a) Compressing Classbench ACLs.



(b) Table size overhead for global programs.



(c) Compilation time for global programs.

Figure 2.13: Experimental results: table compression and global compilation.

2.6 Evaluation

To evaluate our compiler, we conducted experiments on a diverse set of real-world topologies and benchmarks. In practice, our compiler is a module that is used by the Frenetic SDN controller to map NetKAT programs to flow tables. Whenever network events occur, *e.g.*, a host connects, a link fails, traffic patterns change, and so on, the controller may react by generating a new NetKAT program. Since network events may occur rapidly, a slow compiler can easily be a bottleneck that prevents the controller

from reacting quickly to network events. In addition, the flow tables that the compiler generates must be small enough to fit on the available switches. Moreover, as small tables can be updated faster than large tables, table size affects the controller’s reaction time too.

Therefore, in all the following experiments we measure flow-table compilation time and flow-table size. We apply the compiler to programs for a variety of topologies, from topology designs for very large datacenters to a dataset of real-world topologies. We highlight the effect of important optimizations to the fundamental FDD-based algorithms. We perform all experiments on 32-core, 2.6 GHz Intel Xeon E5-2650 machines with 64GB RAM.³ We repeat all timing experiments ten times and plot their average.

Fat trees. A fat-tree [3] is a modern datacenter network design that uses commodity switches to minimize cost. It provides several redundant paths between hosts that can be used to maximize available bandwidth, provide backup paths, and so on. A fat-tree is organized into pods, where a k -pod fat-tree topology can support up to $\frac{k^3}{4}$ hosts. A real-world datacenter might have up to 48 pods [3]. Therefore, our compiler should be able to generate forwarding programs for a 48-pod fat tree relatively quickly.

Figure 2.12a shows how the time needed to generate all flow tables varies with the number of pods in a fat-tree.⁴ The graph shows that we take approximately 30 seconds to produce tables for 48-pod fat trees (*i.e.*, 27,000 hosts) and less than 120 seconds to generate programs for 60-pod fat trees (*i.e.*, 54,000 hosts).

This experiment shows that the compiler can generate tables for large datacenters. But, this is partly because the fat-tree forwarding algorithm is topology-dependent and leverages symmetries to minimize the amount of forwarding rules needed. Many real-world topologies are not regular and require topology-independent forwarding programs.

³Our compiler is single-threaded and doesn’t leverage multicore.

⁴This benchmark uses the switch-specialization optimization, which we describe in the next section.

In the next section, we demonstrate that our compiler scales well with these topologies too.

Topology Zoo. The *Topology Zoo* [82] is a dataset of a few hundred real-world network topologies of varying size and structure. For every topology in this dataset, we use *destination-based routing* to connect all nodes to each other. In destination-based routing, each switch filters packets by their destination address and forwards them along a spanning-tree rooted at the destination. Since each switch must be able to forward to any destination, the total number of rules must be $\mathcal{O}(n^2)$ for an n -node network.

Figure 2.12b shows how the running time of the compiler varies across the topology zoo benchmarks. The curves are not as smooth as the curve for fat-trees, since the complexity of forwarding depends on features of network topology. Since the topology zoo is so diverse, this is a good suite to exercise the *switch specialization* optimization that dramatically reduces compile time.

A direct implementation of the local compiler builds one FDD for the entire network and uses it to generate flow tables for each switch. However, since several FDD (and BDD) algorithms are fundamentally quadratic, it helps to first specialize the program for each switch and then generate a small FDD for each switch in the network (*switch specialization*). Building FDDs for several smaller programs is typically much faster than building a single FDD for the entire network. As the graph shows, this optimization has a dramatic effect on all but the smallest topologies.

SDX. Our experiments thus far have considered some quite large forwarding programs, but none of them leverage software-defined networking in any interesting way. In this section, we report on our performance on benchmarks from a recent SIGCOMM paper [58] that proposes a new application of SDN.

An Internet exchange point (IXP) is a physical location where networks from several

ISPs connect to each other to exchange traffic. Legal contracts between networks are often implemented by routing programs at IXPs. However, today’s IXPs use baroque protocols that needlessly limit the kinds of programs that can be implemented. A Software-defined IXP (an “SDX” [58]) gives participants fine-grained control over packet-processing and peering using a high-level network programming language. The SDX prototype uses Pyretic [116] to encode policies and presents several examples that demonstrate the power of an expressive network programming language.

We build a translator from Pyretic to NetKAT and use it to evaluate our compiler on SDX’s own benchmarks. These benchmarks simulate a large IXP where a few hundred peers apply programs to several hundred prefix groups. The dashed lines in Figure 2.12c reproduce a graph from the SDX paper, which shows how compilation time varies with the number of prefix groups and the number of participants in the SDX.⁵ The solid lines show that our compiler is orders of magnitude faster. Pyretic takes over 10 minutes to compile the largest benchmark, but our compiler only takes two seconds.

Although Pyretic is written in Python, which is a lot slower than OCaml, the main problem is that Pyretic has a simple table-based compiler that does not scale (Section 2.2). In fact, the authors of SDX had to add several optimizations to get the graph depicted. Despite these optimizations, our FDD-based approach is substantially faster.

The SDX paper also reports flow-table sizes for the same benchmark. At first, our compiler appeared to produce tables that were twice as large as Pyretic. Naturally, we were unhappy with this result and investigated. Our investigation revealed a bug in the Pyretic compiler, which would produce incorrect tables that were artificially small. The authors of SDX have confirmed this bug and it has been fixed in later versions of Pyretic. We are actively working with them to port SDX to NetKAT to help SDX scale further.

⁵We get nearly the same numbers as the SDX paper on our hardware.

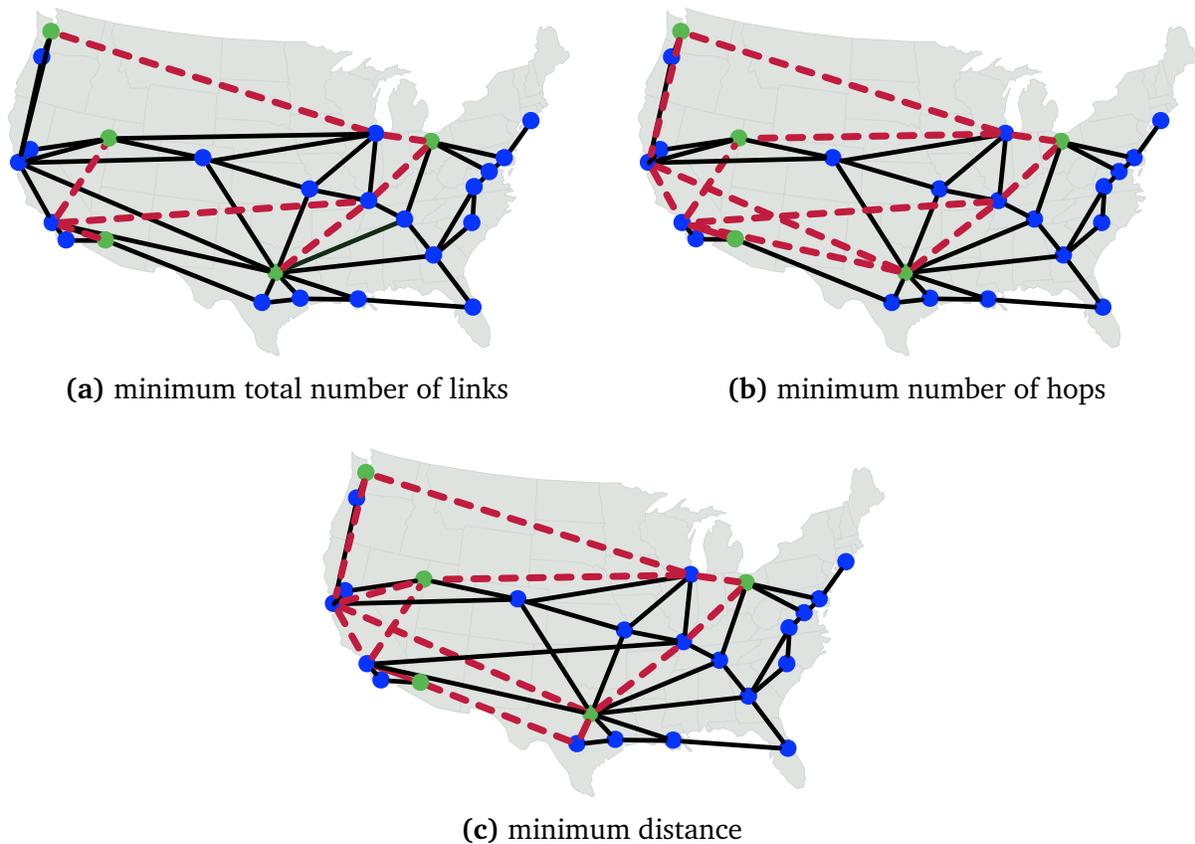


Figure 2.14: Three fabrics optimizing different metrics

Classbench. Lastly, we compile ACLs generated using *Classbench* [168]. These are realistic firewall rules that showcase another optimization: it is often possible to significantly compress tables by combining and eliminating redundant rules.

We build an optimizer for the flow-table generation algorithm in Figure 2.8. Recall that that we generate flow-tables by converting every complete path in the FDD into a rule. Once a path has been traversed, we can remove it from the FDD without harm. However, naively removing a path may produce an FDD that is not reduced. Our optimization is simple: we remove paths from the FDD as they are turned into rules and ensure that the FDD is reduced at each step. When the last path is turned into a rule, we are left with a trivial FDD. This iterative procedure prevents several

unnecessary rules from being generated. It is possible to implement other canonical optimizations. But, this optimization is unique because it leverages properties of reduced FDDs. Figure 2.13a shows that this approach can produce 30% fewer rules on average than a direct implementation of flow-table generation. We do not report running times for the optimizer, but it is negligible in all our experiments.

Global compiler. The benchmarks discussed so far only use the local compiler. In this section, we focus on the global compiler. Since the global compiler introduces new abstractions, we can't apply it to existing benchmarks, such as SDX, which use local programs. Instead, we need to build our own benchmark suite of global programs. To do so, we build a generator that produces global programs that describe paths between hosts. Again, an n -node topology has $\mathcal{O}(n^2)$ paths. We apply this generator to the Topology Zoo, measuring compilation time and table size:

- *Compilation time:* since the global compiler leverages FDDs, we can expect automata generation to be fast. However, global compilation involves other steps such as determinization and localization and their effects on compilation time may matter. Figure 2.13c shows how compilation time varies with the total number of rules generated. This graph does grow faster than local compilation time on the same benchmark (the red, dashed line in Figure 2.12b). Switch-specialization, which dramatically reduces the size of FDDs and hence compilation time, does not work on global programs. Therefore, it makes most sense to compare this graph to local compilation with a single FDD.
- *Table size:* The global compiler has some optimizations to eliminate unnecessary states, which produces fewer rules. However, it does not fully minimize NetKAT automata thus it may produce more rules than equivalent local programs. Figure 2.13b shows that on the topology zoo, global routing produces tables that are no more than twice as large as local routing.

We believe these results are promising: we spent a lot of time tuning the local compiler, but the global compiler is an early prototype with much room for improvement.

Virtualization case study. Finally, we present a small case study that showcases the virtual compiler on a snapshot of the AT&T backbone network circa 2007–2008. This network is part of the Topology Zoo and shown in Figure 2.14. We construct a “one big switch” virtual network and use it to connect five nodes (highlighted in green) to each other:

$$\sum_{n=1}^5 \text{dst}=10.0.0.n \cdot \text{pt} \leftarrow n$$

To map the virtual network to the physical network, we generate three different fabrics: (a) a fabric that minimizes the total number of links used across the network, (b) a fabric that minimizes the number of hops between hosts, and (c) a fabric that minimizes the physical length of the path between hosts. In the figure, the links utilized by each of these fabrics is highlighted in red.

The three fabrics give rise to three very different implementations of the same virtual program. Note that the program and the fabric are completely independent of each other and can be updated independently. For example, the operator managing the physical network could change the fabric to implement a new SLA, *e.g.* move from minimum-utilization to shortest-paths. This change requires no update to the virtual program; the network would witness performance improvement for free. Similarly, the virtual network operator could decide to implement a new firewall policy in the virtual network or change the forwarding behavior. The old fabric would work seamlessly with this new virtual program without intervention by the physical network operator. In principle, our compiler could even be used repeatedly to virtualize virtual networks.

2.7 Related Work

A large body of work has explored the design of high-level languages for SDN programming [39, 85, 115, 116, 139, 140, 174]. Our work is unique in its focus on the task of engineering efficient compilers that scale up to large topologies as well as expressive global and virtual programs.

An early paper by Monsanto et al. proposed the NetCore language and presented an algorithm for compiling programs based on forwarding tables [115]. Subsequent work by Guha *et al.* developed a verified implementation of NetCore in the Coq proof assistant [57]. Anderson et al. developed NetKAT as an extension to NetCore and proposed a compilation algorithm based on manipulating nested conditionals, which are essentially equivalent to forwarding tables. The correctness of the algorithm was justified using NetKAT’s equational axioms, but didn’t handle global programs or Kleene star. Concurrent NetCore [154] grows NetCore with features that target next-generation SDN-switches. The original Pyretic paper implemented a “reactive microflow interpreter” and not a compiler [116]. However later work developed a compiler in the style of NetCore. SDX uses Pyretic to program Internet exchange points [58]. CoVisor develops incremental algorithms for maintaining forwarding table in the presence of changes to programs composed using NetCore-like operators [70]. Recent work by Jose *et al.* developed a compiler based on integer linear programming for next-generation switches, each with multiple, programmable forwarding tables [73].

A number of papers in the systems community have proposed mechanisms for implementing virtual network programs. An early workshop paper by Casado proposed the idea of network virtualization and sketched an implementation strategy based on a hypervisor [23]. Our virtual compiler extends this basic strategy by introducing a generalized notion of a fabric, developing concrete algorithms for computing and selecting

fabrics, and showing how to compose fabrics with virtual programs in the context of a high-level language. Subsequent work by Koponen et al. described VMware’s NVP platform, which implements hypervisor-based virtualization in multi-tenant datacenters [85]. Pyretic [116], CoVisor [70], and OpenVirteX [5] all support virtualization—the latter at three different levels of abstraction: topology, address, and control application. However, none of these papers present a complete description of algorithms for computing the forwarding state needed to implement virtual networks.

The FDDs used in our local compiler as well as our algorithms for constructing NetKAT automata are inspired by Pous’s work on symbolic KAT automata [137] and work by some of the authors on a verification tool for NetKAT [45]. The key differences between this work and ours is that they focus on verification of programs whereas we develop compilation algorithms. BDDs have been used for verification for several decades [2, 21]. In the context of networks, BDDs and BDD-like structures have been used to optimize access control policies [103], TCAMs [104], and to verify [79] data plane configurations, but our work is the first to use BDDs to compile network programs.

2.8 Conclusion

This chapter describes the first complete compiler for the NetKAT language. It presents a suite of tools that leverage BDDs, graph algorithms, and symbolic automata to efficiently compile programs in the NetKAT language down to compact forwarding tables for SDN switches. In the future, it would be interesting to investigate whether richer constructs such as stateful and probabilistic programs can be implemented using our techniques, how classic algorithms from the automata theory literature can be adapted to optimize global programs, how incremental algorithms can be incorporated into our compiler, and how the compiler can assist in performing graceful dynamic updates to network state.

Part II

Probabilistic Networks

Chapter 3

Semantic Foundations

“Symbols should not be confused with the concepts they denote.”

—Dana Scott

ProbNetKAT is a probabilistic extension of NetKAT with a denotational semantics based on Markov kernels. The language is expressive enough to generate continuous distributions, which raises the question of how to compute effectively in the language. This chapter gives a new characterization of ProbNetKAT’s semantics using domain theory, which provides the foundation needed to build a practical implementation. We show how to use the semantics to approximate the behavior of arbitrary ProbNetKAT programs using distributions with finite support. We develop a prototype implementation and show how to use it to solve a variety of problems including characterizing the expected congestion induced by different routing schemes and reasoning probabilistically about reachability in a network.

3.1 Introduction

The recent emergence of software-defined networking (SDN) has led to the development of a number of domain-specific programming languages [43, 116, 120, 174] and reason-

ing tools [6, 45, 77, 79] for networks. But there is still a large gap between the models provided by these languages and the realities of modern networks. In particular, most existing SDN languages have semantics based on deterministic packet-processing functions, which makes it impossible to encode probabilistic behaviors. This is unfortunate because in the real world, network operators often use randomized protocols and probabilistic reasoning to achieve good performance.

Previous work on ProbNetKAT [44] proposed an extension to the NetKAT language [6, 45] with a random choice operator that can be used to express a variety of probabilistic behaviors. ProbNetKAT has a compositional semantics based on Markov kernels that conservatively extends the deterministic NetKAT semantics and has been used to reason about various aspects of network performance including congestion, fault tolerance, and latency. However, although the language enjoys a number of attractive theoretical properties, there are some major impediments to building a practical implementation: (i) the semantics of iteration is formulated as an infinite process rather than a fixpoint in a suitable order, and (ii) some programs generate continuous distributions. These factors make it difficult to determine when a computation has converged to its final value, and there are also challenges related to representing and analyzing distributions with infinite support.

This chapter introduces a new semantics for ProbNetKAT, following the approach pioneered by Saheb-Djahromi, Jones, and Plotkin [71, 72, 135, 150, 151]. Whereas the original semantics of ProbNetKAT was somewhat imperative in nature, being based on stochastic processes, the semantics introduced in this chapter is purely functional. Nevertheless, the two semantics are closely related—we give a precise, technical characterization of the relationship between them. The new semantics provides a suitable foundation for building a practical implementation, it provides new insights into the nature of probabilistic behavior in networks, and it opens up several interesting theoretical

questions for future work.

Our new semantics follows the order-theoretic tradition established in previous work on Scott-style domain theory [1, 155]. In particular, Scott-continuous maps on algebraic and continuous DCPOs both play a key role in our development. However, there is an interesting twist: NetKAT and ProbNetKAT are not *state-based* as with most other probabilistic systems, but are rather *throughput-based*. A ProbNetKAT program can be thought of as a filter that takes an input set of packet histories and generates an output randomly distributed on the measurable space 2^H of sets of packet histories. The closest thing to a “state” is a set of packet histories, and the structure of these sets (e.g., the lengths of the histories they contain and the standard subset relation) are important considerations. Hence, the fundamental domains are not flat domains as in traditional domain theory, but are instead the DCPO of sets of packet histories ordered by the subset relation. Another point of departure from prior work is that the structures used in the semantics are not subprobability distributions, but genuine probability distributions: with probability 1, some set of packets is output, although it may be the empty set.

It is not obvious that such an order-theoretic semantics should exist at all. Traditional probability theory does not take order and compositionality as fundamental structuring principles, but prefers to work in monolithic sample spaces with strong topological properties such as Polish spaces. Prototypical examples of such spaces are the real line, Cantor space, and Baire space. The space of sets of packet histories 2^H is homeomorphic to the Cantor space, and this was the guiding principle in the development of the original ProbNetKAT semantics. Although the Cantor topology enjoys a number of attractive properties (compactness, metrizability, strong separation) that are lost when shifting to the Scott topology, the sacrifice is compensated by a more compelling least-fixpoint characterization of iteration that aligns better with the traditional domain-theoretic treatment. Intuitively, the key insight that underpins our development is the observation

that ProbNetKAT programs are monotone: if a larger set of packet histories is provided as input, then the likelihood of seeing any particular set of packets as a subset of the output set can only increase. From this germ of an idea, we formulate an order-theoretic semantics for ProbNetKAT.

In addition to the strong theoretical motivation for this work, our new semantics also provides a source of practical useful reasoning techniques, notably in the treatment of iteration and approximation. The original paper on ProbNetKAT showed that the Kleene star operator satisfies the usual fixpoint equation $P^* = \text{skip} \ \& \ P \cdot P^*$, and that its finite approximants $P^{(n)}$ converge weakly (but not pointwise) to it. However, it was not characterized as a least fixpoint in any order or as a canonical solution in any sense. This was a bit unsettling and raised questions as to whether it was the “right” definition—questions for which there was no obvious answer. This chapter characterizes P^* as the least fixpoint of the Scott-continuous map $X \mapsto \text{skip} \ \& \ P \cdot X$ on a continuous DCPO of Scott-continuous Markov kernels. This not only corroborates the original definition as the “right” one, but provides a powerful tool for monotone approximation. Indeed, this result implies the correctness of our prototype implementation, which we have used to build and evaluate several applications inspired by common real-world scenarios.

Contributions. This main contributions of this chapter are as follows: (i) we develop a domain-theoretic foundation for probabilistic network programming, (ii) using this semantics, we build a prototype implementation of the ProbNetKAT language, and (iii) we evaluate the applicability of the language on several case studies.

Outline. The paper is structured as follows. In §3.2 we give a high-level overview of our technical development using a simple running example. In §3.3 we review basic definitions from domain theory and measure theory. In §3.4 we formalize the syntax and semantics of ProbNetKAT abstractly in terms of a monad. In §3.5 we prove a general

theorem relating the Scott and Cantor topologies on 2^H . Although the Scott topology is much weaker, the two topologies generate the same Borel sets, so the probability measures are the same in both. We also show that the bases of the two topologies are related by a countably infinite-dimensional triangular linear system, which can be viewed as an infinite analog of the inclusion-exclusion principle. The cornerstone of this result is an extension theorem (Theorem 3.5.4) that determines when a function on the basic Scott-open sets extends to a measure. In §3.6 we give the new domain-theoretic semantics for ProbNetKAT in which programs are characterized as Markov kernels that are Scott-continuous in their first argument. We show that this class of kernels forms a continuous DCPO, the basis elements being those kernels that drop all but fixed finite sets of input and output packets. In §3.7 we show that ProbNetKAT’s primitives are (Scott-)continuous and its program operators preserve continuity. Other operations such as product and Lebesgue integration are also treated in this framework. In proving these results, we attempt to reuse general results from domain theory whenever possible, relying on the specific properties of 2^H only when necessary. We supply complete proofs for folklore results and in cases where we could not find an appropriate original source. We also show that the two definitions of the Kleene star operator—one in terms of an infinite stochastic process and one as the least fixpoint of a Scott-continuous map—coincide. In §3.8 we apply the continuity results from §3.7 to derive monotone convergence theorems. In §3.9 we describe a prototype implementation based on §3.8 and several applications. In §3.10 we review related work. We conclude in §3.11 by discussing open problems and future directions.

3.2 Overview

This section provides motivation for the ProbNetKAT language and summarizes our main results using a simple example.

Example. Consider the topology shown in Figure 3.1 and suppose we are asked to implement a routing application that forwards all traffic to its destination while minimizing congestion, gracefully adapting to shifts in load, and also handling unexpected failures. This problem is known as traffic engineering in the networking literature and has been extensively studied [9, 42, 61, 68, 141]. Note that standard shortest-path routing (SPF) does not solve the problem as stated—in general, it can lead to bottlenecks and also makes the network vulnerable to failures. For example, consider sending a large amount of traffic from host h_1 to host h_3 : there are two paths in the topology, one via switch S_2 and one via switch S_4 , but if we only use a single path we sacrifice half of the available capacity. The most widely-deployed approaches to traffic engineering today are based on using multiple paths and randomization. For example, Equal Cost Multipath Routing (ECMP), which is widely supported on commodity routers, selects a least-cost path for each traffic flow uniformly at random. The intention is to spread the offered load across a large set of paths, thereby reducing congestion without increasing latency.

ProbNetKAT Language. Using ProbNetKAT, it is straightforward to write a program that captures the essential behavior of ECMP. We first construct programs that model the routing tables and topology, and build a program that models the behavior of the entire network.

Routing: We model the routing tables for the switches using simple ProbNetKAT programs that match on destination addresses and forward packets on the next hop toward their destination. To randomly map packets to least-cost paths, we use the choice operator (\oplus). For example, the program for switch S1 in Figure 3.1 is as follows:

$$\begin{aligned}
 p_1 &:= (\text{dst}=h_1 \cdot \text{pt} \leftarrow 1) \\
 &\quad \& (\text{dst}=h_2 \cdot \text{pt} \leftarrow 2) \\
 &\quad \& (\text{dst}=h_3 \cdot (\text{pt} \leftarrow 2 \oplus \text{pt} \leftarrow 4)) \\
 &\quad \& (\text{dst}=h_4 \cdot \text{pt} \leftarrow 4)
 \end{aligned}$$

The programs for other switches are similar. To a first approximation, this program can be read as a routing table, whose entries are separated by the parallel composition operator (&). The first entry states that packets whose destination is h_1 should be forwarded out on port 1 (which is directly connected to h_1). Likewise, the second entry states that packets whose destination is host h_2 should be forwarded out on port 2, which is the next hop on the unique shortest path to h_2 . The third entry, however, is different: it states that packets whose destination is h_3 should be forwarded out on ports 2 and 4 with equal probability. This divides traffic going to h_3 among the clockwise path via S_2 and the counter-clockwise path via S_4 . The final entry states that packets whose destination is h_4 should be forwarded out on port 4, which is again the next hop on the unique shortest path to h_4 . The routing program for the network is the parallel composition of the programs for each switch:

$$p := (\mathbf{sw}=S_1 \cdot p_1) \ \& \ (\mathbf{sw}=S_2 \cdot p_2) \ \& \ (\mathbf{sw}=S_3 \cdot p_3) \ \& \ (\mathbf{sw}=S_4 \cdot p_4)$$

Topology: We model a directed link as a program that matches on the switch and port at one end of the link and modifies the switch and port to the other end of the link. We model an undirected link l as a parallel composition of directed links in each direction. For example, the link between switches S_1 and S_2 is modeled as follows:

$$l_{1,2} := (\mathbf{sw}=S_1 \cdot \mathbf{pt}=2 \cdot \mathbf{dup} \cdot \mathbf{sw}\leftarrow S_2 \cdot \mathbf{pt}\leftarrow 1 \cdot \mathbf{dup}) \\ \& \ (\mathbf{sw}=S_2 \cdot \mathbf{pt}=1 \cdot \mathbf{dup} \cdot \mathbf{sw}\leftarrow S_1 \cdot \mathbf{pt}\leftarrow 2 \cdot \mathbf{dup})$$

Note that at each hop we use ProbNetKAT's `dup` operator to store the headers in the packet's history, which records the trajectory of the packet as it goes through the network. Histories are useful for tasks such as measuring path length and analyzing link congestion. We model the topology as a parallel composition of individual links:

$$t := l_{1,2} \ \& \ l_{2,3} \ \& \ l_{3,4} \ \& \ l_{1,4}$$

To delimit the network edge, we define ingress and egress predicates:

$$in := (sw=1 \cdot pt=1) \ \& \ (sw=2 \cdot pt=2) \ \& \ \dots$$

$$out := (sw=1 \cdot pt=1) \ \& \ (sw=2 \cdot pt=2) \ \& \ \dots$$

Here, since every ingress is an egress, the predicates are identical.

Network: We model the end-to-end behavior of the entire network by combining p , t , in and out into a single program:

$$net := in \cdot (p \cdot t)^* \cdot p \cdot out$$

This program models processing each input from ingress to egress across a series of switches and links. Formally it denotes a Markov kernel that, when supplied with an input distribution on packet histories μ produces an output distribution ν .

Queries: Having constructed a probabilistic model of the network, we can use standard tools from measure theory to reason about performance. For example, to compute the expected congestion on a given link l , we would introduce a function Q from sets of packets to $\mathbb{R} \cup \{\infty\}$ (formally a random variable):

$$Q(a) := \sum_{h \in a} \#_l(h)$$

where $\#_l(h)$ is the function on packet histories that returns the number of times that link l occurs in h , and then compute the expected value of Q using integration:

$$\mathbf{E}_\nu[Q] = \int Q \, d\nu$$

We can compute queries that capture other aspects of network performance such as latency, reliability, etc. in similar fashion.

Limitations. Unfortunately there are several issues with the approach just described:

- One problem is that computing the results of a query can require complicated measure theory since a ProbNetKAT program may generate a continuous distribution

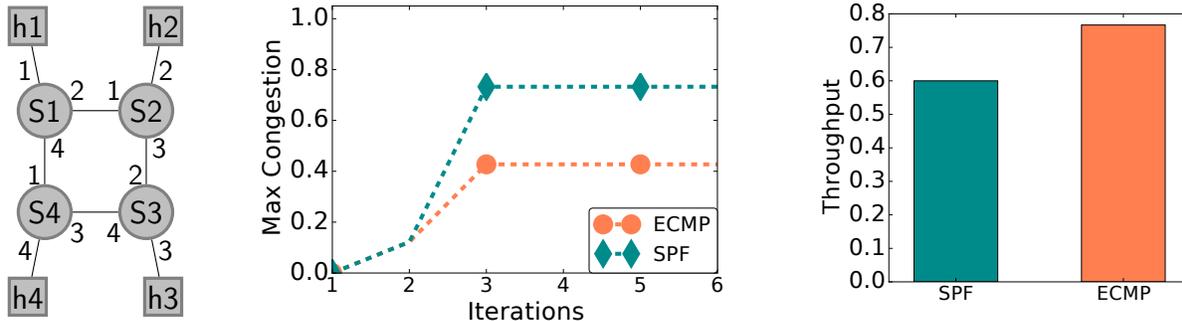


Figure 3.1: topology, congestion, failure throughput.

in general (Lemma 3.4.3). Formally, instead of summing over the support of the distribution, we have to use Lebesgue integration in an appropriate measurable space. Of course, there are also challenges in representing infinite distributions in an implementation.

- Another issue is that the semantics of iteration is modeled in terms of an infinite stochastic process rather than a standard fixpoint. The original ProbNetKAT paper showed that it is possible to approximate a program using a series of star-free programs that weakly converge to the correct result, but the approximations need not converge monotonically, which makes this result difficult to apply in practice.
- Even worse, many of the queries that we would like to answer are not actually continuous in the Cantor topology, meaning that the weak convergence result does not even apply! The notion of distance on sets of packet histories is $d(a, b) = 2^{-n}$ where n is the length of the smallest history in a but not in b , or vice versa. It is easy to construct a sequence of histories h_n of length n such that $\lim_{n \rightarrow \infty} d(\{h_n\}, \{\}) = 0$ but $\lim_{n \rightarrow \infty} Q(\{h_n\}) = \infty$ which is not equal to $Q(\{\}) = 0$.

Together, these issues are significant impediments that make it difficult to apply ProbNetKAT in many scenarios.

Domain-Theoretic Semantics. This chapter develops a new semantics for ProbNetKAT that overcomes these problems and provides the key building blocks needed to engineer a practical implementation. The main insight is that we can formulate the semantics in terms of the Scott topology rather than the Cantor topology. It turns out that these two topologies generate the same Borel sets, and the relationship between them can be characterized using an extension theorem that captures when functions on the basic Scott-open sets extend to a measure. We show how to construct a DCPO equipped with a natural partial order that also lifts to a partial order on Markov kernels. We prove that standard program operators are continuous, which allows us to formulate the semantics of the language—in particular Kleene star—using standard tools from domain theory, such as least fixpoints. Finally, we formalize a notion of approximation and prove a monotone convergence theorem.

The problems with the original ProbNetKAT semantics identified above are all solved using the new semantics. Because the new semantics models iteration as a least fixpoint, we can work with finite distributions and star-free approximations that are guaranteed to monotonically converge to the analytical solution (Corollary 3.8.2). Moreover, whereas our query Q was not Cantor continuous, it is straightforward to show that it is Scott continuous. Let A be an increasing chain $a_0 \subseteq a_1 \subseteq a_2 \subseteq \dots$ ordered by inclusion. Scott continuity requires $\bigsqcup_{a \in A} Q(a) = Q(\bigsqcup A)$ which is easy to prove. Hence, the convergence theorem applies and we can compute a monotonically increasing chain of approximations that converge to $\mathbf{E}_\nu[Q]$.

Implementation and Applications. We developed the first implementation of ProbNetKAT using the new semantics. We built an interpreter for the language and implemented a variety of traffic engineering schemes including ECMP, k -shortest path routing, and oblivious routing [141]. We analyzed the performance of each scheme in terms of congestion and latency on real-world demands drawn from Internet2’s Abilene

backbone, and in the presence of link failures. We showed how to use the language to reason probabilistically about reachability properties such as loops and black holes. Figures 3.1 (b-c) depict the expected throughput and maximum congestion when using shortest paths (SPF) and ECMP on the 4-node topology as computed by our ProbNetKAT implementation. We set the demand from h_1 to h_3 to be $\frac{1}{2}$ units of traffic, and the demand between all other pairs of hosts to be $\frac{1}{8}$ units. The first graph depicts the maximum congestion induced under successive approximations of the Kleene star, and shows that ECMP achieves much better congestion than SPF. With SPF, the most congested link (from S_1 to S_2) carries traffic from h_1 to h_2 , from h_4 to h_2 , and from h_1 to h_3 , resulting in $\frac{3}{4}$ total traffic. With ECMP, the same link carries traffic from h_1 to h_2 , half of the traffic from h_2 to h_4 , half of the traffic from h_1 to h_3 , resulting in $\frac{7}{16}$ total traffic. The second graph depicts the loss of throughput when the same link fails. The total aggregate demand is $1\frac{7}{8}$. With SPF, $\frac{3}{4}$ units of traffic are dropped leaving $1\frac{1}{8}$ units, which is 60% of the demand, whereas with ECMP only $\frac{7}{16}$ units of traffic are dropped leaving $1\frac{7}{16}$ units, which is 77% of the demand.

3.3 Preliminaries

This section briefly reviews basic concepts from topology, measure theory, and domain theory, and defines *Markov kernels*, the objects on which ProbNetKAT's semantics is based. For a more detailed account, the reader is invited to consult standard texts [1, 35].

Topology. A *topology* $\mathcal{O} \subseteq 2^X$ on a set X is a collection of subsets including X and \emptyset that is closed under finite intersection and arbitrary union. A pair (X, \mathcal{O}) is called a *topological space* and the sets $U, V \in \mathcal{O}$ are called the *open sets* of (X, \mathcal{O}) . A function $f : X \rightarrow Y$ between topological spaces (X, \mathcal{O}_X) and (Y, \mathcal{O}_Y) is *continuous* if the preimage

of any open set in Y is open in X , i.e. if

$$f^{-1}(U) = \{x \in X \mid f(x) \in U\} \in \mathcal{O}_X$$

for any $U \in \mathcal{O}_Y$.

Measure Theory. A σ -algebra $\mathcal{F} \subseteq 2^X$ on a set X is a collection of subsets including X that is closed under complement, countable union, and countable intersection. A *measurable space* is a pair (X, \mathcal{F}) . A *probability measure* μ over such a space is a function $\mu : \mathcal{F} \rightarrow [0, 1]$ that assigns probabilities $\mu(A) \in [0, 1]$ to the *measurable sets* $A \in \mathcal{F}$, and satisfies the following conditions:

- $\mu(X) = 1$
- $\mu(\bigcup_{i \in I} A_i) = \sum_{i \in I} \mu(A_i)$ whenever $\{A_i\}_{i \in I}$ is a countable collection of disjoint measurable sets.

Note that these conditions already imply that $\mu(\emptyset) = 0$. Elements $a, b \in X$ are called *points* or *outcomes*, and measurable sets $A, B \in \mathcal{F}$ are also called *events*. The σ -algebra $\sigma(U)$ generated by a set $U \subseteq X$ is the smallest σ -algebra containing U :

$$\sigma(U) := \bigcap \{ \mathcal{F} \subseteq 2^X \mid \mathcal{F} \text{ is a } \sigma\text{-algebra and } U \subseteq \mathcal{F} \}.$$

Note that it is well-defined because the intersection is not empty (2^X is trivially a σ -algebra containing U) and intersections of σ -algebras are again σ -algebras. If $\mathcal{O} \subseteq 2^X$ are the open sets of X , then the smallest σ -algebra containing the open sets $\mathcal{B} = \sigma(\mathcal{O})$ is the *Borel algebra*, and the measurable sets $A, B \in \mathcal{B}$ are the *Borel sets* of X .

Let $P_\mu := \{a \in X \mid \mu(\{a\}) > 0\}$ denote the points (not events!) with non-zero probability. It can be shown that P_μ is countable. A probability measure is called *discrete* if $\mu(P_\mu) = 1$. Such a measure can simply be represented by a function $Pr : X \rightarrow [0, 1]$ with $Pr(a) = \mu(\{a\})$. If $|P_\mu| < \infty$, the measure is called *finite* and can be represented

by a finite map $Pr : P_\mu \rightarrow [0, 1]$. In contrast, measures for which $\mu(P_\mu) = 0$ are called *continuous*, and measures for which $0 < \mu(P_\mu) < 1$ are called *mixed*. The *Dirac measure* or *point mass* puts all probability on a single point $a \in X$: $\delta_a(A) = 1$ if $a \in A$ and 0 otherwise. The uniform distribution on $[0, 1]$ is a continuous measure.

A function $f : X \rightarrow Y$ between measurable spaces (X, \mathcal{F}_X) and (Y, \mathcal{F}_Y) is called *measurable* if the preimage of any measurable set in Y is measurable in X , i.e. if

$$f^{-1}(A) := \{x \in X \mid f(x) \in A\} \in \mathcal{F}_X$$

for all $A \in \mathcal{F}_Y$. If $Y = \mathbb{R} \cup \{-\infty, +\infty\}$, then f is called a *random variable* and its *expected value* with respect to a measure μ on X is given by the Lebesgue integral

$$\mathbf{E}_\mu[f] := \int f d\mu = \int_{x \in X} f(x) \cdot \mu(dx)$$

If μ is discrete, the integral simplifies to the sum

$$\mathbf{E}_\mu[f] = \sum_{x \in X} f(x) \cdot \mu(\{x\}) = \sum_{x \in P_\mu} f(x) \cdot Pr(x)$$

Markov Kernels. Consider a probabilistic transition system with states X that makes a random transition between states at each step. If X is finite, the system can be captured by a transition matrix $T \in [0, 1]^{X \times X}$, where the matrix entry T_{xy} gives the probability that the system transitions from state x to state y . Each row T_x describes the transition function of a state x and must sum to 1. Suppose that the start state is initially distributed according to the row vector $V \in [0, 1]^X$, i.e. the system starts in state $x \in X$ with probability V_x . Then, the state distribution is given by the matrix product $VT \in [0, 1]^X$ after one step and by VT^n after n steps.

Markov kernels generalize this idea to infinite state systems. Given measurable spaces (X, \mathcal{F}_X) and (Y, \mathcal{F}_Y) , a Markov kernel with source X and target Y is a function $P : X \times \mathcal{F}_Y \rightarrow [0, 1]$ (or equivalently, $X \rightarrow \mathcal{F}_Y \rightarrow [0, 1]$) that maps each source state $x \in X$ to a distribution over target states $P(x, -) : \mathcal{F}_Y \rightarrow [0, 1]$. If the initial distribution

is given by a measure ν on X , then the target distribution μ after one step is given by Lebesgue integration:

$$\mu(A) := \int_{x \in X} P(x, A) \cdot \nu(dx) \quad (A \in \mathcal{F}_Y) \quad (3.1)$$

If ν and $P(x, -)$ are discrete, the integral simplifies to the sum

$$\mu(\{y\}) = \sum_{x \in X} P(x, \{y\}) \cdot \nu(\{x\}) \quad (y \in Y)$$

which is just the familiar vector-matrix-product VT . Similarly, two kernels P, Q from X to Y and from Y to Z , respectively, can be sequentially composed to a kernel $P \cdot Q$ from X to Z :

$$(P \cdot Q)(x, A) := \int_{y \in Y} P(x, dy) \cdot Q(y, A) \quad (3.2)$$

This is the continuous analog of the matrix product TT . A Markov kernel P must satisfy two conditions:

- (i) For each source state $x \in X$, the map $A \mapsto P(x, A)$ must be a probability measure on the target space.
- (ii) For each event $A \in \mathcal{F}_Y$ in the target space, the map $x \mapsto P(x, A)$ must be a measurable function.

Condition ((ii)) is required to ensure that integration is well-defined. A kernel P is called *deterministic* if $P(a, -)$ is a dirac measure for each a .

Domain Theory. A *partial order* (PO) is a pair (D, \sqsubseteq) where D is a set and \sqsubseteq is a reflexive, transitive, and antisymmetric relation on D . For two elements $x, y \in D$ we let $x \sqcup y$ denote their \sqsubseteq -least upper bound (*i.e.*, their supremum), provided it exists. Analogously, the least upper bound of a subset $C \subseteq D$ is denoted $\sqcup C$, provided it exists. A non-empty subset $C \subseteq D$ is *directed* if for any two $x, y \in C$ there exists *some* upper

bound $x, y \sqsubseteq z$ in C . A *directed complete partial order* (DCPO) is a PO for which any directed subset $C \subseteq D$ has a supremum $\bigsqcup C$ in D . If a PO has a least element it is denoted by \perp , and if it has a greatest element it is denoted by \top . For example, the nonnegative real numbers with infinity $\mathbb{R}_+ := [0, \infty]$ form a DCPO under the natural order \leq with suprema $\bigsqcup C = \sup C$, least element $\perp = 0$, and greatest element $\top = \infty$. The unit interval is a DCPO under the same order, but with $\top = 1$. Any powerset 2^X is a DCPO under the subset order, with suprema given by union.

A function f from D to E is called (*Scott-*)*continuous* if

- (i) it is monotone, i.e. $x \sqsubseteq y$ implies $f(x) \sqsubseteq f(y)$, and
- (ii) it preserves suprema, i.e. $f(\bigsqcup C) = \bigsqcup_{x \in C} f(x)$ for any directed set C in D .

Equivalently, f is continuous with respect to the *Scott topologies* on D and E [1, Proposition 2.3.4], which we define next. (Note how condition ((ii)) looks like the classical definition of continuity of a function f , but with suprema taking the role of limits). The set of all continuous functions $f : D \rightarrow E$ is denoted $[D \rightarrow E]$.

A subset $A \subseteq D$ is called *up-closed* (or an *upper set*) if $a \in A$ and $a \sqsubseteq b$ implies $b \in A$. The smallest up-closed superset of A is called its *up-closure* and is denoted A^\uparrow . A is called (*Scott-*)*open* if it is up-closed and intersects every directed subset $C \subseteq D$ that satisfies $\bigsqcup C \in A$. For example, the Scott-open sets of \mathbb{R}_+ are the upper semi-infinite intervals $(r, \infty]$, $r \in \mathbb{R}_+$. The Scott-open sets form a topology on D called the *Scott topology*.

DCPOs enjoy many useful closure properties:

- (i) The cartesian product of any collection of DCPOs is a DCPO with componentwise order and suprema.
- (ii) If E is a DCPO and D any set, the function space $D \rightarrow E$ is a DCPO with pointwise order and suprema.
- (iii) The continuous functions $[D \rightarrow E]$ between DCPOs D and E form a DCPO with pointwise order and suprema.

If D is a DCPO with least element \perp , then any Scott-continuous self-map $f \in [D \rightarrow D]$ has a \sqsubseteq -least fixpoint, and it is given by the supremum of the chain $\perp \sqsubseteq f(\perp) \sqsubseteq f(f(\perp)) \sqsubseteq \dots$:

$$\text{lfp}(f) = \bigsqcup_{n \geq 0} f^n(\perp)$$

Moreover, the least fixpoint operator, $\text{lfp} \in [[D \rightarrow D] \rightarrow D]$ is itself continuous, that is: $\text{lfp}(\bigsqcup C) = \bigsqcup_{f \in C} \text{lfp}(f)$, for any directed set of functions $C \subseteq [D \rightarrow D]$.

An element a of a DCPO is called *finite* (Abramsky and Jung [1] use the term *compact*) if for any directed set A , if $a \sqsubseteq \bigsqcup A$, then there exists $b \in A$ such that $a \sqsubseteq b$. Equivalently, a is finite if its up-closure $\{a\}^\uparrow$ is Scott-open. A DCPO is called *algebraic* if for every element b , the finite elements \sqsubseteq -below b form a directed set and b is the supremum of this set. An element a of a DCPO *approximates* another element b , written $a \ll b$, if for any directed set A , $a \sqsubseteq c$ for some $c \in A$ whenever $b \sqsubseteq \bigsqcup A$. A DCPO is called *continuous* if for every element b , the elements \ll -below b form a directed set and b is the supremum of this set. Every algebraic DCPO is continuous. A set in a topological space is *compact-open* if it is compact (every open cover has a finite subcover) and open.

Here we recall some basic facts about DCPOs. These are all well-known, but we state them as a lemma for future reference.

Lemma 3.3.1 (DCPO Basic Facts).

- (i) Let E be a DCPO and D_1, D_2 sets. There is a homeomorphism (bicontinuous bijection) curry between the DCPOs $D_1 \times D_2 \rightarrow E$ and $D_1 \rightarrow D_2 \rightarrow E$, where the function spaces are ordered pointwise. The inverse of curry is uncurry.
- (ii) In an algebraic DCPO, the open sets $\{a\}^\uparrow$ for finite a form a base for the Scott topology.
- (iii) A subset of an algebraic DCPO is compact-open iff it is a finite union of basic open sets $\{a\}^\uparrow$.

Syntax		Semantics $\boxed{\llbracket p \rrbracket \in 2^{\mathsf{H}} \rightarrow \mathcal{M}(2^{\mathsf{H}})}$
Naturals	$n ::= 0 \mid 1 \mid 2 \mid \dots$	$\llbracket \text{drop} \rrbracket(a) := \eta(\emptyset)$
Fields	$f ::= f_1 \mid \dots \mid f_k$	$\llbracket \text{skip} \rrbracket(a) := \eta(a)$
Packets	$\text{Pk} \ni \pi ::= \{f_1 = n_1, \dots, f_k = n_k\}$	$\llbracket f = n \rrbracket(a) := \eta(\{\pi :: \bar{h} \in a \mid \pi.f = n\})$
Histories	$\text{H} \ni h ::= \pi :: \bar{h}$	$\llbracket \neg t \rrbracket(a) := \llbracket t \rrbracket(a) \ggg \lambda b. \eta(a - b)$
	$\bar{h} ::= \langle \rangle \mid \pi :: \bar{h}$	$\llbracket f \leftarrow n \rrbracket(a) := \eta(\{\pi [f := n] :: \bar{h} \mid \pi :: \bar{h} \in a\})$
Probabilities	$r \in [0, 1]$	$\llbracket \text{dup} \rrbracket(a) := \eta(\{\pi :: \pi :: \bar{h} \mid \pi :: \bar{h} \in a\})$
Predicates	$t, u ::= \text{drop}$	$\llbracket p \& q \rrbracket(a) :=$
	$\quad \mid \text{skip}$	$\llbracket p \rrbracket(a) \ggg \lambda b_1. \llbracket q \rrbracket(a) \ggg \lambda b_2. \eta(b_1 \cup b_2)$
	$\quad \mid f = n$	$\llbracket p \cdot q \rrbracket(a) := \llbracket p \rrbracket(a) \ggg \llbracket q \rrbracket$
	$\quad \mid t \& u$	$\llbracket p \oplus_r q \rrbracket(a) := r \cdot \llbracket p \rrbracket(a) + (1 - r) \cdot \llbracket q \rrbracket(a)$
	$\quad \mid t \cdot u$	$\llbracket p^* \rrbracket(a) := \bigsqcup_{n \in \mathbb{N}} \llbracket p^{(n)} \rrbracket(a), \text{ where}$
	$\quad \mid \neg t$	$p^{(0)} := \text{skip}$ $p^{(n+1)} := \text{skip} \& p \cdot p^{(n)} \quad (n \geq 0)$
Programs	$p, q ::= t$	Probability Monad $\langle \mathcal{D}, \eta, \ggg \rangle$
	$\quad \mid f \leftarrow n$	$\mathcal{D}(X) := \{\text{prob. measures } \mu: \mathcal{B} \rightarrow [0, 1]\}$
	$\quad \mid \text{dup}$	$\eta(a) := \delta_a$
	$\quad \mid p \& q$	$\mu \ggg P := \lambda A. \int_{a \in X} P(a)(A) \cdot \mu(da)$
	$\quad \mid p \cdot q$	
	$\quad \mid p \oplus_r q$	
	$\quad \mid p^*$	

Figure 3.2: ProbNetKAT: syntax and semantics, parameterized over monad $\mathcal{M}(-)$.

3.4 ProbNetKAT

This section defines the syntax and semantics of ProbNetKAT formally (see Figure 3.2) and establishes some basic properties. ProbNetKAT is a core calculus designed to capture the essential forwarding behavior of probabilistic network programs. In particular, the language includes primitives that model fundamental constructs such as parallel and sequential composition, iteration, and random choice. It does not model features such as mutable state, asynchrony, and dynamic updates, although extensions to NetKAT-like languages with several of these features have been studied in previous work [110, 146].

Syntax. A packet π is a record mapping a finite set of fields f_1, f_2, \dots, f_k to bounded integers n . Fields include standard header fields such as the source (`src`) and destination (`dst`) of the packet, and two logical fields (`sw` for switch and `pt` for port) that record the current location of the packet in the network. The logical fields are not present in a physical network packet, but it is convenient to model them as proper header fields. We write $\pi.f$ to denote the value of field f of π and $\pi[f:=n]$ for the packet obtained from π by updating field f to n . We let Pk denote the (finite) set of all packets.

A history $h = \pi::\tilde{h}$ is a non-empty list of packets with *head packet* π and (possibly empty) *tail* \tilde{h} . The head packet models the packet's current state and the tail contains its prior states, which capture the trajectory of the packet through the network. Operationally, only the head packet exists, but it is useful to discriminate between identical packets with different histories. We write H to denote the (countable) set of all histories.

We differentiate between *predicates* (t, u) and *programs* (p, q) . The predicates form a Boolean algebra and include the primitives *false* (`drop`), *true* (`skip`), and *tests* $(f=n)$, as well as the standard Boolean operators disjunction $(t \& u)$, conjunction $(t \cdot u)$, and negation $(\neg t)$. Programs include *predicates* (t) and *modifications* $(f \leftarrow n)$ as primitives, and the operators *parallel composition* $(p \& q)$, *sequential composition* $(p \cdot q)$, and *iteration* (p^*) . The primitive `dup` records the current state of the packet by extending the tail with the head packet. Intuitively, we may think of a history as a log of a packet's activity, and of `dup` as the logging command. Finally, *choice* $p \oplus_r q$ executes p with probability r or q with probability $1 - r$. We write $p \oplus q$ when $r = 0.5$.

Predicate conjunction and sequential composition use the same syntax $(t \cdot u)$ as their semantics coincide (as we will see shortly). The same is true for disjunction of predicates and parallel composition $(t \& u)$. The distinction between *predicates* and *programs* is merely to restrict negation to predicates and rule out programs like $\neg(p^*)$.

Syntactic Sugar. The language as presented in Figure 3.2 is reduced to its core primitives. It is worth noting that many useful constructs can be derived from this core. In particular, it is straightforward to encode conditionals and while loops:

$$\mathbf{if } t \mathbf{ then } p \mathbf{ else } q := t \cdot p \ \& \ \neg t \cdot q$$

$$\mathbf{while } t \mathbf{ do } p := (t \cdot p)^* \cdot \neg t$$

These encodings are well-known from KAT [90]. While loops are useful for implementing higher level abstractions such as network virtualization in NetKAT [164].

Example. Consider the programs

$$p_1 := \text{pt}=1 \cdot (\text{pt}\leftarrow 2 \ \& \ \text{pt}\leftarrow 3)$$

$$p_2 := (\text{pt}=2 \ \& \ \text{pt}=3) \cdot \text{dst}\leftarrow 10.0.0.1 \cdot \text{pt}\leftarrow 1$$

The first program forwards packets entering at port 1 out of ports 2 and 3—a simple form of multicast—and drops all other packets. The second program matches on packets coming in on ports 2 *or* 3, modifies their destination to the IP address 10.0.0.1, and sends them out through port 1. The program $p_1 \ \& \ p_2$ acts like p_1 for packets entering at port 1, and like p_2 for packets entering at ports 2 or 3.

Monads. We define the semantics of NetKAT programs parametrically over a monad \mathcal{M} . This allows us to give two concrete semantics at once: the classical deterministic semantics (using the identity monad), and the new probabilistic semantics (using the probability monad). For simplicity, we refrain from giving a categorical treatment and simply model a monad in terms of three components:

- a constructor \mathcal{M} that lifts X to a domain $\mathcal{M}(X)$;
- an operator $\eta : X \rightarrow \mathcal{M}(X)$ that lifts objects into the domain $\mathcal{M}(X)$; and

- an infix operator

$$\ggg : \mathcal{M}(X) \rightarrow (X \rightarrow \mathcal{M}(X)) \rightarrow \mathcal{M}(X)$$

that lifts a function $f : X \rightarrow \mathcal{M}(X)$ to a function

$$(- \ggg f) : \mathcal{M}(X) \rightarrow \mathcal{M}(X)$$

These components must satisfy three axioms:

$$\eta(a) \ggg f = f(a) \tag{M1}$$

$$m \ggg \eta = m \tag{M2}$$

$$(m \ggg f) \ggg g = m \ggg (\lambda x. f(x) \ggg g) \tag{M3}$$

The semantics of deterministic programs (not containing probabilistic choices $p \oplus_r q$) uses as underlying objects the set of packet histories 2^H and the identity monad $\mathcal{M}(X) = X$: η is the identify function and $x \ggg f$ is simply function application $f(x)$. The identity monad trivially satisfies the three axioms.

The semantics of probabilistic programs uses the probability (or Giry) monad [52, 72, 142] $\mathcal{D}(-)$, that maps a measurable space to the domain of probability measures over that space. The operator η maps a to the point mass (or Dirac measure) δ_a on a . Composition $\mu \ggg (\lambda a. \nu_a)$ can be thought of as a two-stage probabilistic experiment where the second experiment ν_a depends on the outcome a of the first experiment μ . Operationally, we first sample from μ to obtain a random outcome a ; then, we sample from ν_a to obtain the final outcome b . What is the distribution over final outcomes? It can be obtained by observing that $\lambda a. \nu_a$ is a Markov kernel (§3.3), and so composition with μ is given by the familiar integral

$$\mu \ggg (\lambda a. \nu_a) = \lambda A. \int_{a \in X} \nu_a(A) \cdot \mu(da)$$

introduced in (3.1). It is well known that these definitions satisfy the monad axioms [52, 72, 87]. (M1) and (M2) are trivial properties of the Lebesgue Integral. (M3) is

essentially Fubini’s theorem, which permits changing the order of integration in a double integral.

Deterministic Semantics. In deterministic NetKAT (without $p \oplus_r q$), a program p denotes a function $\llbracket p \rrbracket \in 2^H \rightarrow 2^H$ mapping a set of input histories $a \in 2^H$ to a set of output histories $\llbracket p \rrbracket(a)$. Note that the input and output sets do *not* encode non-determinism but represent sets of “in-flight” packets in the network. Histories record the processing done to each packet as it traverses the network. In particular, histories enable reasoning about path properties and determining which outputs were generated from common inputs.

Formally, a predicate t maps the input set a to the subset $b \subseteq a$ of histories satisfying the predicate. In particular, the false primitive drop denotes the function mapping any input to the empty set; the true primitive skip is the identity function; the test $f=n$ retains those histories with field f of the head packet equal to n ; and negation $\neg t$ returns only those histories not satisfying t . Modification $f \leftarrow n$ sets the f -field of all head-packets to the value n . Duplication dup extends the tails of all input histories with their head packets, thus permanently recording the current state of the packets.

Parallel composition $p \& q$ feeds the input to both p and q and takes the union of their outputs. If p and q are predicates, a history is thus in the output iff it satisfies at least one of p or q , so that union acts like logical disjunction on predicates. Sequential composition $p \cdot q$ feeds the input to p and then feeds p ’s output to q to produce the final result. If p and q are predicates, a history is thus in the output iff it satisfies both p and q , acting like logical conjunction. Iteration p^* behaves like the parallel composition of p sequentially composed with itself zero or more times (because \sqcup is union in 2^H).

Probabilistic Semantics. The semantics of ProbNetKAT is given using the probability monad $\mathcal{D}(-)$ applied to the set of history sets 2^H (seen as a measurable space). A

program p denotes a function

$$\llbracket p \rrbracket \in 2^{\mathbb{H}} \rightarrow \{\mu : \mathcal{B} \rightarrow [0, 1] \mid \mu \text{ is a probability measure}\}$$

mapping a set of input histories a to a *distribution* over output sets $\llbracket p \rrbracket(a)$. Here, \mathcal{B} denotes the Borel sets of $2^{\mathbb{H}}$ (§3.5). Equivalently, $\llbracket p \rrbracket$ is a Markov kernel with source and destination $(2^{\mathbb{H}}, \mathcal{B})$. The semantics of all primitive programs is identical to the deterministic case, except that they now return point masses on output sets (rather than just output sets). In fact, it follows from **(M1)** that all programs without choices and iteration are point masses.

Parallel composition $p \& q$ feeds the input a to p and q , samples b_1 and b_2 from the output distributions $\llbracket p \rrbracket(a)$ and $\llbracket q \rrbracket(a)$, and returns the union of the samples $b_1 \cup b_2$. Probabilistic choice $p \oplus_r q$ feeds the input to both p and q and returns a convex combination of the output distributions according to r . Sequential composition $p \cdot q$ is just sequential composition of Markov kernels. Operationally, it feeds the input to p , obtains a sample b from p 's output distribution, and feeds the sample to q to obtain the final distribution. Iteration p^* is defined as the least fixpoint of the map on Markov kernels $X \mapsto 1 \& \llbracket p \rrbracket; X$, which is continuous in a DCPO that we will develop in the following sections. We will show that this definition, which is simple and is based on standard techniques from domain theory, coincides with the semantics proposed in previous work [44].

Basic Properties. To clarify the nature of predicates and other primitives, we establish two intuitive properties:

Lemma 3.4.1. *Any predicate t satisfies $\llbracket t \rrbracket(a) = \eta(a \cap b_t)$, where $b_t := \llbracket t \rrbracket(\mathbb{H})$ in the identity monad.*

Proof. By induction on t , using **(M1)** in the induction step. □

Lemma 3.4.2. *All atomic programs p (i.e., predicates, dup, and modifications) satisfy*

$$\llbracket p \rrbracket(a) = \eta(\{f_p(h) \mid h \in a\})$$

for some partial function $f_p : H \rightarrow H$.

Proof. Immediate from Figure 3.2 and Lemma 3.4.1. □

Lemma 3.4.1 captures the intuition that predicates act like packet *filters*. Lemma 3.4.2 establishes that the behavior of atomic programs is captured by their behavior on individual histories.

Note however that ProbNetKAT’s semantic domain is rich enough to model interactions between packets. For example, it would be straightforward to extend the language with new primitives whose behavior depends on properties of the input set of packet histories—e.g., a rate-limiting construct $@_n$ that selects at most n packets uniformly at random from the input and drops all other packets. Our results continue to hold when the language is extended with arbitrary continuous Markov kernels of appropriate type, or continuous operations on such kernels.

Another important observation is that although ProbNetKAT does not include continuous distributions as primitives, there are programs that generate continuous distributions by combining choice and iteration:

Lemma 3.4.3 (Theorem 3 in Foster et al. [44]). *Let π_0, π_1 denote distinct packets. Let p denote the program that changes the head packet of all inputs to either π_0 or π_1 with equal probability. Then*

$$\llbracket p \cdot (\text{dup} \cdot p)^* \rrbracket(\{\pi\}, -)$$

is a continuous distribution.

Hence, ProbNetKAT programs cannot be modeled by functions of type $2^H \rightarrow (2^H \rightarrow [0, 1])$ in general. We need to define a measure space over 2^H and consider general probability measures.

3.5 Cantor Meets Scott

To define continuous probability measures on an infinite set X , one first needs to endow X with a topology—some additional structure that, intuitively, captures which elements of X are close to each other or approximate each other. Although the choice of topology is arbitrary in principle, different topologies induce different notions of continuity and limits, thus profoundly impacting the concepts derived from these primitives. Which topology is the “right” one for 2^H ? A fundamental contribution of this chapter is to show that there are (at least) two answers to this question:

- The initial work on ProbNetKAT [44] uses the *Cantor topology*. This makes 2^H a *standard Borel space*, which is well-studied and known to enjoy many desirable properties.
- This chapter is based on the *Scott topology*, the standard choice of domain theorists. Although this topology is weaker in the sense that it lacks much of the useful structure and properties of a standard Borel space, it leads to a simpler and more computational account of ProbNetKAT’s semantics.

Despite this, one view is not better than the other. The main advantage of the Cantor topology is that it allows us to reason in terms of a metric. With the Scott topology, we sacrifice this metric, but in return we are able to interpret all program operators and programs as continuous functions. The two views yield different convergence theorems, both of which are useful. Remarkably, we can have the best of both worlds: it turns out that the two topologies generate the same Borel sets, so the probability measures are the same regardless. We will prove (Theorem 3.7.7) that the semantics in Figure 3.2 coincides with the original semantics [44], recovering all the results from previous work. This allows us to freely switch between the two views as convenient. The rest of this section illustrates the difference between the two topologies intuitively, defines

the topologies formally and endows 2^H with Borel sets, and proves a general theorem relating the two.

Cantor and Scott, intuitively. The Cantor topology is best understood in terms of a distance $d(a, b)$ of history sets a, b , formally known as a *metric*. Define this metric as $d(a, b) = 2^{-n}$, where n is the length of the shortest packet history in the symmetric difference of a and b if $a \neq b$, or $d(a, b) = 0$ if $a = b$. Intuitively, history sets are close if they differ only in very long histories. This gives the following notions of limit and continuity:

- a is the limit of a sequence a_1, a_2, \dots iff the distance $d(a, a_n)$ approaches 0 as $n \rightarrow \infty$.
- a function $f : 2^H \rightarrow [0, \infty]$ is continuous at point a iff $f(a_n)$ approaches $f(a)$ whenever a_n approaches a .

The Scott topology cannot be described in terms of a metric. It is captured by a complete partial order $(2^H, \sqsubseteq)$ on history sets. If we choose the subset order (with suprema given by union) we obtain the following notions:

- a is the limit of a sequence $a_1 \subseteq a_2 \subseteq \dots$ iff $a = \bigcup_{n \in \mathbb{N}} a_n$.
- a function $f : 2^H \rightarrow [0, \infty]$ is continuous at point a iff $f(a) = \sup_{n \in \mathbb{N}} f(a_n)$ whenever a is the limit of $a_1 \subseteq a_2 \subseteq \dots$.

Example. To illustrate the difference between Cantor-continuity and Scott-continuity, consider the function $f(a) := |a|$ that maps a history set to its (possibly infinite) cardinality. The function is not Cantor-continuous. To see this, let h_n denote a history of length n and consider the sequence of singleton sets $a_n := \{h_n\}$. Then $d(a_n, \emptyset) = 2^{-n}$, *i.e.* the sequence approaches the empty set as n approaches infinity. But the cardinality $|a_n| = 1$ does *not* approach $|\emptyset| = 0$. In contrast, the function is easily seen to be Scott-continuous.

As a second example, consider the function $f(a) := 2^{-k}$, where k is the length of the smallest history *not* in a . This function is Cantor-continuous: if $d(a_n, a) = 2^{-n}$, then

$$|f(a_n) - f(a)| \leq 2^{-(n-1)} - 2^{-n} \leq 2^{-n}$$

Therefore $f(a_n)$ approaches $f(a)$ as the distance $d(a_n, a)$ approaches 0. However, the function is not Scott-continuous¹, as all Scott-continuous functions are monotone.

Approximation. The computational importance of limits and continuity comes from the following idea. Assume a is some complicated (say infinite) mathematical object. If a_1, a_2, \dots is a sequence of simple (say finite) objects with limit a , then it may be possible to approximate a using the sequence (a_n) . This gives us a computational way of working with infinite objects, even though the available resources may be fundamentally finite. Continuity captures precisely when this is possible: we can perform a computation f on a if f is continuous in a , for then we can compute the sequence $f(a_1), f(a_2), \dots$ which (by continuity) converges to $f(a)$.

We will show later that any measure μ can be approximated by a sequence of finite measures μ_1, μ_2, \dots , and that the expected value $\mathbf{E}_\mu[f]$ of a Scott-continuous random variable f is continuous with respect to the measure. Our implementation exploits this to compute a monotonically improving sequence of approximations for performance metrics such as latency and congestion (§3.9).

Notation. We use lower case letters $a, b, c \subseteq \mathbb{H}$ to denote history sets, uppercase letters $A, B, C \subseteq 2^{\mathbb{H}}$ to denote measurable sets (*i.e.*, sets of history sets), and calligraphic letters $\mathcal{B}, \mathcal{O}, \dots \subseteq 2^{2^{\mathbb{H}}}$ to denote sets of measurable sets. For a set X , we let $\wp_\omega(X) := \{Y \subseteq X \mid |Y| < \infty\}$ denote the finite subsets of X and $\mathbf{1}_X$ the characteristic function of X . For a statement φ , such as $a \subseteq b$, we let $[\varphi]$ denote 1 if φ is true and 0 otherwise.

¹with respect to the orders \subseteq on $2^{\mathbb{H}}$ and \leq on \mathbb{R}

Cantor and Scott, formally. For $h \in H$ and $b \in 2^H$, define

$$B_h := \{c \mid h \in c\} \qquad B_b := \bigcap_{h \in b} B_h = \{c \mid b \subseteq c\}. \quad (3.3)$$

The Cantor space topology, denoted \mathcal{C} , can be generated by closing $\{B_h, \sim B_h \mid h \in H\}$ under finite intersection and arbitrary union. The Scott topology of the DCPO $(2^H, \subseteq)$, denoted \mathcal{O} , can be generated by closing $\{B_h \mid h \in H\}$ under the same operations and adding the empty set. The Borel algebra \mathcal{B} is the smallest σ -algebra containing the Cantor-open sets, *i.e.* $\mathcal{B} := \sigma(\mathcal{C})$. We write \mathcal{B}_b for the Boolean subalgebra of \mathcal{B} generated by $\{B_h \mid h \in b\}$.

Lemma 3.5.1.

- (i) $b \subseteq c \Leftrightarrow B_c \subseteq B_b$
- (ii) $B_b \cap B_c = B_{b \cup c}$
- (iii) $B_\emptyset = 2^H$
- (iv) $\mathcal{B}_H = \bigcup_{b \in \wp_\omega(H)} \mathcal{B}_b$.

Note that if b is finite, then so is \mathcal{B}_b . Moreover, the atoms of \mathcal{B}_b are in one-to-one correspondence with the subsets $a \subseteq b$. The subsets a determine which of the B_h occur positively in the construction of the atom,

$$\begin{aligned} A_{ab} &:= \bigcap_{h \in a} B_h \cap \bigcap_{h \in b-a} \sim B_h \\ &= B_a - \bigcup_{a \subset c \subseteq b} B_c = \{c \in 2^H \mid c \cap b = a\}, \end{aligned} \quad (3.4)$$

where \subset denotes proper subset. The atoms A_{ab} are the basic open sets of the Cantor space. The notation A_{ab} is reserved for such sets.

Lemma 3.5.2 (Figure 3.3). *For b finite and $a \subseteq b$, $B_a = \bigcup_{a \subset c \subseteq b} A_{cb}$.*

Proof. By (3.4),

$$\begin{aligned} \bigcup_{a \subseteq c \subseteq b} A_{cb} &= \bigcup_{a \subseteq c \subseteq b} \{d \in 2^{\mathbb{H}} \mid d \cap b = c\} \\ &= \{d \in 2^{\mathbb{H}} \mid a \subseteq d\} = B_a. \end{aligned} \quad \square$$

Scott Topology Properties. Let \mathcal{O} denote the family of Scott-open sets of $(2^{\mathbb{H}}, \subseteq)$. Following are some facts about this topology.

- The DCPO $(2^{\mathbb{H}}, \subseteq)$ is algebraic. The finite elements of $2^{\mathbb{H}}$ are the finite subsets $a \in \wp_{\omega}(\mathbb{H})$, and their up-closures are $\{a\}\uparrow = B_a$.
- By Lemma 3.3.1(ii), the up-closures $\{a\}\uparrow = B_a$ form a base for the Scott topology. The sets B_h for $h \in \mathbb{H}$ are therefore a subbase.
- Thus, a subset $B \subseteq 2^{\mathbb{H}}$ is Scott-open iff there exists $F \subseteq \wp_{\omega}(\mathbb{H})$ such that $B = \bigcup_{a \in F} B_a$.
- The Scott topology is weaker than the Cantor space topology, e.g., $\sim B_h$ is Cantor-open but not Scott-open. However, the Borel sets of the topologies are the same, as $\sim B_h$ is a Π_1^0 Borel set.²
- Although any Scott-open set in $2^{\mathbb{H}}$ is also Cantor-open, a Scott-continuous function $f : 2^{\mathbb{H}} \rightarrow \mathbb{R}_+$ is not necessarily Cantor-continuous. This is because for Scott-continuity we consider \mathbb{R}_+ (ordered by \leq) with the Scott topology, but for Cantor-continuity we consider \mathbb{R}_+ with the standard Euclidean topology.
- Any Scott-continuous function $f : 2^{\mathbb{H}} \rightarrow \mathbb{R}_+$ is measurable, because the Scott-open sets of (\mathbb{R}_+, \leq) (i.e., the upper semi-infinite intervals $(r, \infty] = \{r\}\uparrow$ for $r \geq 0$) generate the Borell sets on \mathbb{R}_+ .

²References to the Borel hierarchy Σ_n^0 and Π_n^0 refer to the Scott topology. The Cantor and Scott topologies have different Borel hierarchies.

- The open sets \mathcal{O} ordered by the subset relation forms an ω -complete lattice with bottom \emptyset and top $B_\emptyset = 2^H$.
- The finite sets $a \in \wp_\omega(H)$ are dense and countable, thus the space is separable.
- The Scott topology is not Hausdorff, metrizable, or compact. It is not Hausdorff, as any nonempty open set contains H , but it satisfies the weaker T_0 separation property: for any pair of points a, b with $a \not\subseteq b$, $a \in B_a$ but $b \notin B_a$.
- There is an up-closed Π_2^0 Borel set with an uncountable set of minimal elements.
- There are up-closed Borel sets with no minimal elements; for example, the family of cofinite subsets of H , a Σ_3^0 Borel set.
- The compact-open sets are those of the form F^\uparrow , where F is a finite set of finite sets. There are plenty of open sets that are not compact-open, e.g. $B_\emptyset - \{\emptyset\} = \bigcup_{h \in H} B_h$.

Lemma 3.5.3 (see Halmos [60, Theorem III.13.A]). *Any probability measure is uniquely determined by its values on B_b for b finite.*

Proof. For b finite, the atoms of \mathcal{B}_b are of the form (3.4). By the inclusion-exclusion principle (see Figure 3.3),

$$\mu(A_{ab}) = \mu(B_a - \bigcup_{a \subset c \subseteq b} B_c) = \sum_{a \subseteq c \subseteq b} (-1)^{|c-a|} \mu(B_c). \quad (3.5)$$

Thus μ is uniquely determined on the atoms of \mathcal{B}_b and therefore on \mathcal{B}_b . As \mathcal{B}_H is the union of the \mathcal{B}_b for finite b , μ is uniquely determined on \mathcal{B}_H . By the monotone class theorem, the Borel sets \mathcal{B} are the smallest monotone class containing \mathcal{B}_H , and since $\mu(\bigcup_n A_n) = \sup_n \mu(A_n)$ and $\mu(\bigcap_n A_n) = \inf_n \mu(A_n)$, we have that μ is determined on all Borel sets. \square

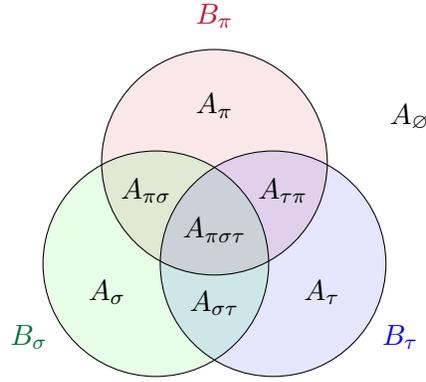


Figure 3.3: Relationship of the basic Scott-open sets B_a to the basic Cantor-open sets A_{ab} for $b = \{\pi, \sigma, \tau\}$ and $a \subseteq b$. The regions labeled $A_\emptyset, A_\pi, A_{\pi\sigma}$, etc. represent the basic Cantor-open sets $A_{\emptyset,b}, A_{\{\pi\},b}, A_{\{\pi,\sigma\},b}$, etc. These are the atoms of the Boolean algebra \mathcal{B}_b . Several basic Scott-open sets are not shown, e.g. $B_{\{\pi,\sigma\}} = B_\pi \cap B_\sigma = A_{\{\pi,\sigma\},b} \cup A_{\{\pi,\sigma,\tau\},b}$.

Extension Theorem. We now prove a useful extension theorem (Theorem 3.5.4) that identifies necessary and sufficient conditions for extending a function $\mathcal{O} \rightarrow [0, 1]$ defined on the Scott-open sets of 2^H to a measure $\mathcal{B} \rightarrow [0, 1]$. The theorem yields a remarkable linear correspondence between the Cantor and Scott topologies (Theorem 3.5.6). We prove it for 2^H only, but generalizations may be possible.

Theorem 3.5.4. A function $\mu : \{B_b \mid b \text{ finite}\} \rightarrow [0, 1]$ extends to a measure $\mu : \mathcal{B} \rightarrow [0, 1]$ if and only if for all finite b and all $a \subseteq b$,

$$\sum_{a \subseteq c \subseteq b} (-1)^{|c-a|} \mu(B_c) \geq 0.$$

Moreover, the extension to \mathcal{B} is unique.

Proof. The condition is clearly necessary by (3.5). For sufficiency and uniqueness, we use the Carathéodory extension theorem. For each atom A_{ab} of \mathcal{B}_b , $\mu(A_{ab})$ is already determined uniquely by (3.5) and nonnegative by assumption. For each $B \in \mathcal{B}_b$, write B uniquely as a union of atoms and define $\mu(B)$ to be the sum of the $\mu(A_{ab})$ for all atoms A_{ab} of \mathcal{B}_b contained in B . We must show that $\mu(B)$ is well-defined. Note that the definition is given in terms of b , and we must show that the definition is independent of

the choice of b . It suffices to show that the calculation using atoms of $b' = b \cup \{h\}$, $h \notin b$, gives the same result. Each atom of \mathcal{B}_b is the disjoint union of two atoms of $\mathcal{B}_{b'}$:

$$A_{ab} = A_{a \cup \{h\}, b \cup \{h\}} \cup A_{a, b \cup \{h\}}$$

It suffices to show the sum of their measures is the measure of A_{ab} :

$$\begin{aligned} \mu(A_{a, b \cup \{h\}}) &= \sum_{a \subseteq c \subseteq b \cup \{h\}} (-1)^{|c-a|} \mu(B_c) \\ &= \sum_{a \subseteq c \subseteq b} (-1)^{|c-a|} \mu(B_c) + \sum_{a \cup \{h\} \subseteq c \subseteq b \cup \{h\}} (-1)^{|c-a|} \mu(B_c) \\ &= \mu(A_{ab}) - \mu(A_{a \cup \{h\}, b \cup \{h\}}). \end{aligned}$$

To apply the Carathéodory extension theorem, we must show that μ is countably additive, *i.e.* that $\mu(\bigcup_n A_n) = \sum_n \mu(A_n)$ for any countable sequence $A_n \in \mathcal{B}_H$ of pairwise disjoint sets whose union is in \mathcal{B}_H . For finite sequences $A_n \in \mathcal{B}_H$, write each A_n uniquely as a disjoint union of atoms of \mathcal{B}_b for some sufficiently large b such that all $A_n \in \mathcal{B}_b$. Then $\bigcup_n A_n \in \mathcal{B}_b$, the values of the atoms are given by (3.5), and the value of $\mu(\bigcup_n A_n)$ is well-defined and equal to $\sum_n \mu(A_n)$. We cannot have an infinite set of pairwise disjoint nonempty $A_n \in \mathcal{B}_H$ whose union is in \mathcal{B}_H by compactness. All elements of \mathcal{B}_H are clopen in the Cantor topology. If $\bigcup_n A_n = A \in \mathcal{B}_H$, then $\{A_n \mid n \geq 0\}$ would be an open cover of A with no finite subcover. \square

Cantor Meets Scott. We now establish a correspondence between the Cantor and Scott topologies on 2^H . Consider the infinite triangular matrix E and its inverse E^{-1} with rows and columns indexed by the finite subsets of H , where

$$E_{ac} = [a \subseteq c] \qquad E_{ac}^{-1} = (-1)^{|c-a|} [a \subseteq c].$$

These matrices are indeed inverses: For $a, d \in \wp_\omega(\mathbb{H})$,

$$\begin{aligned} (E \cdot E^{-1})_{ad} &= \sum_c E_{ac} \cdot E_{cd}^{-1} \\ &= \sum_c [a \subseteq c] \cdot [c \subseteq d] \cdot (-1)^{|d-c|} \\ &= \sum_{a \subseteq c \subseteq d} (-1)^{|d-c|} = [a = d], \end{aligned}$$

thus $E \cdot E^{-1} = I$, and similarly $E^{-1} \cdot E = I$.

Recall that the Cantor basic open sets are the elements A_{ab} for b finite and $a \subseteq b$. Those for fixed finite b are the atoms of the Boolean algebra \mathcal{B}_b . They form the basis of a $2^{|b|}$ -dimensional linear space. The Scott basic open sets B_a for $a \subseteq b$ are another basis for the same space. The two bases are related by the matrix $E[b]$, the $2^b \times 2^b$ submatrix of E with rows and columns indexed by subsets of b . One can show that the finite matrix $E[b]$ is invertible with inverse $E[b]^{-1} = (E^{-1})[b]$.

Lemma 3.5.5. *Let μ be a measure on $2^{\mathbb{H}}$ and $b \in \wp_\omega(\mathbb{H})$. Let X, Y be vectors indexed by subsets of b such that $X_a = \mu(B_a)$ and $Y_a = \mu(A_{ab})$ for $a \subseteq b$. Let $E[b]$ be the $2^b \times 2^b$ submatrix of E . Then $X = E[b] \cdot Y$.*

The matrix-vector equation $X = E[b] \cdot Y$ captures the fact that for $a \subseteq b$, B_a is the disjoint union of the atoms A_{cb} of \mathcal{B}_b for $a \subseteq c \subseteq b$ (see Figure 3.3), and consequently $\mu(B_a)$ is the sum of $\mu(A_{cb})$ for these atoms. The inverse equation $X = E[b]^{-1} \cdot Y$ captures the inclusion-exclusion principle for \mathcal{B}_b .

In fact, more can be said about the structure of E . For any $b \in 2^{\mathbb{H}}$, finite or infinite, let $E[b]$ be the submatrix of E with rows and columns indexed by the subsets of b . If $a \cap b = \emptyset$, then $E[a \cup b] = E[a] \otimes E[b]$, where \otimes denotes Kronecker product. The formation of the Kronecker product requires a notion of pairing on indices, which in our case is

given by disjoint set union. For example,

$$E[\{h_1\}] = \begin{array}{c} \emptyset \quad \{h_1\} \\ \emptyset \quad \left[\begin{array}{cc} 1 & 1 \\ 0 & 1 \end{array} \right] \\ \{h_1\} \end{array} \quad E[\{h_2\}] = \begin{array}{c} \emptyset \quad \{h_2\} \\ \emptyset \quad \left[\begin{array}{cc} 1 & 1 \\ 0 & 1 \end{array} \right] \\ \{h_2\} \end{array}$$

$$E[\{h_1, h_2\}] = E[\{h_1\}] \otimes E[\{h_2\}] = \begin{array}{c} \emptyset \quad \{h_1\} \quad \{h_2\} \quad \{h_1, h_2\} \\ \emptyset \quad \left[\begin{array}{cccc} 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{array} \right] \\ \{h_1\} \\ \{h_2\} \\ \{h_1, h_2\} \end{array}$$

As $(E \otimes F)^{-1} = E^{-1} \otimes F^{-1}$ for Kronecker products of invertible matrices, we also have

$$E[\{h_1\}]^{-1} = \begin{bmatrix} 1 & -1 \\ 0 & 1 \end{bmatrix} \quad E[\{h_2\}]^{-1} = \begin{bmatrix} 1 & -1 \\ 0 & 1 \end{bmatrix}$$

$$E[\{h_1, h_2\}]^{-1} = E[\{h_1\}]^{-1} \otimes E[\{h_2\}]^{-1} = \begin{bmatrix} 1 & -1 & -1 & 1 \\ 0 & 1 & 0 & -1 \\ 0 & 0 & 1 & -1 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

E can be viewed as the infinite Kronecker product $\bigotimes_{h \in \mathcal{H}} E[\{h\}]$.

Theorem 3.5.6. *The probability measures on $(2^{\mathcal{H}}, \mathcal{B})$ are in one-to-one correspondence with pairs of matrices $M, N \in \mathbb{R}^{\mathcal{Q}_\omega(\mathcal{H}) \times \mathcal{Q}_\omega(\mathcal{H})}$ such that*

- (i) M is diagonal with entries in $[0, 1]$,
- (ii) N is nonnegative, and
- (iii) $N = E^{-1} M E$.

The correspondence associates the measure μ with the matrices

$$N_{ab} = \mu(A_{ab}) \quad M_{ab} = [a = b] \cdot \mu(B_a). \quad (3.6)$$

3.6 A DCPO on Markov Kernels

In this section we define a continuous DCPO on Markov kernels. Proofs omitted from this section can be found in the appendix.

We will interpret all program operators defined in Figure 3.2 also as operators on Markov kernels: for an operator $\llbracket p \otimes q \rrbracket$ defined on programs p and q , we obtain a definition of $P \otimes Q$ on Markov kernels P and Q by replacing $\llbracket p \rrbracket$ with P and $\llbracket q \rrbracket$ with Q in the original definition. Additionally we define $\&$ on probability measures as follows:

$$(\mu \& \nu)(A) := (\mu \times \nu)(\{(a, b) \mid a \cup b \in A\})$$

The corresponding operation on programs and kernels as defined in Figure 3.2 can easily be shown to be equivalent to a pointwise lifting of the definition here.

For measures μ, ν on 2^H , define $\mu \sqsubseteq \nu$ if $\mu(B) \leq \nu(B)$ for all $B \in \mathcal{O}$. This order was first defined by Saheb-Djahromi [151].

Theorem 3.6.1 ([151]). *The probability measures on the Borel sets generated by the Scott topology of an algebraic DCPO ordered by \sqsubseteq form a DCPO.*

Because $(2^H, \sqsubseteq)$ is an algebraic DCPO, Theorem 3.6.1 applies.³ In this case, the bottom and top elements are δ_\emptyset and δ_H respectively.

Lemma 3.6.2. $\mu \sqsubseteq \mu \& \nu$ and $\nu \sqsubseteq \mu \& \nu$.

Surprisingly, despite Lemma 3.6.2, the probability measures do not form an upper semilattice under \sqsubseteq , although counterexamples are somewhat difficult to construct. See the appendix for an example.

Next we lift the order \sqsubseteq to Markov kernels $P : 2^H \times \mathcal{B} \rightarrow [0, 1]$. The order is defined pointwise on kernels regarded as functions $2^H \times \mathcal{O} \rightarrow [0, 1]$; that is,

$$P \sqsubseteq Q \iff \forall a \in 2^H. \forall B \in \mathcal{O}. P(a, B) \leq Q(a, B).$$

³A beautiful proof based on Theorem 3.5.4 can be found in the appendix.

There are several ways of viewing the lifted order \sqsubseteq , as shown in the next lemma.

Lemma 3.6.3. *The following are equivalent:*

- (i) $P \sqsubseteq Q$, i.e., $\forall a \in 2^H$ and $B \in \mathcal{O}$, $P(a, B) \leq Q(a, B)$;
- (ii) $\forall a \in 2^H$, $P(a, -) \sqsubseteq Q(a, -)$ in the DCPO $\mathcal{D}(2^H)$;
- (iii) $\forall B \in \mathcal{O}$, $P(-, B) \sqsubseteq Q(-, B)$ in the DCPO $2^H \rightarrow [0, 1]$;
- (iv) $\text{curry } P \sqsubseteq \text{curry } Q$ in the DCPO $2^H \rightarrow \mathcal{D}(2^H)$.

A Markov kernel $P : 2^H \times \mathcal{B} \rightarrow [0, 1]$ is *continuous* if it is Scott-continuous in its first argument; i.e., for any fixed $A \in \mathcal{O}$, $P(a, A) \leq P(b, A)$ whenever $a \subseteq b$, and for any directed set $D \subseteq 2^H$ we have $P(\bigcup D, A) = \sup_{a \in D} P(a, A)$. This is equivalent to saying that $\text{curry } P : 2^H \rightarrow \mathcal{D}(2^H)$ is Scott-continuous as a function from the DCPO 2^H ordered by \subseteq to the DCPO of probability measures ordered by \sqsubseteq . We will show later that all ProbNetKAT programs give rise to continuous kernels.

Theorem 3.6.4. *The continuous kernels $P : 2^H \times \mathcal{B} \rightarrow [0, 1]$ ordered by \sqsubseteq form a continuous DCPO with basis consisting of kernels of the form $b \cdot P \cdot d$ for P an arbitrary continuous kernel and b, d filters on finite sets b and d ; that is, kernels that drop all input packets except for those in b and all output packets except those in d .*

It is not true that the space of continuous kernels is algebraic with finite elements $b \cdot P \cdot d$. See the appendix for a counterexample.

3.7 Continuity and Semantics of Iteration

This section develops the technology needed to establish that all ProbNetKAT programs give continuous Markov kernels and that all program operators are themselves continuous. These results are needed for the least fixpoint characterization of iteration and also pave the way for our approximation results (§3.8).

The key fact that underpins these results is that Lebesgue integration respects the orders on measures and on functions:

Theorem 3.7.1. *Integration is Scott-continuous in both arguments:*

(i) *For any Scott-continuous function $f : 2^H \rightarrow [0, \infty]$, the map*

$$\mu \mapsto \int f d\mu \quad (3.7)$$

is Scott-continuous with respect to the order \sqsubseteq on $\mathcal{M}(2^H)$.

(ii) *For any probability measure μ , the map*

$$f \mapsto \int f d\mu \quad (3.8)$$

is Scott-continuous with respect to the order on $[2^H \rightarrow [0, \infty]]$.

The proofs of the remaining results in this section are somewhat long and mostly routine, but can be found in the appendix.

Theorem 3.7.2. *The deterministic kernels associated with any Scott-continuous function $f : D \rightarrow E$ are continuous, and the following operations on kernels preserve continuity: product, integration, sequential composition, parallel composition, choice, iteration.*

The above theorem implies that $Q \mapsto \text{skip} \ \& \ P \cdot Q$ is a continuous map on the DCPO of continuous Markov kernels. Hence $P^* = \bigsqcup_n P^{(n)}$ is well-defined as the least fixed point of that map.

Corollary 3.7.3. *Every ProbNetKAT program denotes a continuous Markov kernel.*

The next theorem is the key result that enables a practical implementation:

Theorem 3.7.4. *The following semantic operations are continuous functions of the DCPO of continuous kernels: product, parallel composition, curry, sequential composition, choice, iteration. (Figure 3.4.)*

$$\begin{aligned}
\left(\bigsqcup_{n \geq 0} P_n\right) \& Q &= \bigsqcup_{n \geq 0} (P_n \& Q) & \left(\bigsqcup_{n \geq 0} P_n\right) \cdot Q &= \bigsqcup_{n \geq 0} (P_n \cdot Q) \\
\left(\bigsqcup_{n \geq 0} P_n\right) \oplus_r Q &= \bigsqcup_{n \geq 0} (P_n \oplus_r Q) & Q \cdot \left(\bigsqcup_{n \geq 0} P_n\right) &= \bigsqcup_{n \geq 0} (Q \cdot P_n) \\
\left(\bigsqcup_{n \geq 0} P_n\right)^* &= \bigsqcup_{n \geq 0} (P_n^*)
\end{aligned}$$

Figure 3.4: Scott-Continuity of program operators (Theorem 3.7.4).

The semantics of iteration presented in [44], defined in terms of an infinite process, coincides with the least fixpoint semantics presented here. The key observation is the relationship between weak convergence in the Cantor topology and fixpoint convergence in the Scott topology:

Theorem 3.7.5. *Let A be a directed set of probability measures with respect to \sqsubseteq and let $f : 2^{\mathbb{H}} \rightarrow [0, 1]$ be a Cantor-continuous function. Then*

$$\lim_{\mu \in A} \int_{c \in 2^{\mathbb{H}}} f(c) \cdot d\mu = \int_{c \in 2^{\mathbb{H}}} f(c) \cdot d(\bigsqcup A).$$

This theorem implies that $P^{(n)}$ weakly converges to P^* in the Cantor topology. [44] showed that $P^{(n)}$ also weakly converges to P^{\circledast} in the Cantor topology, where we let P^{\circledast} denote the iterate of P as defined in [44]. But since $(2^{\mathbb{H}}, \mathcal{C})$ is a Polish space, this implies that $P^* = P^{\circledast}$.

Lemma 3.7.6. *In a Polish space D , the values of*

$$\int_{a \in D} f(a) \cdot \mu(da)$$

for continuous $f : D \rightarrow [0, 1]$ determine μ uniquely.

Corollary 3.7.7. $P^{\circledast} = \bigsqcup_n P^{(n)} = P^*$.

3.8 Approximation

We now formalize a notion of approximation for ProbNetKAT programs. Given a program p , we define the n -th approximant $[p]_n$ inductively as

$$\begin{aligned} [p]_n &:= p \quad (\text{for } p \text{ primitive}) \\ [q \oplus_r r]_n &:= [q]_n \oplus_r [r]_n \\ [q \&r]_n &:= [q]_n \& [r]_n \\ [q \cdot r]_n &:= [q]_n \cdot [r]_n \\ [q^*]_n &:= ([q]_n)^{(n)} \end{aligned}$$

Intuitively, $[p]_n$ is just p where iteration $-^*$ is replaced by bounded iteration $-^{(n)}$. Let $\llbracket p \rrbracket_n$ denote the Markov kernel obtained from the n -th approximant: $\llbracket [p]_n \rrbracket$.

Theorem 3.8.1. *The approximants of a program p form a \sqsubseteq -increasing chain with supremum p , that is*

$$\llbracket p \rrbracket_1 \sqsubseteq \llbracket p \rrbracket_2 \sqsubseteq \dots \quad \text{and} \quad \bigsqcup_{n \geq 0} \llbracket p \rrbracket_n = \llbracket p \rrbracket$$

Proof. By induction on p and continuity of the operators. □

This means that any program can be approximated by a sequence of star-free programs, which—in contrast to general programs (Lemma 3.4.3)—can only produce finite distributions. These finite distributions are sufficient to compute the expected values of Scott-continuous random variables:

Corollary 3.8.2. *Let $\mu \in \mathcal{M}(2^H)$ be an input distribution, p be a program, and $Q : 2^H \rightarrow [0, \infty]$ be a Scott-continuous random variable. Let*

$$\nu := \mu \gg \llbracket p \rrbracket \quad \text{and} \quad \nu_n := \mu \gg \llbracket p \rrbracket_n$$

denote the output distribution and its approximations. Then

$$\mathbf{E}_{\nu_0}[Q] \leq \mathbf{E}_{\nu_1}[Q] \leq \dots \quad \text{and} \quad \sup_{n \in \mathbb{N}} \mathbf{E}_{\nu_n}[Q] = \mathbf{E}_{\nu}[Q]$$

Proof. Follows directly from Theorems 3.8.1 and 3.7.1. □

Note that the approximations ν_n of the output distribution ν are always finite, provided the input distribution μ is finite. Computing an expected value with respect to ν thus simply amounts to computing a sequence of finite sums $\mathbf{E}_{\nu_0}[Q], \mathbf{E}_{\nu_1}[Q], \dots$, which is guaranteed to converge monotonically to the analytical solution $\mathbf{E}_\nu[Q]$. The approximate semantics $\llbracket - \rrbracket_n$ can be thought of as an executable version of the denotational semantics $\llbracket - \rrbracket$. We implement it in the next section and use it to approximate network metrics based on the above result. The rest of this section gives more general approximation results for measures and kernels on 2^H , and shows that we can in fact handle continuous input distributions as well.

A measure is a *finite discrete measure* if it is of the form $\sum_{a \in F} r_a \delta_a$, where $F \in \wp_\omega(\wp_\omega(H))$ is a finite set of finite subsets of packet histories H , $r_a \geq 0$ for all $a \in F$, $\sum_{a \in F} r_a = 1$. Without loss of generality, we can write any such measure in the form $\sum_{a \subseteq b} r_a \delta_a$ for any $b \in \wp_\omega(H)$ such that $\bigcup F \subseteq b$ by taking $r_a = 0$ for $a \in 2^b - F$.

Saheb-Djahromi [151, Theorem 3] shows that every measure is a supremum of a directed set of finite discrete measures. This implies that the measures form a continuous DCPO with basis consisting of the finite discrete measures. In our model, the finite discrete measures have a particularly nice characterization:

For μ a measure and $b \in \wp_\omega(H)$, define the *restriction of μ to b* to be the finite discrete measure

$$\mu \upharpoonright b := \sum_{a \subseteq b} \mu(A_{ab}) \delta_a.$$

Theorem 3.8.3. *The set $\{\mu \upharpoonright b \mid b \in \wp_\omega(H)\}$ is a directed set with supremum μ . Moreover, the DCPO of measures is continuous with basis consisting of the finite discrete measures.*

We can lift the result to continuous kernels, which implies that *every program is approximated arbitrarily closely by programs whose outputs are finite discrete measures.*

Lemma 3.8.4. *Let $b \in \wp_\omega(H)$. Then $(P \cdot b)(a, -) = P(a, -) \upharpoonright b$.*

Now suppose the input distribution μ in Corollary 3.8.2 is continuous. By Theorem 3.8.3, μ is the supremum of an increasing chain of finite discrete measures $\mu_1 \sqsubseteq \mu_2 \sqsubseteq \dots$. If we redefine $\nu_n := \mu_n \gg \llbracket p \rrbracket_n$ then by Theorem 3.7.1 the ν_n still approximate the output distribution ν and Corollary 3.8.2 continues to hold. Even though the input distribution is now continuous, the output distribution can still be approximated by a chain of finite distributions and hence the expected value can still be approximated by a chain of finite sums.

3.9 Implementation and Case Studies

We built a simple interpreter for ProbNetKAT in OCaml that implements the denotational semantics as presented in Figure 3.2. Given a query, the interpreter approximates the answer through a monotonically increasing sequence of values (Theorems 3.8.1 and 3.8.2). Although preliminary in nature—more work on data structures and algorithms for manipulating distributions would be needed to obtain an efficient implementation—we were able to use our implementation to conduct several case studies involving probabilistic reasoning about properties of a real-world network: Internet2’s Abilene backbone.

Routing. In the networking literature, a large number of traffic engineering (TE) approaches have been explored. We built ProbNetKAT implementations of each of the following routing schemes:

- **Equal Cost Multipath Routing (ECMP):** The network uses all least-cost paths between each source-destination pair, and maps incoming traffic flows onto those paths randomly. ECMP can reduce congestion and increase throughput, but can also perform poorly when multiple paths traverse the same bottleneck link.

- **k -Shortest Paths (KSP):** The network uses the top k -shortest paths between each pair of hosts, and again maps incoming traffic flows onto those paths randomly. This approach inherits the benefits of ECMP and provides improved fault-tolerance properties since it always spreads traffic across k distinct paths.
- **Multipath Routing (Multi):** This is similar to KSP, except that it makes an independent choice from among the k -shortest paths at each hop rather than just once at ingress. This approach dynamically routes around bottlenecks and failures but can use extremely long paths—even ones containing loops.
- **Oblivious Routing (Räcke):** The network forwards traffic using a pre-computed probability distribution on carefully constructed overlays. The distribution is constructed in such a way that guarantees worst-case congestion within a polylogarithmic factor of the optimal scheme, regardless of the demands for traffic.

Note that all of these schemes rely on some form of randomization and hence are probabilistic in nature.

Traffic Model. Network operators often use traffic models constructed from historical data to predict future performance. We built a small OCaml tool that translates traffic models into ProbNetKAT programs using a simple encoding. Assume that we are given a traffic matrix (TM) that relates pairs of hosts (u, v) to the amount of traffic that will be sent from u to v . By normalizing each TM entry using the aggregate demand $\sum_{(u,v)} TM(u, v)$, we get a probability distribution d over pairs of hosts. For a pair of source and destination (u, v) , the associated probability $d(u, v)$ denotes the amount of traffic from u to v relative to the total traffic. Assuming uniform packet sizes, this is also the probability that a random packet generated in the network has source u and destination v . So, we can encode a TM as a program that generates packets according to

d :

$$inp := \bigoplus_{d(u,v)} \pi_{(u,v)}!$$

$$\text{where, } \pi_{(u,v)}! := \text{src} \leftarrow u \cdot \text{dst} \leftarrow v \cdot \text{sw} \leftarrow u$$

$\pi_{(u,v)}!$ generates a packet at u with source u and destination v . For any (non-empty) input, inp generates a distribution μ on packet histories which can be fed to the network program. For instance, consider a uniform traffic distribution for our 4-switch example (see Figure 3.1) where each node sends equal traffic to every other node. There are twelve (u, v) pairs with $u \neq v$. So, $d(u, v)_{u \neq v} = \frac{1}{12}$ and $d(u, u) = 0$. We also store the aggregate demand as it is needed to model queries such as expected link congestion, throughput etc.

Queries. Our implementation can be used to answer probabilistic queries about a variety of network performance properties. §3.2 showed an example of using a query to compute expected congestion. We can also measure expected mean latency in terms of path length:

```
let path_length (h: Hist.t) : Real.t =
  Real.of_int ((Hist.length h)/2 + 1)
let lift_query_avg (q: Hist.t -> Real.t) : (HSet.t -> Real.t) =
  fun hset ->
    let n = HSet.length hset in
    if n = 0 then Real.zero else
    let sum = HSet.fold hset ~init:Real.zero
      ~f:(fun acc h -> Real.(acc + q h)) in
    Real.(sum / of_int n)
```

The latency function (`path_length`) counts the number of switches in a history. We lift this function to sets and compute the expectation (`lift_query_avg`) by computing the average over all histories in the set (after discarding empty sets).

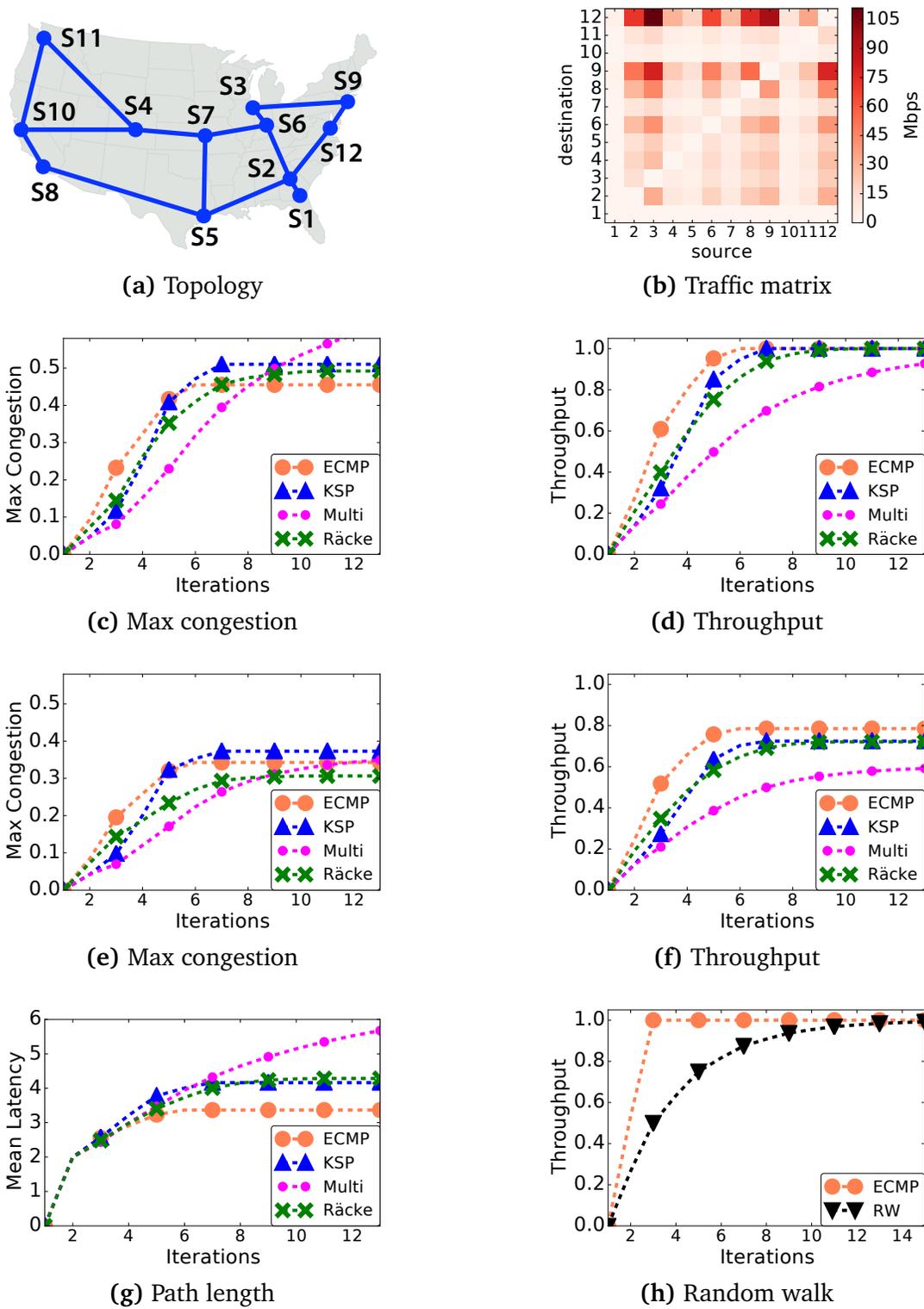


Figure 3.5: Case study with Abilene: (c, d) without loss; (e, f) with faulty links; (h) random walk in 4-cycle (all packets are eventually delivered).

Case Study: Abilene. To demonstrate the applicability of ProbNetKAT for reasoning about a real network, we performed a case study based on the topology and traffic demands from Internet2’s Abilene backbone network as shown in Figure 3.5 (a). We evaluate the traffic engineering approaches discussed above by modeling traffic matrices based on NetFlow traces gathered from the production network. A sample TM is depicted in Figure 3.5 (b).

Figures 3.5 (c,d,g) show the expected maximum congestion, throughput and mean latency. Because we model a network using the Kleene star operator, we see that the values converge monotonically as the number of iterations used to approximate Kleene star increases, as guaranteed by Corollary 3.8.2.

Failures. Network failures such as a faulty router or a link going down are common in large networks [51]. Hence, it is important to be able to understand the behavior and performance of a network in the presence of failures. We can model failures by assigning empirically measured probabilities to various components—e.g., we can modify our encoding of the topology so that every link in the network drops packets with probability $\frac{1}{10}$:

$$\begin{aligned} \ell_{1,2} := & \text{sw}=S_1 \cdot \text{pt}=2 \cdot \text{dup} \cdot ((\text{sw}\leftarrow S_2 \cdot \text{pt}\leftarrow 1 \cdot \text{dup}) \oplus_{0.9} \text{drop}) \\ & \& \text{sw}=S_2 \cdot \text{pt}=1 \cdot \text{dup} \cdot ((\text{sw}\leftarrow S_1 \cdot \text{pt}\leftarrow 2 \cdot \text{dup}) \oplus_{0.9} \text{drop}) \end{aligned}$$

Figures 3.5 (e-f) show the network performance for Abilene under this failure model. As expected, congestion and throughput decrease as more packets are dropped. As every link drops packets probabilistically, the relative fraction of packets delivered using longer links decreases—hence, there is a decrease in mean latency.

Loop detection. Forwarding loops in a network are extremely undesirable as they increase congestion and can even lead to black holes. With probabilistic routing, not all loops will necessarily result in a black hole—if there is a non-zero probability of exiting

a loop, every packet entering it will eventually exit. Consider the example of random walk routing in the four-node topology from Figure 3.1. In a random walk, a switch either forwards traffic directly to its destination or to a random neighbor. As packets are never duplicated and only exit the network when they reach their destination, the total throughput is equivalent to the fraction of packets that exit the network. Figure 3.5 (h) shows that the fraction of packets exiting increases monotonically with number of iterations and converges to 1. Moreover, histories can be queried to test if it encountered a topological loop by checking for duplicate locations. Hence, given a model that computes all possible history prefixes that appear in the network, we can query it for presence of loops. We do this by removing *out* from our standard network model and using $in \cdot (p \cdot t)^* \cdot p$ instead. This program generates the required distribution on history prefixes. Moreover, if we generalize packets with wildcard fields, similar to HSA [77], we can check for loops symbolically. We have extended our implementation in this way, and used it to check whether the network exhibits loops on a number of routing schemes based on probabilistic forwarding.

3.10 Related Work

This chapter builds on previous work on NetKAT [6, 45] and ProbNetKAT [44], but develops a semantics based on ordered domains as well as new applications to traffic engineering.

Domain Theory. The domain-theoretic treatment of probability measures goes back to the seminal work of Saheb-Djahromi [151], who was the first to identify and study the CPO of probability measures. Jones and Plotkin [71, 72] generalized and extended this work by giving a category-theoretical treatment and proving that the probabilistic powerdomain is a monad. It is an open problem if there exists a cartesian-closed

category of continuous DCPOs that is closed under the probabilistic powerdomain; see [74] for a discussion. This is an issue for higher-order probabilistic languages, but not for ProbNetKAT, which is strictly first-order. Edalat [36–38] gives a computational account of measure theory and integration for general metric spaces based on domain theory. More recent papers on probabilistic powerdomains are [55, 62, 74]. All this work is ultimately based on Scott’s pioneering work [155].

Probabilistic Logic and Semantics. Computational models and logics for probabilistic programming have been extensively studied. Denotational and operational semantics for probabilistic while programs were first studied by Kozen [87]. Early logical systems for reasoning about probabilistic programs were proposed in [88, 144, 150]. There are also numerous recent efforts [53, 56, 94, 102, 117]. Sankaranarayanan et al. [153] propose static analysis to bound the the value of probability queries. Probabilistic programming in the context of artificial intelligence has also been extensively studied in recent years [19, 148]. Probabilistic automata in several forms have been a popular model going back to the early work of Paz [132], as well as more recent efforts [112, 156, 157]. Denotational models combining probability and nondeterminism have been proposed by several authors [111, 170, 173], and general models for labeled Markov processes, primarily based on Markov kernels, have been studied extensively [34, 128, 129].

Our semantics is also related to the work on event structures [125, 172]. A (Prob)NetKAT program denotes a simple (probabilistic) event structure: packet histories are events with causal dependency given by extension and with all finite subsets consistent. We have to yet explore whether the event structure perspective on our semantics could lead to further applications and connections to e.g. (concurrent) games.

Networking. Network calculus is a general framework for analyzing network behavior using tools from queuing theory [28]. It has been used to reason about quantitative

properties such as latency, bandwidth, and congestion. The stochastic branch of network calculus provides tools for reasoning about the probabilistic behavior, especially in the presence of statistical multiplexing, but is often considered difficult to use. In contrast, ProbNetKAT is a self-contained framework based on a precise denotational semantics.

Traffic engineering has been extensively studied and a wide variety of approaches have been proposed for data-center networks [4, 69, 133, 161, 180] and wide-area networks [9, 42, 61, 64, 68, 75, 141, 166]. These approaches optimize for metrics such as congestion, throughput, latency, fault tolerance, fairness etc. Optimal techniques typically have high overheads [29], but oblivious [9, 83] and hybrid approaches with near-optimal performance [64, 68] have recently been adopted.

3.11 Conclusion

This chapter presents a new order-theoretic semantics for ProbNetKAT in the style of classical domain theory. The semantics allows a standard least-fixpoint treatment of iteration, and enables new modes of reasoning about the probabilistic network behavior. We have used these theoretical tools to analyze several randomized routing protocols on real-world data.

The main technical insight is that all programs and the operators defined on them are continuous, provided we consider the right notion of continuity: that induced by the Scott topology. Continuity enables precise approximation, and we exploited this to build an implementation. But continuity is also a powerful tool for reasoning that we expect to be very helpful in the future development of ProbNetKAT's meta theory. To establish continuity we had to switch from the Cantor to the Scott topology, and give up reasoning in terms of a metric. Luckily we were able to show a strong correspondence between the two topologies and that the Cantor-perspective and the Scott-perspective lead to equivalent definitions of the semantics. This allows us to choose whichever perspective

is best-suited for the task at hand.

Future Work. The results of this chapter are general enough to accommodate arbitrary extensions of ProbNetKAT with continuous Markov kernels or continuous operators on such kernels. An obvious next step is therefore to investigate extension of the language that would enable richer network models. Previous work on deterministic NetKAT included a decision procedure and a sound and complete axiomatization. In the presence of probabilities we expect a decision procedure will be hard to devise, as witnessed by several undecidability results on probabilistic automata. We intend to explore decision procedures for restricted fragments of the language. Another interesting direction is to compile ProbNetKAT programs into suitable automata that can then be analyzed by a probabilistic model checker such as PRISM [101]. A sound and complete axiomatization remains subject of further investigation, we can draw inspiration from recent work [94, 109]. Another opportunity is to investigate a weighted version of NetKAT, where instead of probabilities we consider weights from an arbitrary semiring, opening up several other applications—e.g. in cost analysis. Finally, we would like to explore efficient implementation techniques including compilation, as well as approaches based on sampling, following several other probabilistic languages [19, 130].

Chapter 4

Scalable Verification

“Take it to the limit, take it to the limit.

Take it to the limit one more time.”

—Eagles

This chapter presents McNetKAT, a scalable tool for verifying probabilistic network programs. McNetKAT is based on a new semantics for the guarded and history-free fragment of Probabilistic NetKAT in terms of finite-state, absorbing Markov chains. This view allows the semantics of all programs to be computed exactly, enabling construction of an automatic verification tool. Domain-specific optimizations and a parallelizing backend enable McNetKAT to analyze networks with thousands of nodes, automatically reasoning about general properties such as probabilistic program equivalence and refinement, as well as networking properties such as resilience to failures. We evaluate McNetKAT’s scalability using real-world topologies, compare its performance against state-of-the-art tools, and develop an extended case study on a recently proposed data center network design.

4.1 Introduction

Networks are among the most complex and critical computing systems used today. Researchers have long sought to develop automated techniques for modeling and analyzing network behavior [177], but only over the last decade has programming language methodology been brought to bear on the problem [20, 22, 113], opening up new avenues for reasoning about networks in a rigorous and principled way [6, 45, 77, 79, 105]. Building on these initial advances, researchers have begun to target more sophisticated networks that exhibit richer phenomena. In particular, there is renewed interest in *randomization* as a tool for designing protocols and modeling behaviors that arise in large-scale systems—from uncertainty about the inputs, to expected load, to likelihood of device and link failures.

Although programming languages for describing randomized networks exist [44, 49], support for automated reasoning remains limited. Even basic properties require quantitative reasoning in the probabilistic setting, and seemingly simple programs can generate complex distributions. Whereas state-of-the-art tools can easily handle deterministic networks with hundreds of thousands of nodes, probabilistic tools are currently orders of magnitude behind.

This chapter presents McNetKAT, a new tool for reasoning about probabilistic network programs written in the guarded and history-free fragment of Probabilistic NetKAT (ProbNetKAT) [6, 44, 45, 165]. ProbNetKAT is an expressive programming language based on Kleene Algebra with Tests, capable of modeling a variety of probabilistic behaviors and properties including randomized routing [99, 171], uncertainty about demands [147], and failures [51]. The history-free fragment restricts the language semantics to input-output behavior rather than tracking paths, and the guarded fragment provides conditionals and while loops rather than union and iteration operators. Al-

though the fragment we consider is a restriction of the full language, it is still expressive enough to encode a wide range of practical networking models. Existing deterministic tools, such as Anteater [108], HSA [77], and Veriflow [79], also use guarded and history-free models.

To enable automated reasoning, we first reformulate the semantics of ProbNetKAT in terms of finite state Markov chains. We introduce a *big-step* semantics that models programs as Markov chains that transition from input to output in a single step, using an auxiliary *small-step* semantics to compute the closed-form solution for the semantics of the iteration operator. We prove that the Markov chain semantics coincides with the domain-theoretic semantics for ProbNetKAT developed in Chapter 3. Our new semantics also has a key benefit: the limiting distribution of the resulting Markov chains can be computed exactly in closed form, yielding a concise representation that can be used as the basis for building a practical tool.

We have implemented McNetKAT in an OCaml prototype that takes a ProbNetKAT program as input and produces a stochastic matrix that models its semantics in a finite and explicit form. McNetKAT uses the UMFPACK linear algebra library as a back-end solver to efficiently compute limiting distributions [30], and exploits algebraic properties to automatically parallelize the computation across multiple machines. To facilitate comparisons with other tools, we also developed a back-end based on PRISM [100].

To evaluate the scalability of McNetKAT, we conducted experiments on realistic topologies, routing schemes, and properties. Our results show that McNetKAT scales to networks with thousands of switches, and performs orders of magnitude better than a state-of-the-art tool based on general-purpose symbolic inference [49, 50]. We also used McNetKAT to carry out a case study of the resilience of a fault-tolerant data center design proposed by Liu et al. [106].

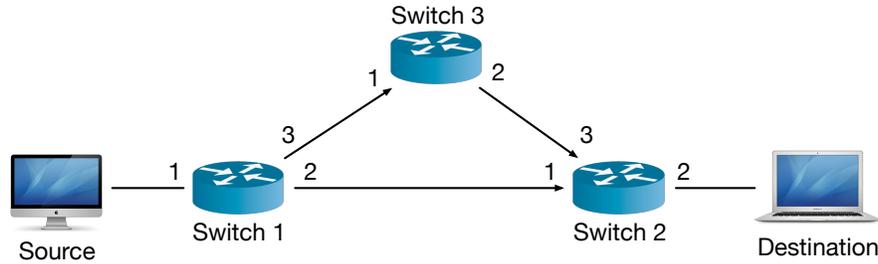


Figure 4.1: Network topology for running example.

Contributions and outline. The central contribution of this chapter is the development of a *scalable probabilistic network verification tool*. We develop a new, tractable semantics that is sound with respect to ProbNetKAT’s original denotational model. We present a prototype implementation and evaluate it on a variety of scenarios drawn from real-world networks. In Section 4.2, we introduce ProbNetKAT using a running example. In Section 4.3, we present a semantics based on *finite stochastic matrices* and show that it fully characterizes the behavior of ProbNetKAT programs (Theorem 4.3.1). In Section 4.4, we show how to compute the matrix associated with iteration in closed form. In Section 4.5, we discuss our implementation, including symbolic data structures and optimizations that are needed to handle the large state space efficiently. In Section 4.6, we evaluate the scalability of McNetKAT on a common data center design and compare its performance against state-of-the-art probabilistic tools. In Section 4.7, we present a case study using McNetKAT to analyze resilience in the presence of link failures. We survey related work in Section 4.8 and conclude in Section 4.9. We defer proofs to appendix Appendix B.

4.2 Overview

This section introduces a running example that illustrates the main features of the ProbNetKAT language as well as some quantitative network properties that arise in practice.

Background on ProbNetKAT. Consider the network in Figure 4.1, which connects a source to a destination in a topology with three switches. We will first introduce a program that forwards packets from the source to the destination, and then verify that it correctly implements the desired behavior. Next, we will show how to enrich our program to model the possibility of link failures, and develop a fault-tolerant forwarding scheme that automatically routes around failures. Using a quantitative version of program refinement, we will show that the fault-tolerant program is indeed more resilient than the initial program. Finally, we will show how to compute the expected degree of resilience analytically.

To a first approximation, a ProbNetKAT program can be thought of as a randomized function that maps input packets to sets of output packets. Packets are modeled as records, with fields for standard headers—such as the source (`src`) and destination (`dst`) addresses—as well as two fields `sw` and `pt` encoding the current location of the packet. ProbNetKAT provides several primitives for manipulating packets: a *modification* $f \leftarrow n$ returns the input packet with field f updated to n , while a *test* $f = n$ returns either the input packet unmodified if the test succeeds, or the empty set if the test fails. The primitives `skip` and `drop` behave like a test that always succeeds and fails, respectively. In the guarded fragment of the language, programs can be composed sequentially ($p \cdot q$), using conditionals (**if** p **then** q_1 **else** q_2), while loops (**while** p **do** q), or probabilistic choice ($p \oplus q$).

Although ProbNetKAT programs can be freely constructed by composing primitive operations, a typical network model is expressed using two programs: a *forwarding* program (sometimes called a *policy*) and a *link* program (sometimes called a *topology*). The forwarding program describes how packets are transformed locally by the switches at each hop. In our running example, to route packets from the source to the destination, switches 1 and 2 can simply forward all incoming packets out on port 2 by modifying

the port field (pt). This program can be encoded in ProbNetKAT by performing a case analysis on the location of the input packet, and then setting the port field to 2:

$$p := \mathbf{if\ sw=1\ then\ pt\leftarrow 2\ else}$$

$$\mathbf{if\ sw=2\ then\ pt\leftarrow 2\ else\ drop}$$

The final drop at the end of this program encodes the policy for switch 3, which is unreachable.

We can model the topology as a cascade of conditionals that match packets at the end of each link and update their locations to the link’s destination:

$$t := \mathbf{if\ sw=1 \cdot pt=2\ then\ sw\leftarrow 2 \cdot pt\leftarrow 1\ else\ \dots}$$

To build the overall network model, we first define predicates for the ingress and egress locations,

$$in := sw=1 \cdot pt=1 \qquad out := sw=2 \cdot pt=2$$

and then combine the forwarding policy p with the topology t . More specifically, a packet traversing the network starts at an ingress and is repeatedly processed by switches and links until it reaches an egress:

$$M(p, t) := in \cdot p \cdot \mathbf{while\ \neg out\ do\ (t \cdot p)}$$

We can now state and prove properties about the network by reasoning about this model. For instance, the following equivalence states that p forwards all packets to the destination:

$$M(p, t) \equiv in \cdot sw\leftarrow 2 \cdot pt\leftarrow 2$$

The program on the right can be regarded as an ideal specification that “teleports” each packet to its destination. Such equations were also used in previous work to reason about properties such as waypointing, reachability, isolation, and loop freedom [6, 45].

Probabilistic reasoning. Real-world networks often exhibit nondeterministic behaviors such as fault tolerant routing schemes to handle unexpected failures [106] and randomized algorithms to balance load across multiple paths [99]. Verifying that networks behave as expected in these more complicated scenarios requires a form of probabilistic reasoning, but most state-of-the-art network verification tools model only deterministic behaviors [45, 77, 79].

To illustrate, suppose we want to extend our example with link failures. Most modern switches execute low-level protocols such as Bidirectional Forwarding Detection (BFD) that compute real-time health information about the link connected to each physical port [15]. We can enrich our model so that each switch has a boolean flag up_i that indicates whether the link connected to the switch at port i is up. Then, we can adjust the forwarding logic to use backup paths when the link is down: for switch 1,

$$\begin{aligned} \widehat{p}_1 := & \text{if } up_2=1 \text{ then } pt \leftarrow 2 \text{ else} \\ & \text{if } up_2=0 \text{ then } pt \leftarrow 3 \text{ else drop} \end{aligned}$$

and similarly for switches 2 and 3. As before, we can package the forwarding logic for all switches into a single program:

$$\widehat{p} := \text{if } sw=1 \text{ then } \widehat{p}_1 \text{ else if } sw=2 \text{ then } \widehat{p}_2 \text{ else } \widehat{p}_3$$

Next, we update the encoding of our topology to faithfully model link failures. Links can fail for a wide variety of reasons, including human errors, fiber cuts, and hardware faults. A natural way to model such failures is with a *probabilistic model*—i.e., with a distribution that captures how often certain links fail:

$$\begin{aligned} f_0 &:= up_{2 \leftarrow 1} \cdot up_{3 \leftarrow 1} \\ f_1 &:= \oplus \left\{ f_0 @ \frac{1}{2}, (up_{2 \leftarrow 0} \cdot up_{3 \leftarrow 1}) @ \frac{1}{4}, (up_{2 \leftarrow 1} \cdot up_{3 \leftarrow 0}) @ \frac{1}{4} \right\} \\ f_2 &:= (up_{2 \leftarrow 1} \oplus_{.8} up_{2 \leftarrow 0}) \cdot (up_{3 \leftarrow 1} \oplus_{.8} up_{3 \leftarrow 0}) \end{aligned}$$

Intuitively, in f_0 no links fail, in f_1 the links ℓ_{12} and ℓ_{13} fail with probability 25% but at most one link fails, while in f_2 the links fail independently with probability 20%. Using the up flags, we can model a topology with possibly faulty links like so:

$$\hat{t} := \mathbf{if} \text{ sw}=1 \cdot \text{pt}=2 \cdot \text{up}_2=1 \mathbf{then} \text{ sw}\leftarrow 2 \cdot \text{pt}\leftarrow 1 \mathbf{else} \dots$$

Combining the policy, topology, and failure model yields a model of the entire network:

$$\begin{aligned} \widehat{M}(p, t, f) := & \mathbf{var} \text{ up}_2 \leftarrow 1 \mathbf{in} \\ & \mathbf{var} \text{ up}_3 \leftarrow 1 \mathbf{in} \\ & M((f \cdot p), t) \end{aligned}$$

This refined model \widehat{M} wraps our previous model M with declarations of the two local fields up_2 and up_3 and executes the failure model (f) at each hop before executing the programs for the switch (p) and topology (t) .

Now we can analyze our resilient routing scheme \hat{p} . As a sanity check, we can verify that it delivers packets to their destinations in the absence of failures. Formally, it behaves like the program that teleports packets to their destinations:

$$\widehat{M}(\hat{p}, \hat{t}, f_0) \equiv \text{in} \cdot \text{sw}\leftarrow 2 \cdot \text{pt}\leftarrow 2$$

More interestingly, \hat{p} is 1-resilient—i.e., it delivers packets provided at most one link fails. Note that this property does *not* hold for the original, naive routing scheme p :

$$\widehat{M}(\hat{p}, \hat{t}, f_1) \equiv \text{in} \cdot \text{sw}\leftarrow 2 \cdot \text{pt}\leftarrow 2 \not\equiv \widehat{M}(p, \hat{t}, f_1)$$

While \hat{p} is not fully resilient under failure model f_2 , which allows two links to fail simultaneously, we can still show that the refined routing scheme \hat{p} performs strictly better than the naive scheme p by checking

$$\widehat{M}(p, \hat{t}, f_2) < \widehat{M}(\hat{p}, \hat{t}, f_2)$$

where $p < q$ intuitively means that q delivers packets with higher probability than p .

Going a step further, we might want to compute more general quantitative properties of the distributions generated for a given program. For example, we might compute the probability that each routing scheme delivers packets to the destination under f_2 (i.e., 80% for the naive scheme and 96% for the resilient scheme), potentially valuable information to help an Internet Service Provider (ISP) evaluate a network design to check that it meets certain service-level agreements (SLAs). With this motivation in mind, we aim to build a scalable tool that can carry out automated reasoning on probabilistic network programs expressed in ProbNetKAT.

4.3 ProbNetKAT Syntax and Semantics

This section reviews the syntax of ProbNetKAT and presents a new semantics based on finite state Markov chains.

Preliminaries. A *packet* π is a record mapping a finite set of fields f_1, f_2, \dots, f_k to bounded integers n . As we saw in the previous section, fields can include standard header fields such as source (`src`) and destination (`dst`) addresses, as well as logical fields for modeling the current location of the packet in the network or variables such as up_i . These logical fields are not present in a physical network packet, but they can track auxiliary information for the purposes of verification. We write $\pi.f$ to denote the value of field f of π and $\pi[f:=n]$ for the packet obtained from π by updating field f to hold n . We let Pk denote the (finite) set of all packets.

Syntax. ProbNetKAT terms can be divided into two classes: *predicates* (t, u, \dots) and *programs* (p, q, \dots). Primitive predicates include *tests* ($f=n$) and the Boolean constants *false* (`drop`) and *true* (`skip`). Compound predicates are formed using the usual Boolean

connectives: disjunction ($t \& u$), conjunction ($t \cdot u$), and negation ($\neg t$). Primitive programs include *predicates* (t) and *assignments* ($f \leftarrow n$). The original version of the language also provides a *dup* primitive, which logs the current state of the packet, but the history-free fragment omits this operation. Compound programs can be formed using *parallel composition* ($p \& q$), *sequential composition* ($p \cdot q$), and *iteration* (p^*). In addition, the *probabilistic choice* operator $p \oplus_r q$ executes p with probability r and q with probability $1 - r$, where r is rational, $0 \leq r \leq 1$. We sometimes use an n -ary version and omit the r 's: $p_1 \oplus \dots \oplus p_n$ executes a p_i chosen uniformly at random. In addition to these core constructs (summarized in Figure 4.2), many other useful constructs can be derived. For example, mutable local variables (e.g., up_i , used to track link health in Section 4.2), can be desugared into the language:

$$\mathbf{var} \ f \leftarrow n \ \mathbf{in} \ p := f \leftarrow n \cdot p \cdot f \leftarrow 0$$

Here f is a field that is local to p . The final assignment $f \leftarrow 0$ sets the value of f to a canonical value, “erasing” it after the field goes out of scope. We often use local variables to record extra information for verification—e.g., recording whether a packet traversed a given switch allows reasoning about simple waypointing and isolation properties, even though the history-free fragment of ProbNetKAT does not model paths directly.

Guarded fragment. Conditionals and while loops can be encoded using union and iteration:

$$\mathbf{if} \ t \ \mathbf{then} \ p \ \mathbf{else} \ q := t \cdot p \& \neg t \cdot q$$

$$\mathbf{while} \ t \ \mathbf{do} \ p := (t \cdot p)^* \cdot \neg t$$

Note that these constructs use the predicate t as a *guard*, resolving the inherent non-determinism in the union and iteration operators. Our implementation handles programs in the guarded fragment of the language—i.e., with loops and conditionals but without union and iteration—though we will develop the theory in full generality here, to

Syntax		Semantics
Naturals	$n ::= 0 \mid 1 \mid 2 \mid \dots$	$\mathcal{B}[[p]] \in \mathbb{S}(2^{\text{Pk}})$
Fields	$f ::= f_1 \mid \dots \mid f_k$	$\mathcal{B}[[\text{drop}]]_{ab} := [b = \emptyset]$
Packets	$\text{Pk} \ni \pi ::= \{f_1 = n_1, \dots, f_k = n_k\}$	$\mathcal{B}[[\text{skip}]]_{ab} := [a = b]$
Probabilities	$r \in [0, 1] \cap \mathbb{Q}$	$\mathcal{B}[[f=n]]_{ab} := [b = \{\pi \in a \mid \pi.f = n\}]$
Predicates	$t, u ::= \text{drop}$	$\mathcal{B}[[\neg t]]_{ab} := [b \subseteq a] \cdot \mathcal{B}[[t]]_{a, a-b}$
	skip	$\mathcal{B}[[f \leftarrow n]]_{ab} := [b = \{\pi[f := n] \mid \pi \in a\}]$
	$f=n$	$\mathcal{B}[[p \& q]]_{ab} :=$
	$t \& u$	$\sum_{c,d} [c \cup d = b] \cdot \mathcal{B}[[p]]_{a,c} \cdot \mathcal{B}[[q]]_{a,d}$
	$t \cdot u$	$\mathcal{B}[[p \cdot q]] := \mathcal{B}[[p]] \cdot \mathcal{B}[[q]]$
Programs	$\neg t$	$\mathcal{B}[[p \oplus_r q]] := r \cdot \mathcal{B}[[p]] + (1 - r) \cdot \mathcal{B}[[q]]$
	True	$\mathcal{B}[[p^*]]_{ab} := \lim_{n \rightarrow \infty} \mathcal{B}[[p^{(n)}]]_{ab}$
	Test	
	Disjunction	
	Conjunction	
	$\neg t$	Negation
	t	Filter
	$f \leftarrow n$	Assignment
	$p \& q$	Union
	$p \cdot q$	Sequence
	$p \oplus_r q$	Choice
	p^*	Iteration

Figure 4.2: History-free ProbNetKAT: syntax and semantics. The matrix entry $\mathcal{B}[[p]]_{ab}$ gives the probability that program p produces output b on input a .

make connections to previous work on ProbNetKAT clearer. We believe this restriction is acceptable from a practical perspective, as the main purpose of union and iteration is to encode forwarding tables and network-wide processing, and the guarded variants can often perform the same task. A notable exception is multicast, which cannot be expressed in the guarded fragment.

Semantics. Previous work on ProbNetKAT [44] modeled history-free programs as maps $2^{\text{Pk}} \rightarrow \mathcal{D}(2^{\text{Pk}})$, where $\mathcal{D}(2^{\text{Pk}})$ denotes the set of probability distributions on 2^{Pk} . This semantics is useful for establishing fundamental properties of the language, but we will need a more explicit representation to build a practical verification tool. Since the set of packets is finite, probability distributions over sets of packets are discrete and can be characterized by a *probability mass function*, $f : 2^{\text{Pk}} \rightarrow [0, 1]$ such that $\sum_{b \subseteq \text{Pk}} f(b) = 1$. It will be convenient to view f as a *stochastic vector* of non-negative entries that sum to 1.

A program, which maps inputs a to distributions over outputs, can then be represented by a square matrix indexed by $\mathcal{P}k$ in which the stochastic vector corresponding to input a appears as the a -th row. Thus, we can interpret a program p as a matrix $\mathcal{B}[[p]] \in [0, 1]^{2^{\mathcal{P}k} \times 2^{\mathcal{P}k}}$ indexed by packet sets, where the matrix entry $\mathcal{B}[[p]]_{ab}$ gives the probability that p produces output $b \in 2^{\mathcal{P}k}$ on input $a \in 2^{\mathcal{P}k}$. The rows of the matrix $\mathcal{B}[[p]]$ are stochastic vectors, each encoding the distribution produced for an input set a ; such a matrix is called *right-stochastic*, or simply stochastic. We write $\mathbb{S}(2^{\mathcal{P}k})$ for the set of right-stochastic matrices indexed by $2^{\mathcal{P}k}$.

Figure 4.2 defines an interpretation of ProbNetKAT programs as stochastic matrices; the Iverson bracket $[\varphi]$ is 1 if φ is true, and 0 otherwise. Deterministic program primitives are interpreted as $\{0, 1\}$ -matrices—e.g., the program primitive `drop` is interpreted as the following stochastic matrix:

$$\mathcal{B}[[\text{drop}]] = \begin{matrix} & \emptyset & b_2 & \dots & b_n \\ \emptyset & \begin{bmatrix} 1 & 0 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ a_n & 1 & 0 & \dots & 0 \end{bmatrix} & & & \end{matrix} \quad \begin{matrix} a_2 & \xrightarrow{1} & a_1 = \emptyset \\ \vdots & & \uparrow \\ a_n & \xrightarrow{1} & a_1 = \emptyset \end{matrix} \quad (4.1)$$

which assigns all probability mass to the \emptyset -column. Similarly, `skip` is interpreted as the identity matrix. Sequential composition can be interpreted as matrix product,

$$\mathcal{B}[[p \cdot q]]_{ab} = \sum_c \mathcal{B}[[p]]_{ac} \cdot \mathcal{B}[[q]]_{cb} = (\mathcal{B}[[p]] \cdot \mathcal{B}[[q]])_{ab}$$

which reflects the intuitive semantics of composition: to step from a to b in $\mathcal{B}[[p \cdot q]]$, one must step from a to an intermediate state c in $\mathcal{B}[[p]]$, and then from c to b in $\mathcal{B}[[q]]$.

As the picture in (4.1) suggests, a stochastic matrix $B \in \mathbb{S}(2^{\mathcal{P}k})$ can be viewed as a *Markov chain* (MC)—i.e., a probabilistic transition system with state space $2^{\mathcal{P}k}$. The entry B_{ab} gives the probability that the system transitions from a to b .

Soundness. The matrix $\mathcal{B}[[p]]$ is equivalent to the denotational semantics $[[p]]$ defined in Chapter 3 (see Appendix B for a detailed discussion and proof).

Theorem 4.3.1 (Soundness). *Let $a, b \in 2^{\text{Pk}}$. The matrix $\mathcal{B}[[p]]$ satisfies $\mathcal{B}[[p]]_{ab} = [[p]](a)(\{b\})$.*

Hence, checking program equivalence for p and q reduces to checking equality of the matrices $\mathcal{B}[[p]]$ and $\mathcal{B}[[q]]$.

Corollary 4.3.2. $[[p]] = [[q]]$ if and only if $\mathcal{B}[[p]] = \mathcal{B}[[q]]$.

In particular, because the Markov chains are all finite state, the transition matrices are finite dimensional with rational entries. Accordingly, program equivalence and other quantitative properties can be automatically verified provided we can compute the matrices for given programs. This is relatively straightforward for program constructs besides $\mathcal{B}[[p^*]]$, whose matrix is defined in terms of a limit. The next section presents a closed-form definition of the stochastic matrix for this operator.

4.4 Computing Stochastic Matrices

The semantics developed in the previous section can be viewed as a “big-step” semantics in which a single step models the execution of a program from input to output. To compute the semantics of p^* , we will introduce a finer, “small-step” chain in which a transition models one iteration of the loop.

To build intuition, consider simulating p^* using a transition system with states given by triples $\langle p, a, b \rangle$ in which p is the program being executed, a is the set of (input) packets, and b is an accumulator that collects the output packets generated so far. To model the execution of p^* on input a , we start from the initial state $\langle p^*, a, \emptyset \rangle$ and unroll p^* one iteration according to the characteristic equation $p^* \equiv \text{skip} \ \& \ p \cdot p^*$, yielding the following transition:

$$\langle p^*, a, \emptyset \rangle \xrightarrow{1} \langle \text{skip} \ \& \ p \cdot p^*, a, \emptyset \rangle$$

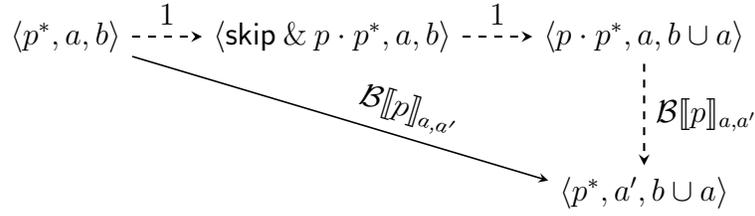


Figure 4.3: The small-step semantics is given by a Markov chain with states of the form $\langle \text{program}, \text{input set}, \text{output accumulator} \rangle$. The three dashed arrows can be collapsed into the single solid arrow, rendering the program component superfluous.

Next, we execute both skip and $p \cdot p^*$ on the input set and take the union of their results.

Executing skip yields the input set as output, with probability 1:

$$\langle \text{skip} \ \& \ p \cdot p^*, a, \emptyset \rangle \xrightarrow{1} \langle p \cdot p^*, a, a \rangle$$

Executing $p \cdot p^*$, executes p and feeds its output into p^* :

$$\forall a' : \quad \langle p \cdot p^*, a, a \rangle \xrightarrow{\mathcal{B}[[p]]_{a,a'}} \langle p^*, a', a \rangle$$

At this point we are back to executing p^* , albeit with a different input set a' and some accumulated output packets. The resulting Markov chain is shown in Figure 4.3.

Note that as the first two steps of the chain are deterministic, we can simplify the transition system by collapsing all three steps into one, as illustrated in Figure 4.3. The program component can then be dropped, as it now remains constant across transitions. Hence, we work with a Markov chain over the state space $2^{\text{Pk}} \times 2^{\text{Pk}}$, defined formally as follows:

$$\begin{aligned} \mathcal{S}[[p]] &\in \mathbb{S}(2^{\text{Pk}} \times 2^{\text{Pk}}) \\ \mathcal{S}[[p]]_{(a,b),(a',b')} &:= [b' = b \cup a] \cdot \mathcal{B}[[p]]_{a,a'}. \end{aligned}$$

We can verify that the matrix $\mathcal{S}[[p]]$ defines a Markov chain.

Lemma 4.4.1. $\mathcal{S}[[p]]$ is stochastic.

Next, we show that each step in $\mathcal{S}[[p]]$ models an iteration of p^* . Formally, the $(n + 1)$ -step of $\mathcal{S}[[p]]$ is equivalent to the big-step behavior of the n -th unrolling of p^* .

Proposition 4.4.2. $\mathcal{B}[[p^{(n)}]]_{a,b} = \sum_{a'} \mathcal{S}[[p]]_{(a,\emptyset),(a',b)}^{n+1}$

Direct induction on the number of steps $n \geq 0$ fails because the hypothesis is too weak. We generalize from start states with empty accumulator to arbitrary start states.

Lemma 4.4.3. *Let p be program. Then for all $n \in \mathbb{N}$ and $a, b, b' \subseteq \text{Pk}$, we have*

$$\sum_{a'} [b' = a' \cup b] \cdot \mathcal{B}[[p^{(n)}]]_{a,a'} = \sum_{a'} \mathcal{S}[[p]]_{(a,b),(a',b')}^{n+1}.$$

Proposition 4.4.2 then follows from Lemma 4.4.3 with $b = \emptyset$.

Intuitively, the long-run behavior of $\mathcal{S}[[p]]$ approaches the big-step behavior of p^* : letting (a_n, b_n) denote the random state of the Markov chain $\mathcal{S}[[p]]$ after taking n steps starting from (a, \emptyset) , the distribution of b_n for $n \rightarrow \infty$ is precisely the distribution of outputs generated by p^* on input a (by Proposition 4.4.2 and the definition of $\mathcal{B}[[p^*]]$).

Closed form. The limiting behavior of finite state Markov chains has been well studied in the literature (e.g., see Kemeny and Snell [78]). For so-called *absorbing* Markov chains, the limit distribution can be computed exactly. A state s of a Markov chain T is *absorbing* if it transitions to itself with probability 1,

$$\begin{array}{c} \textcircled{s} \\ \curvearrowright 1 \end{array} \quad (\text{formally: } T_{s,s'} = [s = s'])$$

and a Markov chain $T \in \mathbb{S}(S)$ is *absorbing* if each state can reach an absorbing state:

$$\forall s \in S. \exists s' \in S, n \geq 0. T_{s,s'}^n > 0 \text{ and } T_{s',s'} = 1$$

The non-absorbing states of an absorbing MC are called *transient*. Assume T is absorbing with n_t transient states and n_a absorbing states. After reordering the states so that absorbing states appear first, T has the form

$$T = \begin{bmatrix} I & 0 \\ R & Q \end{bmatrix}$$

where I is the $n_a \times n_a$ identity matrix, R is an $n_t \times n_a$ matrix giving the probabilities of transient states transitioning to absorbing states, and Q is an $n_t \times n_t$ matrix specifying the probabilities of transitions between transient states. Since absorbing states never transition to transient states by definition, the upper right corner contains a $n_a \times n_t$ zero matrix.

From any start state, a finite state absorbing MC always ends up in an absorbing state eventually, *i.e.* the limit $T^\infty := \lim_{n \rightarrow \infty} T^n$ exists and has the form

$$T^\infty = \begin{bmatrix} I & 0 \\ A & 0 \end{bmatrix}$$

where the $n_t \times n_a$ matrix A contains the so-called *absorption probabilities*. This matrix satisfies the following equation:

$$A = (I + Q + Q^2 + \dots) R$$

Intuitively, to transition from a transient state to an absorbing state, the MC can take an arbitrary number of steps between transient states before taking a single—and final—step into an absorbing state. The infinite sum $X := \sum_{n \geq 0} Q^n$ satisfies $X = I + QX$, and solving for X yields

$$X = (I - Q)^{-1} \quad \text{and} \quad A = (I - Q)^{-1} R. \quad (4.2)$$

(We refer the reader to Kemeny and Snell [78] for the proof that the inverse exists.)

Before we apply this theory to the small-step semantics $\mathcal{S}[-]$, it will be useful to introduce some MC-specific notation. Let T be an MC. We write $s \xrightarrow{T}_n s'$ if s can reach s' in precisely n steps, *i.e.* if $T_{s,s'}^n > 0$; and we write $s \xrightarrow{T} s'$ if s can reach s' in some number of steps, *i.e.* if $T_{s,s'}^n > 0$ for some $n \geq 0$. Two states are said to *communicate*, denoted $s \xleftrightarrow{T} s'$, if $s \xrightarrow{T} s'$ and $s' \xrightarrow{T} s$. The relation \xleftrightarrow{T} is an equivalence relation, and its equivalence classes are called *communication classes*. A communication class is *absorbing* if it cannot reach any states outside the class. Let $\Pr[s \xrightarrow{T}_n s']$ denote the probability $T_{s,s'}^n$.

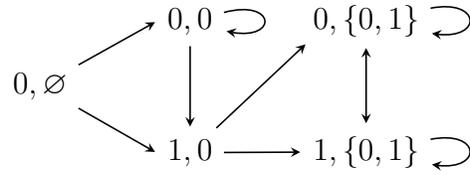
For the rest of the section, we fix a program p and abbreviate $\mathcal{B}[[p]]$ as B and $\mathcal{S}[[p]]$ as S . We also define *saturated states*, those where the accumulator has stabilized.

Definition 4.4.4. A state (a, b) of S is called *saturated* if b has reached its final value, *i.e.* if $(a, b) \xrightarrow{S} (a', b')$ implies $b' = b$. \square

After reaching a saturated state, the output of p^* is fully determined. The probability of ending up in a saturated state with accumulator b , starting from an initial state (a, \emptyset) , is

$$\lim_{n \rightarrow \infty} \sum_{a'} S_{(a, \emptyset), (a', b)}^n$$

and, indeed, this is the probability that p^* outputs b on input a by Proposition 4.4.2. Unfortunately, we cannot directly compute this limit since saturated states are not necessarily absorbing. To see this, consider $p^* = (f \leftarrow 0 \oplus_{1/2} f \leftarrow 1)^*$ over a single $\{0, 1\}$ -valued field f . Then S has the form

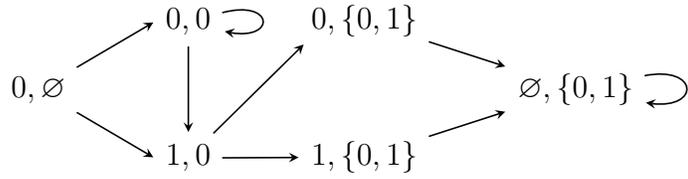


where all edges are implicitly labeled with $\frac{1}{2}$, and 0 and 1 denote the packets with f set to 0 and 1 respectively. We omit states not reachable from $(0, \emptyset)$. The right-most states are saturated, but they communicate and are thus not absorbing.

To align saturated and absorbing states, we can perform a quotient of this Markov chain by collapsing the communicating states. We define an auxiliary matrix,

$$U_{(a,b), (a',b')} := [b' = b] \cdot \begin{cases} [a' = \emptyset] & \text{if } (a, b) \text{ is saturated} \\ [a' = a] & \text{else} \end{cases}$$

which sends a saturated state (a, b) to a canonical saturated state (\emptyset, b) and acts as the identity on all other states. In our example, the modified chain SU is as follows:



and indeed is absorbing, as desired.

Lemma 4.4.5. S , U , and SU are monotone in the sense that: $(a, b) \xrightarrow{S} (a', b')$ implies $b \subseteq b'$ (and similarly for U and SU).

Proof. By definition (S and U) and by composition (SU). □

Next, we show that SU is an absorbing MC:

Proposition 4.4.6. Let $n \geq 1$.

1. $(SU)^n = S^n U$
2. SU is an absorbing MC with absorbing states $\{(\emptyset, b)\}$.

Arranging the states (a, b) in lexicographically ascending order according to \subseteq and letting $n = |2^{\text{Pk}}|$, it then follows from Proposition 4.4.6.2 that SU has the form

$$SU = \begin{bmatrix} I_n & 0 \\ R & Q \end{bmatrix}$$

where, for $a \neq \emptyset$, we have

$$(SU)_{(a,b),(a',b')} = \begin{bmatrix} R & Q \end{bmatrix}_{(a,b),(a',b')}.$$

Moreover, SU converges and its limit is given by

$$(SU)^\infty := \begin{bmatrix} I_n & 0 \\ (I - Q)^{-1}R & 0 \end{bmatrix} = \lim_{n \rightarrow \infty} (SU)^n. \quad (4.3)$$

Putting together the pieces, we can use the modified Markov chain SU to compute the limit of S .

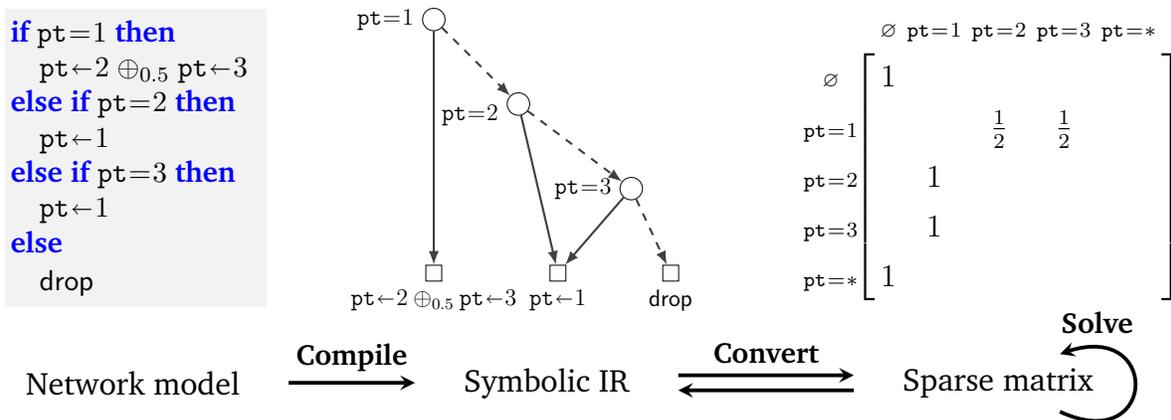


Figure 4.4: Implementation using FDDs and a sparse linear algebra solver.

Theorem 4.4.7 (Closed Form). *Let $a, b, b' \subseteq Pk$. Then*

$$\lim_{n \rightarrow \infty} \sum_{a'} S_{(a,b),(a',b')}^n = (SU)_{(a,b),(\emptyset,b')}^\infty.$$

The limit exists and can be computed exactly, in closed-form.

4.5 Implementation

We have implemented McNetKAT as an embedded DSL in OCaml in roughly 10KLoC. The frontend provides functions for defining and manipulating ProbNetKAT programs and for generating such programs automatically from network topologies encoded using Graphviz. These programs can then be analyzed by one of two backends: the *native backend* (PNK), which compiles programs to (symbolically represented) stochastic matrices; or the *PRISM-based backend* (PPNK), which emits inputs for the state-of-the-art probabilistic model checker PRISM [101].

Pragmatic restrictions. Although our semantics developed in Section 4.3 and Section 4.4 theoretically supports computations on sets of packets, a direct implementation would be prohibitively expensive—the matrices are indexed by the powerset 2^{Pk} of the universe of all possible packets! To obtain a practical analysis tool, we restrict the state

space to single packets. At the level of syntax, we restrict to the guarded fragment of ProbNetKAT, *i.e.* to programs with conditionals and while loops, but without union and iteration. This ensures that no proper packet sets are ever generated, thus allowing us to work over an exponentially smaller state space. While this restriction does rule out some uses of ProbNetKAT—most notably, modeling multicast—we did not find this to be a serious limitation because multicast is relatively uncommon in probabilistic networking. If needed, multicast can often be modeled using multiple unicast programs.

4.5.1 Native Backend

The native backend compiles a program to a symbolic representation of its big step matrix. The translation, illustrated in Figure 4.4, proceeds as follows. First, we translate atomic programs to Forwarding Decision Diagrams (FDDs), a symbolic data structure based on Binary Decision Diagrams (BDDs) that encodes sparse matrices compactly [164]. Second, we translate composite programs by first translating each sub-program to an FDD and then merging the results using standard BDD algorithms. Loops require special treatment: we (i) convert the FDD for the body of the loop to a sparse stochastic matrix, (ii) compute the semantics of the loop by using an optimized sparse linear solver [30] to solve the system from Section 4.4, and finally (iii) convert the resulting matrix back to an FDD. We use exact rational arithmetic in the frontend and FDD-backend to preempt concerns about numerical precision, but trust the linear algebra solver UMFPACK (based on 64 bit floats) to provide accurate solutions.¹ Our implementation relies on several optimizations; we detail two of the more interesting ones below.

Probabilistic FDDs. Binary Decision Diagrams [2] and variants thereof [46] have long been used in verification and model checking to represent large state spaces compactly.

¹UMFPACK is a mature library powering widely-used scientific computing packages such as MATLAB and SciPy.

A variant called Forwarding Decision Diagrams (FDDs) [164] was previously developed specifically for the networking domain, but only supported deterministic behavior. In this work, we extended FDDs to probabilistic FDDs. A probabilistic FDD is a rooted directed acyclic graph that can be understood as a control-flow graph. Interior nodes test packet fields and have outgoing true- and false- branches, which we visualize by solid lines and dashed lines in Figure 4.4. Leaf nodes contain distributions over *actions*, where an action is either a set of modifications or a special action drop. To interpret an FDD, we start at the root node with an initial packet and traverse the graph as dictated by the tests until a leaf node is reached. Then, we apply each action in the leaf node to the packet. Thus, an FDD represents a function of type $P_k \rightarrow \mathcal{D}(P_k + \emptyset)$, or equivalently, a stochastic matrix over the state space $P_k + \emptyset$ where the \emptyset -row puts all mass on \emptyset by convention. Like BDDs, FDDs respect a total order on tests and contain no isomorphic subgraphs or redundant tests, which enables representing sparse matrices compactly.

Dynamic domain reduction. As Figure 4.4 shows, we do not have to represent the state space $P_k + \emptyset$ explicitly even when converting into sparse matrix form. In the example, the state space is represented by *symbolic packets* $pt = 1$, $pt = 2$, $pt = 3$, and $pt = *$, each representing an *equivalence class* of packets. For example, $pt = 1$ can represent all packets π satisfying $\pi.pt = 1$, because the program treats all such packets in the same way. The packet $pt = *$ represents the set $\{\pi \mid \pi.pt \notin \{1, 2, 3\}\}$. The symbol $*$ can be thought of as a wildcard that ranges over all values not explicitly represented by other symbolic packets. The symbolic packets are chosen dynamically when converting an FDD to a matrix by traversing the FDD and determining the set of values appearing in each field, either in a test or a modification. Since FDDs never contain redundant tests or modifications, these sets are typically of manageable size.

4.5.2 PRISM Backend

PRISM is a mature probabilistic model checker that has been actively developed and improved for the last two decades. The tool takes as input a Markov chain model specified symbolically in PRISM’s input language and a property specified using a logic such as Probabilistic CTL, and outputs the probability that the model satisfies the property. PRISM supports various types of models including finite state Markov chains, and can thus be used as a backend for reasoning about ProbNetKAT programs using our results from Section 4.3 and Section 4.4. Accordingly, we implemented a second backend that translates ProbNetKAT to PRISM programs. While the native backend computes the big step semantics of a program—a costly operation that may involve solving linear systems to compute fixed points—the PRISM backend is a purely syntactic transformation; the heavy lifting is done by PRISM itself.

A PRISM program consists of a set of bounded variables together with a set of transition rules of the form

$$\varphi \rightarrow p_1 \cdot u_1 + \cdots + p_k \cdot u_k$$

where φ is a Boolean predicate over the variables, the p_i are probabilities that must sum up to one, and the u_i are sequences of variable updates. The predicates are required to be mutually exclusive and exhaustive. Such a program encodes a Markov chain whose state space is given by the finite set of variable assignments and whose transitions are dictated by the rules: if φ is satisfied under the current assignment σ and σ_i is obtained from σ by performing update u_i , then the probability of a transition from σ to σ_i is p_i .

It is easy to see that any PRISM program can be expressed in ProbNetKAT, but the reverse direction is slightly tricky: it requires the introduction of an additional variable akin to a program counter to emulate ProbNetKAT’s control flow primitives such as loops and sequences. As an additional challenge, we must be economical in our allocation of

the program counter, since the performance of model checking is very sensitive to the size of the state space.

We address this challenge in three steps. First, we translate the ProbNetKAT program to a finite state machine using a Thompson-style construction [169]. Each edge is labeled with a predicate φ , a probability p_i , and an update u_i , subject to the following well-formedness conditions:

1. For each state, the predicates on its outgoing edges form a partition.
2. For each state and predicate, the probabilities of all outgoing edges guarded by that predicate sum to one.

Intuitively, the state machine encodes the control-flow graph.

This intuition serves as the inspiration for the next translation step, which collapses each basic block of the graph into a single state. This step is crucial for reducing the state space, since the state space of the initial automaton is linear in the size of the program. Finally, we obtain a PRISM program from the automaton as follows: for each state s with adjacent predicate φ and φ -guarded outgoing edges $s \xrightarrow{\varphi/p_i/u_i} t_i$ for $1 \leq i \leq k$, produce a PRISM rule

$$(pc=s \wedge \varphi) \rightarrow p_1 \cdot (u_1 \cdot pc \leftarrow t_1) + \dots + p_k \cdot (u_k \cdot pc \leftarrow t_k).$$

The well-formedness conditions of the state machine guarantee that the resulting program is a valid PRISM program. With some care, the entire translation can be implemented in linear time. Indeed, McNetKAT translates all programs in our evaluation to PRISM in under a second.

4.6 Evaluation

To evaluate McNetKAT we conducted experiments on several benchmarks including a family of real-world data center topologies and a synthetic benchmark drawn from the

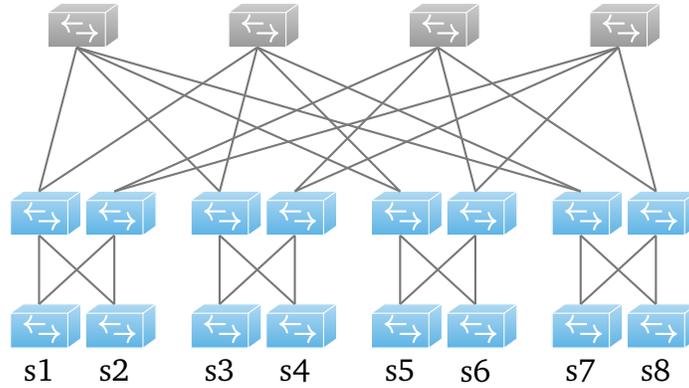


Figure 4.5: A FatTree topology with $p = 4$ ports per switch.

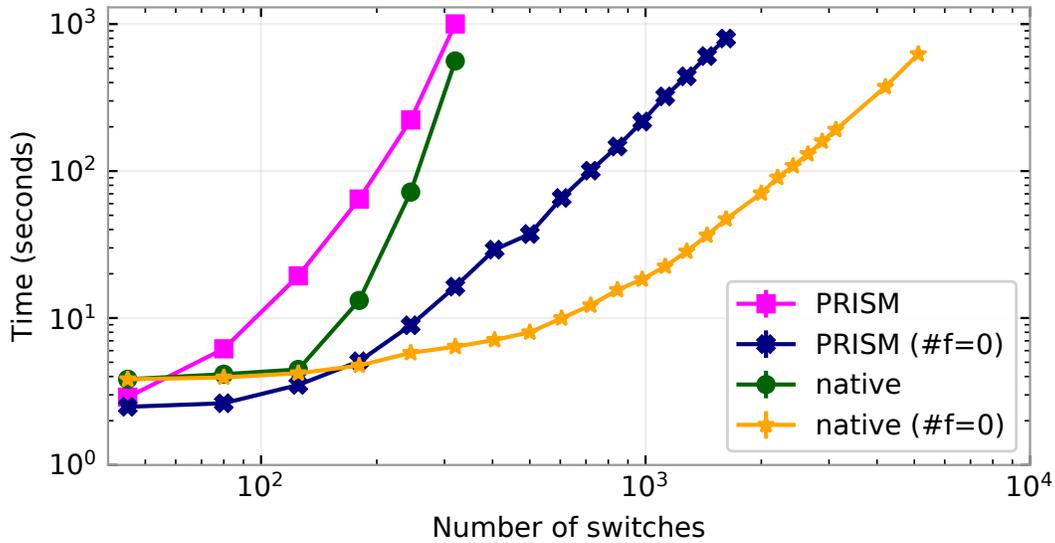


Figure 4.6: Scalability on a family of data center topologies.

literature [49]. We evaluated McNetKAT’s scalability, characterized the effect of optimizations, and compared performance against other state-of-the-art tools. All McNetKAT running times we report refer to the time needed to compile programs to FDDs; the cost of comparing FDDs for equivalence and ordering, or of computing statistics of the encoded distributions, is negligible. All experiments were performed on machines with 16-core, 2.6 GHz Intel Xeon E5-2650 processors with 64 GB of memory.

Scalability on FatTree topologies. We first measured the scalability of McNetKAT by using it to compute network models for a series of FatTree topologies of increasing size. FatTrees [3] (see also Figure 4.5) are multi-level, multi-rooted trees that are widely used as topologies in modern data centers. FatTrees can be specified in terms of a parameter p corresponding to the number of ports on each switch. A p -ary FatTree connects $\frac{1}{4}p^3$ servers using $\frac{5}{4}p^2$ switches. To route packets, we used a form of Equal-Cost Multipath Routing (ECMP) that randomly maps traffic flows onto shortest paths. We measured the time needed to construct the stochastic matrix representation of the program on a single machine using two backends (native and PRISM) and under two failure models (no failures and independent failures with probability $1/1000$).

Figure 4.6 depicts the results, several of which are worth discussing. First, the native backend scales quite well: in the absence of failures ($f = 0$), it scales to a network with 5000 switches in approximately 10 minutes. This result shows that McNetKAT is able to handle networks of realistic size. Second, the native backend consistently outperforms the PRISM backend. We conjecture that the native backend is able to exploit algebraic properties of the ProbNetKAT program to better parallelize the job. Third, performance degrades in the presence of failures. This is to be expected—failures lead to more complex probability distributions which are nontrivial to represent and manipulate.

Parallel speedup. One of the contributors to McNetKAT’s good performance is its ability to parallelize the computation of stochastic matrices across multiple cores in a machine, or even across machines in a cluster. Intuitively, because a network is a large collection of mostly independent devices, it is possible to model its global behavior by first modeling the behavior of each device in isolation, and then combining the results to obtain a network-wide model. In addition to speeding up the computation, this approach can also reduce memory usage, often a bottleneck on large inputs.

To facilitate parallelization, we added an n -ary disjoint branching construct to

ProbNetKAT:

```
case sw=1 then p1 else
case sw=2 then p2 else
...
case sw=n then pn
```

Semantically, this construct is equivalent to a cascade of conditionals; but the native backend compiles it in parallel using a map-reduce-style strategy, using one process per core by default.

To evaluate the impact of parallelization, we compiled two representative FatTree models ($p = 14$ and $p = 16$) using ECMP routing on an increasing number of cores. With m cores, we used one master machine together with $r = \lceil m/16 - 1 \rceil$ remote machines, adding machines one by one as needed to obtain more physical cores. The results are shown in Figure 4.7. We see near linear speedup on a single machine, cutting execution time by more than an order of magnitude on our 16-core test machine. Beyond a single machine, the speedup depends on the complexity of the submodels for each switch—the longer it takes to generate the matrix for each switch, the higher the speedup. For example, with a $p = 16$ FatTree, we obtained a 30x speedup using 40 cores across 3 machines.

Comparison with other tools. Bayonet [49] is a state-of-the-art tool for analyzing probabilistic networks. Whereas McNetKAT has a native backend tailored to the networking domain and a backend based on a probabilistic model checker, Bayonet programs are translated to a general-purpose probabilistic language which is then analyzed by the symbolic inference engine PSI [50]. Bayonet’s approach is more general, as it can model queues, state, and multi-packet interactions under an asynchronous scheduling model. It also supports Bayesian inference and parameter synthesis. Moreover, Bayonet

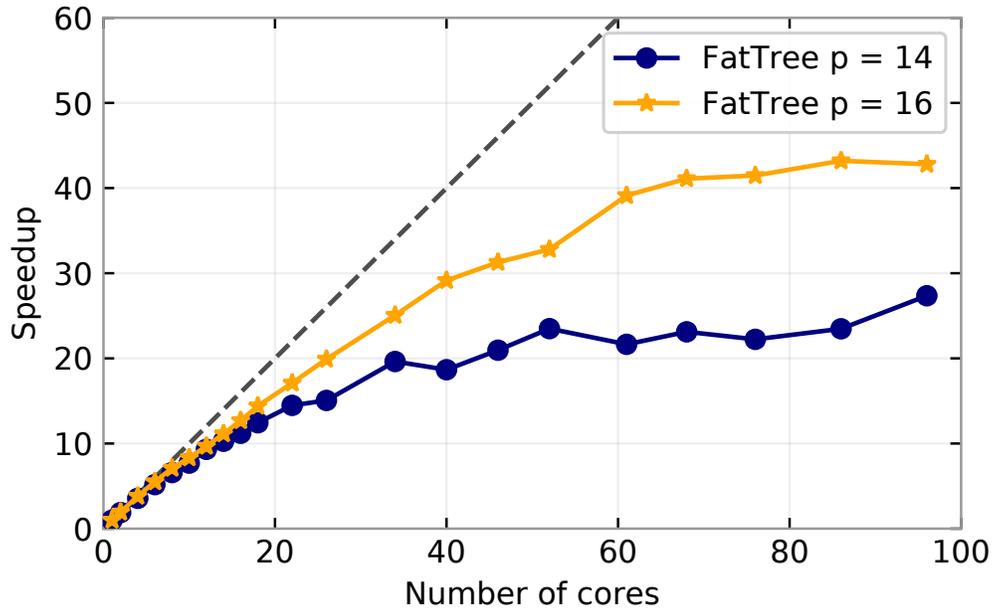


Figure 4.7: Speedup due to parallelization.

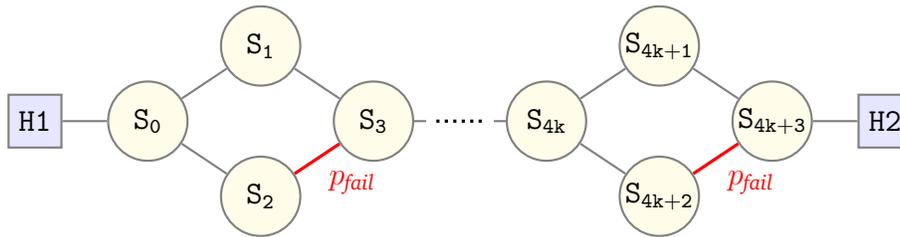


Figure 4.8: Chain topology

is fully symbolic whereas McNetKAT uses a numerical linear algebra solver [30] (based on floating point arithmetic) to compute limits.

To evaluate how the performance of these approaches compares, we reproduced an experiment from the Bayonet paper that analyzes the reliability of a simple routing scheme in a family of “chain” topologies indexed by k , as shown in Figure 4.8.

For $k = 1$, the network consists of four switches organized into a diamond, with a single link that fails with probability $p_{fail} = 1/1000$. For $k > 1$, the network consists of k diamonds linked together into a chain as shown in Figure 4.8. Within each diamond, switch S_0 forwards packets with equal probability to switches S_1 and S_2 , which in

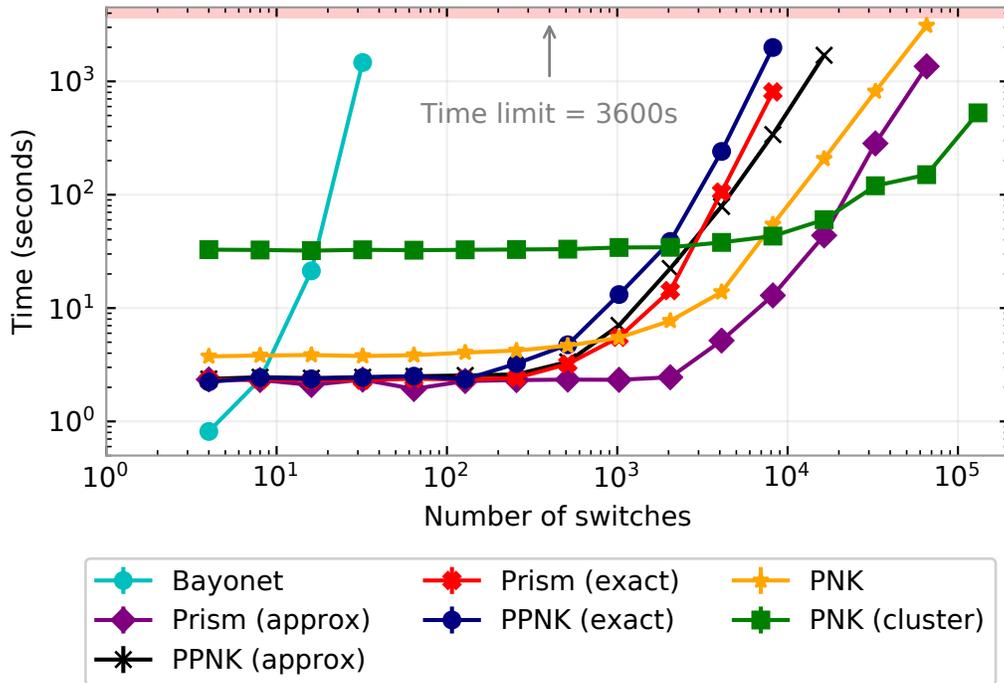


Figure 4.9: Scalability on chain topology.

turn forward to switch S_3 . However, S_2 drops the packet if the link to S_3 fails. We analyze the probability that a packet originating at H1 is successfully delivered to H2. Our implementation does not exploit the regularity of these topologies.

Figure 4.9 gives the running time for several tools on this benchmark: Bayonet, hand-written PRISM, ProbNetKAT with the PRISM backend (PPNK), and ProbNetKAT with the native backend (PNK). Further, we ran the PRISM tools in exact and approximate mode, and we ran the ProbNetKAT backend on a single machine and on the cluster. Note that both axes in the plot are log-scaled.

We see that Bayonet scales to 32 switches in about 25 minutes, before hitting the one hour time limit and 64 GB memory limit at 48 switches. ProbNetKAT answers the same query for 2048 switches in under 10 seconds and scales to over 65000 switches in about 50 minutes on a single core, or just 2.5 minutes using a cluster of 24 machines. PRISM scales similarly to ProbNetKAT, and performs best using the hand-written model

in approximate mode.

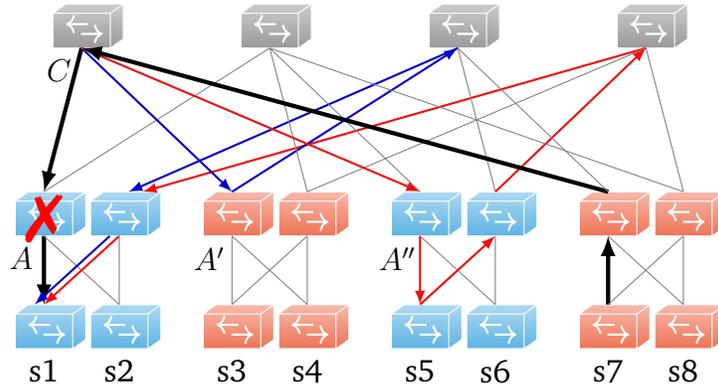
Overall, this experiment shows that for basic network verification tasks, ProbNetKAT’s domain-specific backend based on specialized data structures and an optimized linear-algebra library [30] can outperform an approach based on a general-purpose solver.

4.7 Case Study: Data Center Fault-Tolerance

In this section, we go beyond benchmarks and present a case study that illustrates the utility of McNetKAT for probabilistic reasoning. Specifically, we model the F10 [106] data center design in ProbNetKAT and verify its key properties.

Data center resilience. An influential measurement study by Gill et al. [51] showed that data centers experience frequent failures, which have a major impact on application performance. To address this challenge, a number of data center designs have been proposed that aim to simultaneously achieve high throughput, low latency, and fault tolerance.

F10 topology. F10 uses a novel topology called an *AB FatTree*, see Figure 4.10(a), that enhances a traditional FatTree [3] with additional backup paths that can be used when failures occur. To illustrate, consider routing from s_7 to s_1 in Figure 4.10(a) along one of the shortest paths (in thick black). After reaching the core switch C in a standard FatTree (recall Figure 4.5), if the aggregation switch on the downward path failed, we would need to take a 5-hop detour (shown in red) that goes down to a different edge switch, up to a different core switch, and finally down to s_1 . In contrast, an AB FatTree [106] modifies the wiring of the aggregation later to provide shorter detours—e.g., a 3-hop detour (shown in blue) for the previous scenario.



(a) AB FatTree topology with $p = 4$.

k	$\widehat{M}(\text{F10}_0, f_k)$ \equiv <i>teleport</i>	$\widehat{M}(\text{F10}_3, f_k)$ \equiv <i>teleport</i>	$\widehat{M}(\text{F10}_{3,5}, f_k)$ \equiv <i>teleport</i>
0	✓	✓	✓
1	✗	✓	✓
2	✗	✓	✓
3	✗	✗	✓
4	✗	✗	✗
∞	✗	✗	✗

(b) Evaluating k -resilience.

k	compare F10_0 F10_3	compare F10_3 $\text{F10}_{3,5}$	compare $\text{F10}_{3,5}$ <i>teleport</i>
0	\equiv	\equiv	\equiv
1	<	\equiv	\equiv
2	<	\equiv	\equiv
3	<	<	\equiv
4	<	<	<
∞	<	<	<

(c) Comparing schemes under k failures.

Figure 4.10: Resilience of different schemes on AB FatTree topology.

F10 routing. F10’s routing scheme uses three strategies to re-route packets after a failure occurs. If a link on the current path fails and an equal-cost path exists, the switch simply re-routes along that path. This approach is also known as *equal-cost multi-path routing* (ECMP). If no shortest path exist, it uses a 3-hop detour if one is available, and otherwise falls back to a 5-hop detour if necessary.

We implemented this routing scheme in ProbNetKAT in several steps. The first, F10_0 , approximates the hashing behavior of ECMP by randomly selecting a port along one of the shortest paths to the destination. The second, F10_3 , improves the resilience of F10_0 by augmenting it with 3-hop re-routing—e.g., consider the blue path in Figure 4.10(a).

We find a port on C that connects to a different aggregation switch A' and forward the packet to A' . If there are multiple such ports which have not failed, we choose one uniformly at random. The third, $F10_{3,5}$, attempts 5-hop re-routing in cases where $F10_3$ is unable to find a port on C whose adjacent link is up—e.g., consider the red path in Figure 4.10(a). The 5-hop rerouting strategy requires a flag to distinguish packets taking a detour from regular packets.

F10 network and failure model. We model the network as discussed in Section 4.2, focusing on packets destined to switch 1:

$$M(p) := in \cdot \mathbf{do} (p \cdot t) \mathbf{while} (\neg sw=1)$$

McNetKAT automatically generates the topology program t from a Graphviz description. The ingress predicate in is a disjunction of switch-port tests over all ingress locations. Adding the failure model and some setup code to declare local variables tracking the health of individual links yields the complete network model:

$$\widehat{M}(p, f) := \mathbf{var} up_1 \leftarrow 1 \mathbf{in} \dots \mathbf{var} up_d \leftarrow 1 \mathbf{in} M(f \cdot p)$$

Here, d is the maximum degree of a topology node. The entire model measures about 750 lines of ProbNetKAT code.

To evaluate the effect of different kinds of failures, we define a family of failure models f_k indexed by the maximum number of failures $k \in \mathbb{N} \cup \{\infty\}$ that may occur, where links fail otherwise independently with probability pr ; we leave pr implicit. To simplify the analysis, we focus on failures occurring on downward paths (note that $F10_0$ is able to route around failures on the upward path, unless the topology becomes disconnected).

Verifying refinement. Having implemented F10 as a series of three refinements, we would expect the probability of packet delivery to increase in each refinement, but not to

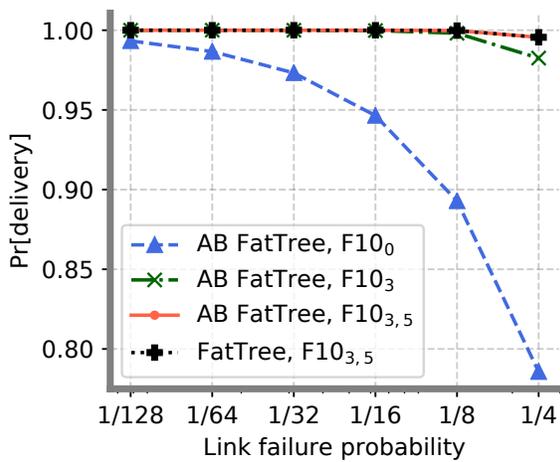
achieve perfect delivery in an unbounded failure model f_∞ . Formally, we should have

$$\begin{aligned} \text{drop} &< \widehat{M}(\text{F10}_0, f_\infty) < \widehat{M}(\text{F10}_3, f_\infty) \\ &< \widehat{M}(\text{F10}_{3,5}, f_\infty) < \text{teleport} \end{aligned}$$

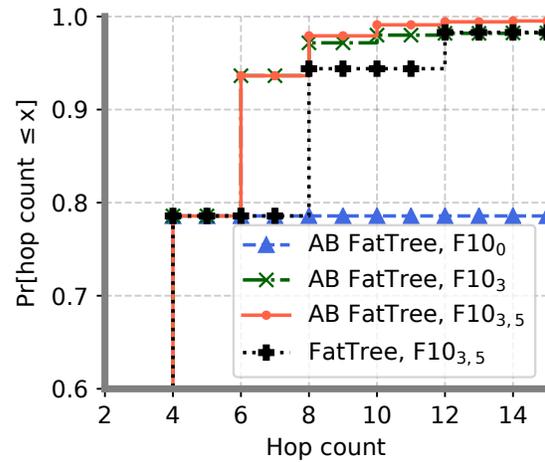
where *teleport* moves the packet directly to its destination, and $p < q$ means the probability assigned to every input-output pair by q is greater than the probability assigned by p . We confirmed that these inequalities hold using McNetKAT.

Verifying k-resilience. Resilience is the key property satisfied by F10. By using McNetKAT, we were able to automatically verify that F10 is resilient to up to three failures in the AB FatTree Figure 4.10(a). To establish this property, we increased the parameter k in our failure model f_k while checking equivalence with teleportation (i.e., perfect delivery), as shown in Figure 4.10(b). The simplest scheme F10₀ drops packets when a failure occurs on the downward path, so it is 0-resilient. The F10₃ scheme routes around failures when a suitable aggregation switch is available, hence it is 2-resilient. Finally, the F10_{3,5} scheme routes around failures as long as any aggregation switch is reachable, hence it is 3-resilient. If the schemes are not equivalent to *teleport*, we can still compare the relative resilience of the schemes using the refinement order, as shown in Figure 4.10(c). Our implementation also enables precise, quantitative comparisons. For example, Figure 4.11(a) considers a failure model in which an unbounded number of failures can occur. We find that F10₀'s delivery probability dips significantly as the failure probability increases, while both F10₃ and F10_{3,5} continue to ensure high delivery probability by routing around failures.

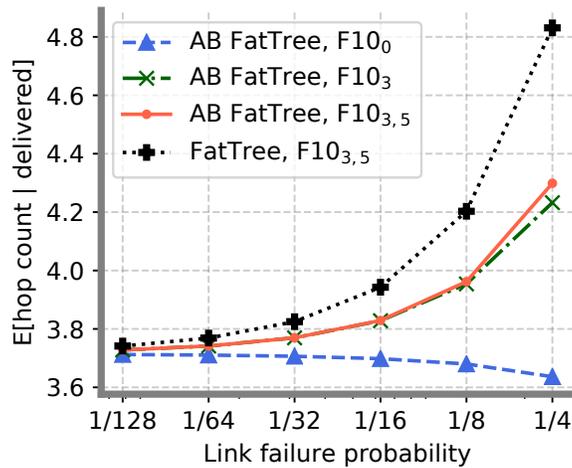
Analyzing path stretch. Routing schemes based on detours achieve a higher degree of resilience at the cost of increasing the lengths of forwarding paths. We can quantify this increase by augmenting our model with a counter that is incremented at each hop and



(a) Probability of delivery vs. link-failure probability.



(b) Increased path length due to resilience ($pr = 1/4$).



(c) Expected of hop-count for delivered packets.

Figure 4.11: Case study results for unbounded number of failures ($k = \infty$).

analyzing the expected path length. Figure 4.11(b) shows the cumulative distribution function of latency as the fraction of traffic delivered within a given hop count. On AB FatTree, F10₀ delivers $\approx 80\%$ of the traffic in 4 hops, since the maximum length of a shortest path from any edge switch to s_1 is 4 and F10₀ does not attempt to recover from failures. F10₃ and F10_{3,5} deliver the same amount of traffic when limited to at most 4 hops, but they can deliver significantly more traffic using 2 additional hops by using

3-hop and 5-hop paths to route around failures. $F10_3$ also delivers more traffic with 8 hops—these are the cases when $F10_3$ performs 3-hop re-routing twice for a single packet as it encountered failure twice. We can also show that on a standard FatTree, $F10_{3,5}$ failures have a higher impact on latency. Intuitively, the topology does not support 3-hop re-routing. This finding supports a key claim of F10: the topology and routing scheme should be co-designed to avoid excessive path stretch. Finally, Figure 4.11(c) shows the expected path length conditioned on delivery. As the failure probability increases, the probability of delivery for packets routed via the core layer decreases for $F10_0$. Thus, the distribution of delivered packets shifts towards 2-hop paths via an aggregation switch, so the expected hop-count decreases.

4.8 Related Work

The most closely related system to McNetKAT is Bayonet [49]. In contrast to the domain-specific approach followed in this chapter, Bayonet uses a general-purpose probabilistic programming language and inference tool [50]. Such an approach, which reuses existing techniques, is naturally appealing. In addition, Bayonet is more expressive than McNetKAT: it supports asynchronous scheduling, stateful transformations, and probabilistic inference, making it possible to model richer phenomena, such as congestion due to packet-level interactions in queues. Of course, the extra generality does not come for free. Bayonet requires programmers to supply an upper bound on loops as the implementation is not guaranteed to find a fixed point. As discussed in Section 4.5, McNetKAT scales better than Bayonet on simple benchmarks. Another issue is that writing a realistic scheduler appears challenging, and one might also need to model host-level congestion control protocols to obtain accurate results. Currently Bayonet programs use deterministic or uniform schedulers and model only a few packets at a time [48].

Prior work on ProbNetKAT (Chapter 3) gave a measure-theoretic semantics and an implementation that approximated programs using sequences of monotonically improving estimates. While these estimates were proven to converge in the limit, (Chapter 3) offered no guarantees about the convergence rate. In fact, there are examples where the approximations do not converge after any finite number of steps, which is obviously undesirable in a tool. The implementation only scaled to 10s of switches. In contrast, this chapter presents a straightforward and implementable semantics; the implementation computes limits precisely in closed form, and it scales to real-world networks with thousands of switches. McNetKAT achieves this by restricting to the guarded and history-free fragment of ProbNetKAT, sacrificing the ability to reason about multicast and path-properties directly. In practice this sacrifice seems well worth the payoff: multicast is somewhat uncommon, and we can often reason about path-properties by maintaining extra state in the packets. In particular, McNetKAT can still model the examples studied in Chapter 3.

Our work is the latest in a long line of techniques using Markov chains as a tool for representing and analyzing probabilistic programs. For an early example, see the seminal paper of Sharir et al. [159]. Markov chains are also used in many probabilistic model checkers, such as PRISM [100].

Beyond networking applications, there are connections to other work on verification of probabilistic programs. Di Pierro, Hankin, and Wicklicky used probabilistic abstract interpretation to statically analyze probabilistic λ -calculus [31]; their work was extended to a language *pWhile*, using a store and program location state space similar to Sharir et al. [159]. However, they do not deal with infinite limiting behavior beyond stepwise iteration, and do not guarantee convergence. Olejnik, Wicklicky, and Cheraghchi provided a probabilistic compiler *pwc* for a variation of *pWhile* [126]; their optimizations could potentially be useful for McNetKAT. A recent survey by Gordon et al. [54] shows how to

give semantics for probabilistic processes using stationary distributions of Markov chains, and studies convergence. Similar to our approach, they use absorbing strongly connected components to represent termination. Finally, probabilistic abstract interpretation is also an active area of research [175]; it would be interesting to explore applications to ProbNetKAT.

4.9 Conclusion

This chapter presents a scalable tool for verifying probabilistic networks based on a new semantics for the history-free fragment of ProbNetKAT in terms of Markov chains. Natural directions for future work include further optimization of our implementation—e.g., using Bayesian networks to represent joint distributions compactly. We are also interested in applying McNetKAT to other systems that implement algorithms for randomized routing [99, 163], load balancing [33], traffic monitoring [158], anonymity [32], and network neutrality [179], among others.

Part III

A Family of Programming Languages

Chapter 5

Guarded Kleene Algebra with Tests

“We must think in terms, not of languages, but of families of languages.”

—Peter J. Landin

“Mechanical reasoning is used in program manipulation to verify routine facts or to catch slippery errors, not to prove mathematically interesting theorems.”

—Greg Nelson and Derek C. Oppen

Guarded Kleene Algebra with Tests (GKAT) is a variation on Kleene Algebra with Tests (KAT) that arises by restricting the union (+) and iteration (*) operations from KAT to predicate-guarded versions. We develop the (co)algebraic theory of GKAT and show how it can be efficiently used to reason about imperative programs. In contrast to KAT, whose equational theory is PSPACE-complete, we show that the equational theory of GKAT is (almost) linear time. We also provide a full Kleene theorem and prove completeness for an analogue of Salomaa’s axiomatization of Kleene Algebra.

5.1 Introduction

Computer scientists have long explored the connections between families of programming languages and abstract machines. This dual perspective has furnished deep theoretical insights as well as practical tools. As an example, Kleene’s classic result establishing

the equivalence of regular expressions and finite automata [81] inspired decades of work across a variety of areas including programming language design, mathematical semantics, and formal verification.

Kleene Algebra with Tests (KAT) [89], which combines Kleene Algebra (KA) with Boolean Algebra (BA), is a modern example of this approach. Viewed from the program-centric perspective, a KAT models the fundamental constructs that arise in programs: sequencing, branching, iteration, non-determinism, etc. The equational theory of KAT enables algebraic reasoning and can be finitely axiomatized [96]. Viewed from the machine-centric perspective, a KAT describes a kind of automaton that generates a regular language of traces. This shift in perspective admits techniques from the theory of coalgebras for reasoning about program behavior. In particular, there are efficient algorithms for checking bisimulation, which can be optimized using properties of bisimulations [18, 65] or symbolic automata representations [138].

KAT has been used to model computation across a wide variety of areas including program transformations [7, 91], concurrency control [25], compiler optimizations [95], cache control [12, 24], and more [24]. A prominent recent application is NetKAT [6], a language for reasoning about the packet-forwarding behavior of software-defined networks. NetKAT has a sound and complete equational theory, and a coalgebraic decision procedure that can be used to automatically verify many important networking properties including reachability, loop-freedom, and isolation [45]. However, while NetKAT's implementation scales well in practice, the complexity of deciding equivalence for NetKAT is PSPACE-complete in the worst case [6].

A natural question to ask is whether there is an efficient fragment of KAT that is reasonably expressive, while retaining a solid foundation. We answer this question positively in this paper with a comprehensive study of Guarded Kleene Algebra with Tests (GKAT), the guarded fragment of KAT. The language is a propositional abstraction of

imperative while programs, which have been well-studied in the literature. We establish the fundamental properties of GKAT and develop its algebraic and coalgebraic theory. GKAT replaces the union ($e + f$) and iteration (e^*) constructs in KAT with guarded versions: conditionals ($e +_b f$) and loops ($e^{(b)}$) guarded by Boolean predicates b . The resulting language is a restriction of full KAT, but it is sufficiently expressive to model typical, imperative programs—e.g., essentially all NetKAT programs needed to solve practical verification problems can be expressed as guarded programs.

In exchange for a modest sacrifice in expressiveness, GKAT offers two significant advantages. First, equivalence is decidable in *nearly linear time*—a substantial improvement over the PSPACE complexity for KAT [26]. Specifically, any GKAT expression e can be represented as a deterministic automaton of size $\mathcal{O}(|e|)$, while KAT expressions can require as many as $\mathcal{O}(2^{|e|})$ states. As a consequence, any property that is efficiently decidable for deterministic automata is also efficiently decidable for GKAT. Second, we believe that GKAT is a better foundation for probabilistic languages due to well-known issues that arise when combining non-determinism—which is native to KAT—with probabilistic choice [114, 173]. For example, ProbNetKAT [44], a probabilistic extension of NetKAT, does not satisfy the KAT axioms, but its guarded restriction forms a proper GKAT.

Although GKAT is a simple restriction of KAT at the syntactic level, the semantics of the language is surprisingly subtle. In particular, the “obvious” notion of GKAT automata can encode behaviors that would require non-local control-flow operators (e.g, **goto** or multi-level **break** statements) [98]. In contrast, GKAT models programs whose control-flow always follows a lexical, nested structure. To overcome this discrepancy, we identify a set of restrictions on automata to enable an analogue of Kleene’s theorem—every GKAT automaton satisfying our restrictions can be converted to a program, and vice versa. Besides the theoretical interest in this result, we believe it may also have practical applications, such as reasoning about optimizations in a compiler for an imperative

language [63]. We also develop an equational axiomatization for GKAT and prove that it is sound and complete over a coequationally-defined language model. The main challenge is that without $+$, the natural order \leq on KAT programs can no longer be used to axiomatize a least fixpoint. We instead axiomatize a unique fixed point, in the style of Salomaa’s work on Kleene Algebra [152].

Outline. We make the following contributions in this paper.

- We initiate a comprehensive study of GKAT, a guarded version of KAT, and show how GKAT models relational and probabilistic programming languages (Section 5.2).
- We give a new construction of linear-size automata from GKAT programs (Section 5.4). As a consequence, the equational theory of GKAT is decidable in nearly linear time (Section 5.5).
- We identify a class of automata representable as GKAT expressions (Section 5.4) that contains all automata produced by the previous construction, yielding a Kleene theorem.
- We present axioms for GKAT (Section 5.3) and prove that our axiomatization is complete for equivalence with respect to a coequationally-defined language model (Section 5.6).

Omitted proofs can be found in the appendix.

5.2 Overview: An Abstract Programming Language

This section introduces the syntax and semantics of GKAT, an abstract programming language with uninterpreted actions. Using examples, we show how GKAT can model relational and probabilistic programming languages—*i.e.*, by giving actions a concrete interpretation. An equivalence between abstract GKAT programs thus implies a corresponding equivalence between concrete programs.

5.2.1 Syntax

The syntax of GKAT is parameterized by abstract sets of *actions* Σ and *primitive tests* T , where Σ and T are assumed to be disjoint and nonempty, and T is assumed to be finite. We reserve p and q to range over actions. The language consists of Boolean expressions, BExp , and GKAT expressions, Exp , as defined by the following grammar:

$b, c, d \in \text{BExp} ::=$ 0 false 1 true $t \in T$ t $b \cdot c$ b and c $b + c$ b or c \bar{b} not b	$e, f, g \in \text{Exp} ::=$ $p \in \Sigma$ do p $b \in \text{BExp}$ assert b $e \cdot f$ $e; f$ $e +_b f$ if b then e else f $e^{(b)}$ while b do e
--	--

The algebraic notation on the left is more convenient when manipulating terms, while the notation on the right may be more intuitive when writing programs. We often abbreviate $e \cdot f$ by ef , and omit parentheses following standard conventions, e.g., writing $bc + d$ instead of $(bc) + d$ and $ef^{(b)}$ instead of $e(f^{(b)})$.

5.2.2 Semantics: Language Model

Intuitively, we interpret a GKAT expression as the set of “legal” execution traces it induces, where a trace is legal if no assertion fails. To make this formal, let $b \equiv_{\text{BA}} c$ denote Boolean equivalence. Entailment is a preorder on the set of Boolean expressions, BExp , and can be characterized in terms of equivalence as follows: $b \leq c \iff b + c \equiv_{\text{BA}} c$. In the quotient set $\text{BExp} / \equiv_{\text{BA}}$ (the *free Boolean algebra* on generators $T = \{t_1, \dots, t_n\}$), entailment is a partial order $[b]_{\equiv_{\text{BA}}} \leq [c]_{\equiv_{\text{BA}}} \iff b + c \equiv_{\text{BA}} c$, with minimum and maximum elements given by the equivalence classes of 0 and 1, respectively. The minimal nonzero elements of this order are called *atoms*. We let At denote the set of atoms and use lowercase

Greek letters α, β, \dots to denote individual atoms. Each atom is the equivalence class of an expression of the form $c_1 \cdot c_2 \cdots c_n \in \text{BExp}$ with $c_i \in \{t_i, \bar{t}_i\}$. Thus we can think of each atom as representing a truth assignment on T , e.g., if $c_i = t_i$ then t_i is set to true, otherwise if $c_i = \bar{t}_i$ then t_i is set to false. Likewise, the set $\{\alpha \in \text{At} \mid \alpha \leq b\}$ can be thought of as the set of truth assignments where b evaluates to true; \equiv_{BA} is *complete* w.r.t. this interpretation in that two Boolean expressions are related by \equiv_{BA} if and only if their atoms coincide [16].

A *guarded string* is an element of the regular set $\text{GS} := \text{At} \cdot (\Sigma \cdot \text{At})^*$. Intuitively, a non-empty string $\alpha_0 p_1 \alpha_1 \cdots p_n \alpha_n \in \text{GS}$ describes a trace of an abstract program: the atoms α_i describe the state of the system at various points in time, starting from an initial state α_0 and ending in a final state α_n , while the actions $p_i \in \Sigma$ are the triggering transitions between the various states. Given two traces, we can combine them sequentially by running one after the other. Formally, guarded strings compose via a partial *fusion product* $\diamond: \text{GS} \times \text{GS} \rightarrow \text{GS}$, defined for $x, y \in (\text{At} \cup \Sigma)^*$ as

$$x\alpha \diamond \beta y := \begin{cases} x\alpha y & \text{if } \alpha = \beta \\ \text{undefined} & \text{otherwise.} \end{cases}$$

This product lifts to a total function on languages $L, K \subseteq \text{GS}$ of guarded strings, given by

$$L \diamond K := \{x \diamond y \mid x \in L, y \in K\}.$$

We need a few more constructions before we can interpret GKAT expressions as languages representing their possible traces. First, 2^{GS} with the fusion product forms a monoid with identity At and so we can define the n -th power L^n of a language L inductively in the usual way:

$$L^0 := \text{At} \qquad L^{n+1} := L^n \diamond L$$

Second, in the special case where $B \subseteq \text{At}$, we write \bar{B} for $\text{At} - B$ and define:

$$L +_B K := (B \diamond L) \cup (\bar{B} \diamond K) \qquad L^{(B)} := \bigcup_{n \geq 0} (B \diamond L)^n \diamond \bar{B}$$

We are now ready to interpret GKAT expressions as languages of guarded strings via the semantic map $\llbracket - \rrbracket : \text{Exp} \rightarrow 2^{\text{GS}}$ as follows:

$$\begin{aligned} \llbracket p \rrbracket &:= \{\alpha p \beta \mid \alpha, \beta \in \text{At}\} & \llbracket e \cdot f \rrbracket &:= \llbracket e \rrbracket \diamond \llbracket f \rrbracket & \llbracket e^{(b)} \rrbracket &:= \llbracket e \rrbracket^{(\llbracket b \rrbracket)} \\ \llbracket b \rrbracket &:= \{\alpha \in \text{At} \mid \alpha \leq b\} & \llbracket e +_b f \rrbracket &:= \llbracket e \rrbracket +_{\llbracket b \rrbracket} \llbracket f \rrbracket \end{aligned}$$

We call this the *language model* of GKAT. Since we make no assumptions about the semantics of actions, we interpret them as sets of traces beginning and ending in arbitrary states; this soundly overapproximates the behavior of any instantiation. A test is interpreted as the set of states satisfying the test. The traces of $e \cdot f$ are obtained by composing traces from e with traces from f in all possible ways that make the final state of an e -trace match up with the initial state of an f -trace. The traces of $e +_b f$ collect traces of e and f , restricting to e -traces whose initial state satisfies b and f -traces whose initial state satisfies \bar{b} . The traces of $e^{(b)}$ are obtained by sequentially composing zero or more be -traces and selecting only traces ending in a state satisfying \bar{b} .

Remark 5.2.1 (Connection to KAT). The expressions for KAT, denoted KExp , are generated by the same grammar as for GKAT, except that KAT's union ($+$) replaces GKAT's guarded union ($+_b$) and KAT's iteration (e^*) replaces GKAT's guarded iteration ($e^{(b)}$). GKAT's guarded operators can be encoded in KAT; this encoding, which goes back to early work on Propositional Dynamic Logic [41], is the standard method to model conditionals and while loops:

$$e +_b f = be + \bar{b}f \qquad e^{(b)} = (be)^* \bar{b}$$

In other words, there is a homomorphic map $\varphi : \text{Exp} \rightarrow \text{KExp}$ from GKAT to KAT expressions. We inherit KAT's language model [96], $\mathcal{K}[\llbracket - \rrbracket] : \text{KExp} \rightarrow 2^{\text{GS}}$, in the following sense: $\llbracket - \rrbracket = \mathcal{K}[\llbracket - \rrbracket] \circ \varphi$. □

Definition 5.2.2 (Deterministic languages). The languages denoted by GKAT programs satisfy a *determinacy property*: whenever x, y are in the language and x and y agree on

their first n atoms, then they agree on their first n actions (or lack thereof). For example, $\{\alpha p \gamma, \beta q \delta\}$ and $\{\alpha p \gamma, \beta\}$ for $\alpha \neq \beta$ satisfy the determinacy property, while $\{\alpha p \beta, \alpha\}$ and $\{\alpha p \beta, \alpha q \delta\}$ for $p \neq q$ do not. \square

We say that two expressions e and f are *equivalent* if they have the same semantics—*i.e.*, if $\llbracket e \rrbracket = \llbracket f \rrbracket$. In the following sections, we show that this notion of equivalence

- is sound and complete for relational and probabilistic interpretations (Section 5.2.3 and Section 5.2.4),
- can be finitely and equationally axiomatized in a sound (Section 5.3) and complete (Section 5.6) way, and
- is efficiently decidable in time nearly linear in the sizes of the expressions (Section 5.5).

5.2.3 Relational Model

This subsection gives an interpretation of GKAT expressions as binary relations, a common model of input-output behavior for many programming languages. We show that the language model is sound and complete for this interpretation. Thus GKAT equivalence implies program equivalence for any programming language with a suitable relational semantics.

Definition 5.2.3 (Relational Interpretation). Let $i = (\text{State}, \text{eval}, \text{sat})$ be a triple consisting of

- a set of *states* State ,
- for each action $p \in \Sigma$, a binary relation $\text{eval}(p) \subseteq \text{State} \times \text{State}$, and
- for each primitive test $t \in T$, a set of states $\text{sat}(t) \subseteq \text{State}$.

Then the *relational interpretation* of an expression e with respect to i is the smallest binary relation $\mathcal{R}_i[[e]] \subseteq \text{State} \times \text{State}$ satisfying the following rules,

$$\begin{array}{c}
\frac{}{\mathcal{R}_i[[p]] = \text{eval}(p)} \qquad \frac{\sigma \in \text{sat}^\dagger(b)}{(\sigma, \sigma) \in \mathcal{R}_i[[b]]} \qquad \frac{(\sigma, \sigma') \in \mathcal{R}_i[[e]] \quad (\sigma', \sigma'') \in \mathcal{R}_i[[f]]}{(\sigma, \sigma'') \in \mathcal{R}_i[[e \cdot f]]} \\
\\
\frac{\sigma \in \text{sat}^\dagger(b) \quad (\sigma, \sigma') \in \mathcal{R}_i[[e]]}{(\sigma, \sigma') \in \mathcal{R}_i[[e +_b f]]} \qquad \frac{\sigma \in \text{sat}^\dagger(\bar{b}) \quad (\sigma, \sigma') \in \mathcal{R}_i[[f]]}{(\sigma, \sigma') \in \mathcal{R}_i[[e +_b f]]} \\
\\
\frac{\sigma \in \text{sat}^\dagger(b) \quad (\sigma, \sigma') \in \mathcal{R}_i[[e]] \quad (\sigma', \sigma'') \in \mathcal{R}_i[[e^{(b)}]]}{(\sigma, \sigma'') \in \mathcal{R}_i[[e^{(b)}]]} \qquad \frac{\sigma \in \text{sat}^\dagger(\bar{b})}{(\sigma, \sigma) \in \mathcal{R}_i[[e^{(b)}]]}
\end{array}$$

where $\text{sat}^\dagger : \text{BExp} \rightarrow 2^{\text{State}}$ denotes the lifting of $\text{sat} : T \rightarrow 2^{\text{State}}$ to Boolean expression over T in the usual way. \square

The rules defining $\mathcal{R}_i[[e]]$ are reminiscent of the big-step semantics of many imperative languages; these languages arise as instances of the model for various choices of i . The following result says that the language model from the previous section abstracts the various relational interpretations in a sound and complete way. It was first proved in [96] for Kleene Algebra with Tests (KAT).

Theorem 5.2.4. *The language model is sound and complete for the relational model:*

$$[[e]] = [[f]] \iff \forall i. \mathcal{R}_i[[e]] = \mathcal{R}_i[[f]]$$

It is worth noting that Theorem 5.2.4 also holds for refinement (*i.e.*, with \subseteq instead of $=$).

Example 5.2.5 (IMP). Consider a simple imperative programming language IMP with variable assignments and arithmetic and boolean expressions:

arithmetic expressions $a \in \mathcal{A} ::= x \in \text{Var} \mid n \in \mathbb{Z} \mid a_1 + a_2 \mid a_1 - a_2 \mid a_1 \times a_2$

boolean expressions $b \in \mathcal{B} ::= \mathbf{false} \mid \mathbf{true} \mid a_1 < a_2 \mid \mathbf{not} \ b \mid b_1 \ \mathbf{and} \ b_2 \mid b_1 \ \mathbf{or} \ b_2$

commands $c \in \mathcal{C} ::= \mathbf{skip} \mid x := a \mid c_1; c_2 \mid \mathbf{if} \ b \ \mathbf{then} \ c_1 \ \mathbf{else} \ c_2 \mid \mathbf{while} \ b \ \mathbf{do} \ c$

IMP can be modeled in GKAT using actions for assignments and primitive tests for comparisons,¹

$$\Sigma = \{x := a \mid x \in \text{Var}, a \in \mathcal{A}\} \quad T = \{a_1 < a_2 \mid a_1, a_2 \in \mathcal{A}\}$$

and interpreting GKAT expressions over the state space of variable assignments $\text{State} := \text{Var} \rightarrow \mathbb{Z}$:

$$\text{eval}(x := a) := \{(\sigma, \sigma[x := n]) \mid \sigma \in \text{State}, n = \mathcal{A}[[a]](\sigma)\}$$

$$\sigma[x := n] := \lambda y. \begin{cases} n & \text{if } x = y \\ \sigma(x) & \text{else} \end{cases}$$

$$\text{sat}(a_1 < a_2) := \{\sigma \in \text{State} \mid \mathcal{A}[[a_1]](\sigma) < \mathcal{A}[[a_2]](\sigma)\},$$

where $\mathcal{A}[[a]] : \text{State} \rightarrow \mathbb{Z}$ denotes arithmetic evaluation. Sequential composition, conditionals, and while loops in IMP are modeled by their GKAT counterparts; **skip** is modeled by 1. Thus, IMP equivalence refines GKAT equivalence (Theorem 5.2.4). For example, the program transformation

$$\begin{aligned} & \text{if } x < 0 \text{ then } (x := 0 - x; x := 2 \times x) \text{ else } (x := 2 \times x) \\ & \rightsquigarrow (\text{if } x < 0 \text{ then } x := 0 - x \text{ else skip}); x := 2 \times x \end{aligned}$$

is sound by the equivalence $pq +_b q \equiv (p +_b 1) \cdot q$. We study such equivalences further in Section 5.3. □

5.2.4 Probabilistic Model

In this subsection, we give a third interpretation of GKAT expressions in terms of sub-Markov kernels, a common model for probabilistic programming languages (PPLs). We show that the language model is sound and complete for this model as well.

¹Technically, we can only reserve a test for a *finite subset* of comparisons, as T is finite. However, for reasoning about pairwise equivalences of programs, which only contain a finite number of comparisons, this restriction is not essential.

We briefly review some basic primitives commonly used in the denotational semantics of PPLs. For a countable set² X , we let $\mathcal{D}(X)$ denote the set of subdistributions over X , *i.e.*, the set of probability assignments $f : X \rightarrow [0, 1]$ summing up to at most 1—*i.e.*, $\sum_{x \in X} f(x) \leq 1$. A common distribution is the *Dirac distribution* or *point mass* on $x \in X$, denoted $\delta_x \in \mathcal{D}(X)$; it is the map $y \mapsto [y = x]$ assigning probability 1 to x , and probability 0 to $y \neq x$. (The *Iverson bracket* $[\varphi]$ is defined to be 1 if the statement φ is true, and 0 otherwise.) Denotational models of PPLs typically interpret programs as *Markov kernels*, maps of type $X \rightarrow \mathcal{D}(X)$. Such kernels can be composed in sequence using Kleisli composition, since $\mathcal{D}(-)$ is a monad [52].

Definition 5.2.6 (Probabilistic Interpretation). Let $i = (\text{State}, \text{eval}, \text{sat})$ be a triple consisting of

- a countable set of *states* State ;
- for each action $p \in \Sigma$, a sub-Markov kernel $\text{eval}(p) : \text{State} \rightarrow \mathcal{D}(\text{State})$; and
- for each primitive test $t \in T$, a set of states $\text{sat}(t) \subseteq \text{State}$.

Then the *probabilistic interpretation* of an expression e with respect to i is the sub-Markov kernel $\mathcal{P}_i[[e]] : \text{State} \rightarrow \mathcal{D}(\text{State})$ defined as follows:

$$\begin{aligned} \mathcal{P}_i[[p]] &:= \text{eval}(p) & \mathcal{P}_i[[b]](\sigma) &:= [\sigma \in \text{sat}^\dagger(b)] \cdot \delta_\sigma \\ \mathcal{P}_i[[e \cdot f]](\sigma)(\sigma') &:= \sum_{\sigma''} \mathcal{P}_i[[e]](\sigma)(\sigma'') \cdot \mathcal{P}_i[[f]](\sigma'')(\sigma') \\ \mathcal{P}_i[[e +_b f]](\sigma) &:= [\sigma \in \text{sat}^\dagger(b)] \cdot \mathcal{P}_i[[e]](\sigma) + [\sigma \in \text{sat}^\dagger(\bar{b})] \cdot \mathcal{P}_i[[f]](\sigma) \\ \mathcal{P}_i[[e^{(b)}]](\sigma)(\sigma') &:= \lim_{n \rightarrow \infty} \mathcal{P}_i[[e +_b 1)^n \cdot \bar{b}]](\sigma)(\sigma') \quad \square \end{aligned}$$

See Lemma C.1.1 for a proof that the limit exists, and that $\mathcal{P}_i[[e]]$ is a sub-Markov kernel for all e .

²We restrict to countable states spaces (and thus, discrete distributions) for ease of presentation, but this section be generalized to uncountable state spaces and continuous distributions.

Theorem 5.2.7. *The language model is sound and complete for the probabilistic model:*

$$\llbracket e \rrbracket = \llbracket f \rrbracket \iff \forall i. \mathcal{P}_i \llbracket e \rrbracket = \mathcal{P}_i \llbracket f \rrbracket$$

Proof Sketch. By mutual implication.

\Rightarrow : For soundness, we define a map $\kappa_i: \text{GS} \rightarrow \text{State} \rightarrow \mathcal{D}(\text{State})$ that interprets guarded strings as sub-Markov kernels as follows:

$$\begin{aligned} \kappa_i(\alpha)(\sigma) &:= [\sigma \in \text{sat}^\dagger(\alpha)] \cdot \delta_\sigma \\ \kappa_i(\alpha p w)(\sigma)(\sigma') &:= [\sigma \in \text{sat}^\dagger(\alpha)] \cdot \sum_{\sigma''} \text{eval}(p)(\sigma)(\sigma'') \cdot \kappa_i(w)(\sigma'')(\sigma) \end{aligned}$$

We then lift κ_i to languages via pointwise summation, $\kappa_i(L) := \sum_{w \in L} \kappa_i(w)$, and establish that any probabilistic interpretation factors through the language model via $\kappa_i: \mathcal{P}_i \llbracket - \rrbracket = \kappa_i \circ \llbracket - \rrbracket$.

\Leftarrow : For completeness, we construct an interpretation $i := (\text{GS}, \text{eval}, \text{sat})$ over GS as follows,

$$\text{eval}(p)(w) := \text{Unif}(\{wp\alpha \mid \alpha \in \text{At}\}) \quad \text{sat}(t) := \{x\alpha \in \text{GS} \mid \alpha \leq t\}$$

and show that $\llbracket e \rrbracket$ is fully determined by $\mathcal{P}_i \llbracket e \rrbracket$:

$$\llbracket e \rrbracket = \{\alpha x \in \text{GS} \mid \mathcal{P}_i \llbracket e \rrbracket(\alpha)(\alpha x) \neq 0\}. \quad \square$$

As for Theorem 5.2.4, Theorem 5.2.7 can also be shown for refinement (*i.e.*, with \sqsubseteq and \leq instead of $=$).

Example 5.2.8 (Probabilistic IMP). We can extend IMP from Example 5.2.5 with a *probabilistic assignment* command $x \sim \mu$, where μ ranges over sub-distributions on \mathbb{Z} , as follows:

$$c ::= \dots \mid x \sim \mu \quad \Sigma = \dots \cup \{x \sim \mu \mid x \in \text{Var}, \mu \in \mathcal{D}(\mathbb{Z})\}$$

The interpretation $i = (\text{Var} \rightarrow \mathbb{Z}, \text{eval}, \text{sat})$ over the state space of variable assignments is as before, except we now restrict to a finite set of variables to guarantee that the state space is countable, and we interpret actions as sub-Markov kernels as follows:

$$\text{eval}(x := n)(\sigma) := \delta_{\sigma[x:=n]} \qquad \text{eval}(x \sim \mu)(\sigma) := \sum_{n \in \mathbb{Z}} \mu(n) \cdot \delta_{\sigma[x:=n]} \quad \square$$

5.3 Axiomatization

In most programming languages, the same behavior can be realized using different programs. For example, we expect the programs **if** b **then** e **else** f and **if** (**not** b) **then** f **else** e to encode the same behavior. Likewise, different expressions in GKAT can denote the same language of guarded strings. For instance, the previous example is reflected in GKAT by the fact that the language semantics of $e +_b f$ and $f +_{\bar{b}} e$ coincide. This raises the questions: what other equivalences hold between GKAT expressions? And, can all equivalences be captured by a finite number of equations? In this section, we give some initial answers to these questions, by proposing a set of axioms for GKAT and showing that they can be used to prove a large class of equivalences.

5.3.1 Some Simple Axioms

As an initial answer to the first question, we propose the following.

Definition 5.3.1. We define \equiv as the smallest congruence (w.r.t. all operators) on Exp that satisfies the axioms given in Figure 5.1 (for all $e, f, g \in \text{Exp}$ and $b, c, d \in \text{BExp}$) and subsumes Boolean equivalence in the sense that $b \equiv_{\text{BA}} c$ implies $b \equiv c$. \square

The guarded union axioms (U1-U5) can be understood intuitively by thinking in terms of if-then-else; for instance, we have the law $e +_b f \equiv f +_{\bar{b}} e$ discussed before, but also $eg +_b fg \equiv (e +_b f) \cdot g$, which says that if both branches of a guarded union end by

Guarded Union Axioms		Sequence Axioms (inherited from KA)	
U1.	$e +_b e \equiv e$	(idempotence)	S1. $(e \cdot f) \cdot g \equiv e \cdot (f \cdot g)$ (associativity)
U2.	$e +_b f \equiv f +_{\bar{b}} e$	(skew commut.)	S2. $0 \cdot e \equiv 0$ (absorbing left)
U3.	$(e +_b f) +_c g \equiv e +_{bc} (f +_c g)$	(skew assoc.)	S3. $e \cdot 0 \equiv 0$ (absorbing right)
U4.	$e +_b f \equiv be +_b f$	(guardedness)	S4. $1 \cdot e \equiv e$ (neutral left)
U5.	$eg +_b fg \equiv (e +_b f) \cdot g$	(right distrib.)	S5. $e \cdot 1 \equiv e$ (neutral right)
Guarded Loop Axioms			
W1.	$e^{(b)} \equiv ee^{(b)} +_b 1$	(unrolling)	W3. $\frac{g \equiv eg +_b f}{g \equiv e^{(b)} f}$ if $E(e) \equiv 0$ (fixpoint)
W2.	$(e +_c 1)^{(b)} \equiv (ce)^{(b)}$	(tightening)	

Figure 5.1: Axioms for GKAT-expressions.

executing g , then g can be “factored out.” Equivalences for sequential composition are also intuitive; for instance, $e \cdot 0 \equiv 0$ encodes that if a program will fail eventually by reaching the statement 0 , then the whole program fails.

The axioms for loops (W1–3) are more subtle. The axiom $e^{(b)} \equiv ee^{(b)} +_b 1$ (W1) says that we can think of a guarded loop as equivalent to its unrolling—*i.e.*, the program **while b do e** has the same behavior as the program **if b then $(e; \text{while } b \text{ do } e)$ else skip**. The axiom $(e +_c 1)^{(c)} \equiv (ce)^{(b)}$ (W2) states that iterations of a loop that do not have any effect (*i.e.*, an execution equivalent to **skip**) can be omitted; we refer to this transformation as *loop tightening*.

To explain the fixpoint axiom (W3), disregard the side-condition for a moment. In a sense, this rule states that if g tests (using b) whether to execute e and loop again or execute f (*i.e.*, if $g \equiv eg +_b f$) then g is a b -guarded loop followed by f (*i.e.*, $g \equiv e^{(b)} f$). However, such a rule is not sound in general. For instance, suppose $e, f, g, b = 1$; in that case, $1 \equiv 1 \cdot 1 +_1 1$ can be proved using the other axioms, but applying the rule would allow us to conclude that $1 \equiv 1^{(1)} \cdot 1$, even though $\llbracket 1 \rrbracket = \text{At}$ and $\llbracket 1^{(1)} \cdot 1 \rrbracket = \emptyset$! The problem here is that, while g is tail-recursive as required by the premise, this self-similarity is trivial because e does not represent a productive program. We thus need to restrict the application of the inference rule to cases where the loop body is *strictly productive*—*i.e.*, where e is guaranteed to execute *some* action. To this end, we define the function E as

follows.

Definition 5.3.2. The function $E : \text{Exp} \rightarrow \text{BExp}$ is defined inductively as follows:

$$\begin{aligned} E(b) &:= b & E(e +_b f) &:= b \cdot E(e) + \bar{b} \cdot E(f) & E(e^{(b)}) &:= \bar{b} \\ E(p) &:= 0 & E(e \cdot f) &:= E(e) \cdot E(f) & & \square \end{aligned}$$

Intuitively, $E(e)$ is the weakest test that guarantees that e terminates successfully, but does not perform any action. For instance, $E(p)$ is 0—the program p is guaranteed to perform the action p . Using E , we can now restrict the application of the fixpoint rule to the cases where $E(e) \equiv 0$, i.e., where e performs an action under any circumstance.

Theorem 5.3.3 (Soundness). *The GKAT axioms are sound for the language model:*

$$e \equiv f \quad \Longrightarrow \quad \llbracket e \rrbracket = \llbracket f \rrbracket.$$

Proof Sketch. By induction on the length of derivation of the congruence \equiv . We provide the full proof in the appendix and show just the proof for the fixpoint rule. Here, we should argue that if $E(e) \equiv 0$ and $\llbracket g \rrbracket = \llbracket eg +_b f \rrbracket$, then also $\llbracket g \rrbracket = \llbracket e^{(b)} f \rrbracket$. We note that, using soundness of (W1) and (U5), we can derive that $\llbracket e^{(b)} f \rrbracket = \llbracket (ee^{(b)} +_b 1)f \rrbracket = \llbracket ee^{(b)} f +_b f \rrbracket$.

We reason by induction on the length of guarded strings. In the base case, we know that $\alpha \in \llbracket g \rrbracket$ if and only if $\alpha \in \llbracket eg +_b f \rrbracket$; since $E(e) \equiv 0$, the latter holds precisely when $\alpha \in \llbracket f \rrbracket$ and $\alpha \leq \bar{b}$, which is equivalent to $\alpha \in \llbracket e^{(b)} f \rrbracket$. For the inductive step, suppose

Guarded Union Facts

- U3'. $e +_b (f +_c g) \equiv (e +_b f) +_{b+c} g$ (skew assoc.)
 U4'. $e +_b f \equiv e +_b \bar{b}f$ (guardedness)
 U5'. $b \cdot (e +_c f) \equiv be +_c bf$ (left distrib.)
 U6. $e +_b 0 \equiv be$ (neutral right)
 U7. $e +_0 f \equiv f$ (trivial right)
 U8. $b \cdot (e +_b f) \equiv be$ (branch sel.)

Guarded Iteration Facts

- W4. $e^{(b)} \equiv e^{(b)} \cdot \bar{b}$ (guardedness)
 W4'. $e^{(b)} \equiv (be)^{(b)}$ (guardedness)
 W5. $e^{(0)} \equiv 1$ (neutrality)
 W6. $e^{(1)} \equiv 0$ (absorption)
 W6'. $b^{(c)} \equiv \bar{c}$ (absorption)
 W7. $e^{(c)} \equiv e^{(bc)} \cdot e^{(c)}$ (fusion)

Figure 5.2: Derivable GKAT facts

the claim holds for y ; then

$$\begin{aligned}
 & \alpha py \in \llbracket g \rrbracket \\
 \iff & \alpha py \in \llbracket eg +_b f \rrbracket \\
 \iff & \alpha py \in \llbracket eg \rrbracket \wedge \alpha \leq b \quad \mathbf{or} \quad \alpha py \in \llbracket f \rrbracket \wedge \alpha \leq \bar{b} \\
 \iff & \exists y. y_1 y_2 \wedge \alpha p y_1 \in \llbracket e \rrbracket \wedge y_2 \in \llbracket g \rrbracket \wedge \alpha \leq b \quad \mathbf{or} \quad \alpha py \in \llbracket f \rrbracket \wedge \alpha \leq \bar{b} \quad (E(e) = 0) \\
 \iff & \exists y. y_1 y_2 \wedge \alpha p y_1 \in \llbracket e \rrbracket \wedge y_2 \in \llbracket e^{(b)} f \rrbracket \wedge \alpha \leq b \quad \mathbf{or} \quad \alpha py \in \llbracket f \rrbracket \wedge \alpha \leq \bar{b} \quad (\text{IH}) \\
 \iff & \alpha py \in \llbracket ee^{(b)} f \rrbracket \wedge \alpha \leq b \quad \mathbf{or} \quad \alpha py \in \llbracket f \rrbracket \wedge \alpha \leq \bar{b} \\
 \iff & \alpha py \in \llbracket ee^{(b)} f +_b f \rrbracket = \llbracket e^{(b)} f \rrbracket \quad \square
 \end{aligned}$$

5.3.2 A Fundamental Theorem

The side condition on (W3) is inconvenient when proving facts about loops. However, it turns out that we can transform any loop into an equivalent, *productive* loop—i.e., one with a loop body e such that $E(e) \equiv 0$. To this end, we need a way of decomposing a GKAT expression into a guarded sum of an expression that describes termination, and another (strictly productive) expression that describes the next steps that the program may undertake. As a matter of fact, we already have a handle on the former term: $E(e)$ is a Boolean term that captures the atoms for which e may halt immediately. It therefore remains to describe the next steps of a program.

Definition 5.3.4 (Derivatives). Let $\alpha \in \text{At}$. We define $D_\alpha: \text{Exp} \rightarrow 2 + \Sigma \times \text{Exp}$ inductively as follows:

$$D_\alpha(b) = \begin{cases} 1 & \alpha \leq b \\ 0 & \alpha \not\leq b \end{cases} \quad D_\alpha(p) = (p, 1) \quad D_\alpha(e +_b f) = \begin{cases} D_\alpha(e) & \alpha \leq b \\ D_\alpha(f) & \alpha \leq \bar{b} \end{cases}$$

$$D_\alpha(e \cdot f) = \begin{cases} (p, e' \cdot f) & D_\alpha(e) = (p, e') \\ 0 & D_\alpha(e) = 0 \\ D_\alpha(f) & D_\alpha(e) = 1 \end{cases}$$

$$D_\alpha(e^{(b)}) = \begin{cases} (p, e' \cdot e^{(b)}) & \alpha \leq b \wedge D_\alpha(e) = (p, e') \\ 0 & \alpha \leq b \wedge D_\alpha(e) \in 2 \\ 1 & \alpha \leq \bar{b} \end{cases}$$

□

We will use a general type of guarded union to sum over an atom-indexed set of expressions.

Definition 5.3.5. Let $\Phi \subseteq \text{At}$, and let $\{e_\alpha\}_{\alpha \in \Phi}$ be a set of expressions indexed by Φ . We write

$$\bigoplus_{\alpha \in \Phi} e_\alpha = \begin{cases} e_\beta +_\beta \left(\bigoplus_{\alpha \in \Phi \setminus \{\beta\}} e_\alpha \right) & \beta \in \Phi \\ 0 & \Phi = \emptyset \end{cases}$$

Like other operators on indexed sets, we may abuse notation and replace Φ by a predicate over some atom α , with e_α a function of α ; for instance, we could write $\bigoplus_{\alpha \leq 1} \alpha \equiv 1$. □

Remark 5.3.6. The definition above is ambiguous in the choice of β . However, that does not change the meaning of the expression, as far as \equiv is concerned. For the details, see Appendix C.2. □

We are now ready to state the desired decomposition of terms; we call this the *fundamental theorem* of GKAT, after [149, 162]. The proof is included in the appendix.

Theorem 5.3.7 (Fundamental Theorem). *For all GKAT programs e , the following equality holds:*

$$e \equiv 1 +_{E(e)} D(e), \quad \text{where } D(e) := \bigoplus_{\alpha: D_\alpha(e)=(p_\alpha, e_\alpha)} p_\alpha \cdot e_\alpha. \quad (5.1)$$

The following observations about D and E are also useful; the proof is deferred to the appendix.

Lemma 5.3.8. *Let e be an expression. Then its components $E(e)$ and $D(e)$ satisfy the following identities:*

$$E(D(e)) \equiv 0 \qquad \overline{E(e)} \cdot D(e) \equiv D(e) \qquad \overline{E(e)} \cdot e \equiv D(e)$$

Using the fundamental theorem and the above, we can now show how to syntactically transform any loop into an equivalent loop whose body e is strictly productive.

Lemma 5.3.9 (Productive Loop). *Let $e \in \text{Exp}$ and $b \in \text{BExp}$. We have $e^{(b)} \equiv D(e)^{(b)}$.*

Proof. Using Lemma 5.3.8, we derive as follows:

$$e^{(b)} \stackrel{\text{FT}}{\equiv} (1 +_{E(e)} D(e))^{(b)} \stackrel{\text{U2}}{\equiv} (D(e) +_{\overline{E(e)}} 1)^{(b)} \stackrel{\text{W2}}{\equiv} (\overline{E(e)} D(e))^{(b)} \equiv D(e)^{(b)} \quad \square$$

5.3.3 Derivable Facts

The GKAT axioms can be used to derive other natural equivalences of programs, such as the ones in Figure 5.2. For instance, $e^{(b)} \equiv e^{(b)} \bar{b}$, labelled (W4), says that b must be false when $e^{(b)}$ ends.

Lemma 5.3.10. *The facts in Figure 5.2 are derivable from the axioms.*

Proof Sketch. We show (U6) and (W7); the remaining proofs are included in the ap-

pendix.

$$\begin{aligned}
e +_b 0 &\equiv be +_b 0 && \text{(U4. } e +_b f \equiv be +_b f) \\
&\equiv 0 +_{\bar{b}} be && \text{(U2. } e +_b f \equiv f +_{\bar{b}} e) \\
&\equiv \bar{b}be +_{\bar{b}} be && \text{(Boolean algebra and S2. } 0 \equiv 0e) \\
&\equiv be +_{\bar{b}} be && \text{(U4. } e +_b f \equiv be +_b f) \\
&\equiv be && \text{(U1. } e +_b e \equiv e)
\end{aligned}$$

To prove (W7), we use the productive loop lemma and the fixpoint axiom (W3).

$$\begin{aligned}
e^{(c)} &\equiv e^{(c)} +_{bc} e^{(c)} && \text{(U1. } e +_b e \equiv e) \\
&\equiv (D(e))^{(c)} +_{bc} e^{(c)} && \text{(Productive loop lemma)} \\
&\equiv D(e)D(e)^{(c)} +_{bc} e^{(c)} && \text{(W2, U4 and Boolean algebra)} \\
&\equiv D(e)e^{(c)} +_{bc} e^{(c)} && \text{(Productive loop lemma)} \\
&\equiv D(e)^{(bc)}e^{(c)} && \text{(W3)} \\
&\equiv e^{(bc)}e^{(c)} && \text{(Productive loop lemma)}
\end{aligned}$$

This completes the proof. □

We conclude our presentation of derivable facts by showing one more interesting fact. Unlike the derived facts above, this one is an implication: if the test c is invariant for the program e given that a test b succeeds, then c is preserved by a b -loop on e .

Lemma 5.3.11 (Invariance). *Let $e \in \text{Exp}$ and $b, c \in \text{BExp}$. If $cbe \equiv cbec$, then $ce^{(b)} \equiv (ce)^{(b)}c$.*

Proof. We first derive a useful equivalence, as follows:

$$\begin{aligned}
cb \cdot D(e) &\equiv cb \cdot \overline{E(e)} \cdot e && \text{(Lemma 5.3.8)} \\
&\equiv \overline{E(e)} \cdot cbe && \text{(Boolean algebra)} \\
&\equiv \overline{E(e)} \cdot cbec && \text{(premise)} \\
&\equiv cb \cdot \overline{E(e)} \cdot ec && \text{(Boolean algebra)} \\
&\equiv cb \cdot D(e) \cdot c && \text{(Lemma 5.3.8)}
\end{aligned}$$

Next, we show the main claim by deriving

$$\begin{aligned}
ce^{(b)} &\equiv c \cdot D(e)^{(b)} && \text{(Productive loop lemma)} \\
&\equiv c \cdot (D(e) \cdot D(e)^{(b)} +_b 1) && \text{(W1)} \\
&\equiv c \cdot (D(e) \cdot e^{(b)} +_b 1) && \text{(Productive loop lemma)} \\
&\equiv c \cdot (b \cdot D(e) \cdot e^{(b)} +_b 1) && \text{(U2)} \\
&\equiv cb \cdot D(e) \cdot e^{(b)} +_b c && \text{(U5')} \\
&\equiv cb \cdot D(e) \cdot ce^{(b)} +_b c && \text{(above derivation)} \\
&\equiv c \cdot D(e) \cdot ce^{(b)} +_b c && \text{(U2)} \\
&\equiv (c \cdot D(e))^{(b)} c && \text{(W3)} \\
&\equiv D(ce)^{(b)} c && \text{(Def. } D, \text{ Boolean algebra)} \\
&\equiv (ce)^{(b)} c && \text{(Productive loop lemma)}
\end{aligned}$$

This completes the proof. □

5.3.4 A Limited Form of Completeness

Above, we considered a number of axioms that were proven sound with respect to the language model. Ultimately, we would like to show that in fact these are sufficient to

prove all equivalences between programs—*i.e.*, whenever $\llbracket e \rrbracket = \llbracket f \rrbracket$, it also holds that $e \equiv f$.

This general completeness result in Section 5.6 requires some additional machinery. However, we can already prove a special case of completeness related to Hoare triples. Suppose e is a GKAT program, and b and c are Boolean expressions encoding pre- and postconditions. If we would like to state that, assuming b holds initially, every terminating run of e finishes in a state satisfying c , then this can be encoded in the language model by saying that $\llbracket be \rrbracket = \llbracket bec \rrbracket$, *i.e.*, every finite run of be is a run where c holds upon termination. The following states that all valid Hoare triples of this kind can be established axiomatically:

Theorem 5.3.12 (Hoare completeness). *Let $e \in \text{Exp}$ and $b, c \in \text{BExp}$. If $\llbracket bec \rrbracket = \llbracket be \rrbracket$, then $bec \equiv be$.*

Proof. By induction on e . In the base, there are two cases to consider.

- If $e = d$ for some Boolean d , then the claim follows by completeness of the Boolean algebra axioms, which \equiv subsumes by definition.
- If $e = a \in \Sigma$, then $\llbracket bec \rrbracket = \llbracket be \rrbracket$ implies $\llbracket c \rrbracket = \llbracket 1 \rrbracket$, hence $c \equiv 1$ by completeness of Boolean algebra; the claim then follows.

For the inductive step, there are three cases:

- If $e = e_0 +_d e_1$, then $\llbracket bec \rrbracket = \llbracket be \rrbracket$ implies that $\llbracket dbe_0c \rrbracket = \llbracket dbe_0 \rrbracket$ and $\llbracket \bar{d}be_1c \rrbracket = \llbracket \bar{d}be_1 \rrbracket$. By induction, we then know that $dbe_0c \equiv dbe_0$ and $\bar{d}be_1c \equiv \bar{d}be_1$. We can then

derive as follows:

$$b(e_0 +_d e_1)c \equiv be_0c +_d be_1c \quad (\text{U5}')$$

$$\equiv dbe_0c +_d \bar{d}be_1c \quad (\text{U4, U4}')$$

$$\equiv dbe_0c +_d \bar{d}be_1 \quad (\bar{d}be_1c \equiv \bar{d}be_1)$$

$$\equiv dbe_0 +_d \bar{d}be_1 \quad (dbe_0c \equiv dbe_0)$$

$$\equiv be_0 +_d be_1 \quad (\text{U4, U4}')$$

$$\equiv b \cdot (e_0 +_d e_1) \quad (\text{U5}')$$

- If $e = e_0 \cdot e_1$, then let $d = \sum\{\alpha \in \text{At} : \llbracket be_0\alpha \rrbracket \neq \emptyset\}$. We then know that $\llbracket be_0d \rrbracket = \llbracket be_0 \rrbracket$, and hence $be_0d \equiv be_0$ by induction. We furthermore claim that $\llbracket de_1c \rrbracket = \llbracket de_1 \rrbracket$. To see this, note that if $\alpha w \beta \in \llbracket de_1 \rrbracket$, then $\alpha \leq d$, and hence there exists an $x\alpha \in \llbracket be_0\alpha \rrbracket \subseteq \llbracket be_0d \rrbracket = \llbracket be_0 \rrbracket$. Thus, we know that $x\alpha w \beta \in \llbracket be_0e_1 \rrbracket = \llbracket be_0e_1c \rrbracket$, meaning that $\beta \leq c$; hence, we know that $\alpha w \beta \in \llbracket de_1c \rrbracket$. By induction, $de_1c \equiv de_1$. We then derive:

$$be_0e_1c \equiv be_0de_1c \quad (be_0 \equiv be_0d)$$

$$\equiv be_0de_1 \quad (de_1 \equiv de_1c)$$

$$\equiv be_0e_1 \quad (be_0 \equiv be_0d)$$

- If $e = e_0^{(d)}$, first note that if $b \equiv 0$, then the claim follows trivially. Otherwise, let

$$h = \sum\{\alpha \in \text{At} : \exists n. \llbracket b \rrbracket \diamond \llbracket de_0 \rrbracket^n \diamond \llbracket \alpha \rrbracket \neq \emptyset\}$$

We make the following observations.

- (i) Since $b \neq 0$, we have that $\llbracket b \rrbracket \diamond \llbracket de_0 \rrbracket^0 \diamond \llbracket b \rrbracket = \llbracket b \rrbracket \neq \emptyset$, and thus $b \leq h$.
- (ii) If $\alpha \leq h\bar{d}$, then in particular $\gamma w \alpha \in \llbracket b \rrbracket \diamond \llbracket de_0 \rrbracket^n \diamond \llbracket \alpha \rrbracket$ for some n and γw . Since $\alpha \leq \bar{d}$, it follows that $\gamma w \alpha \in \llbracket be_0^{(d)} \rrbracket = \llbracket be_0^{(d)}c \rrbracket$, and thus $\alpha \leq c$. Consequently, $h\bar{d} \leq c$.

- (iii) If $\alpha w \beta \in \llbracket dhe_0 \rrbracket$, then $\alpha \leq h$ and hence there exists an n such that $\gamma x \alpha \in \llbracket b \rrbracket \diamond \llbracket de_0 \rrbracket^n \diamond \llbracket \beta \rrbracket$. But then $\gamma x \alpha w \beta \in \llbracket b \rrbracket \diamond \llbracket de_0 \rrbracket^{n+1} \diamond \llbracket \beta \rrbracket$, and therefore $\beta \leq h$. We can conclude that $\llbracket dhe_0 \rrbracket = \llbracket dhe_0 h \rrbracket$; by induction, it follows that $dhe_0 h \equiv dhe_0$.

Using these observations and the invariance lemma (Lemma 5.3.11), we derive

$$\begin{aligned}
be_0^{(d)}c &\equiv bhe_0^{(d)}c && \text{(By (i))} \\
&\equiv b \cdot (he_0)^{(d)}hc && \text{(Invariance and (iii))} \\
&\equiv b \cdot (he_0)^{(d)}\bar{d}hc && \text{(W4)} \\
&\equiv b \cdot (he_0)^{(d)}\bar{d}h && \text{(By (ii))} \\
&\equiv b \cdot (he_0)^{(d)}h && \text{(W4)} \\
&\equiv bhe_0^{(d)} && \text{(Invariance and (iii))} \\
&\equiv be_0^{(d)} && \text{(By (i))}
\end{aligned}$$

This completes the proof. \square

As a special case, the fact that a program has no traces at all can be shown axiomatically.

Corollary 5.3.13 (Partial Completeness). *If $\llbracket e \rrbracket = \emptyset$, then $e \equiv 0$.*

Proof. We have that $\llbracket 1 \cdot e \rrbracket = \llbracket e \rrbracket = \emptyset = \llbracket 1 \cdot e \cdot 0 \rrbracket$, and thus $e \equiv 1 \cdot e \equiv 1 \cdot e \cdot 0 \equiv 0$ by Theorem 5.3.12. \square

5.4 Automaton Model and Kleene Theorem

In this section, we present an automaton model that accepts traces (*i.e.*, guarded strings) of GKAT programs. We then present language-preserving constructions from GKAT expressions to automata, and conversely, from automata to expressions. Our automaton model

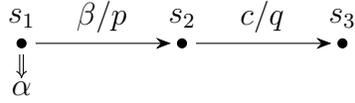


Figure 5.3: Graphical depiction of a G -coalgebra $\langle X, \delta^X \rangle$. States are represented by dots, labeled with the name of that state whenever relevant. In this example, $\delta^X(s_1)(\alpha) = 1$, and $\delta^X(s_1)(\beta) = (p, s_2)$. When $\gamma \in \text{At}$ such that $\delta^X(s)(\gamma) = 0$, we draw no edge at all. We may abbreviate drawings by combining transitions with the same target into a Boolean expression; for instance, when $c = \alpha + \beta$, we have $\delta^X(s_2)(\alpha) = \delta^X(s_2)(\beta) = (q, s_3)$.

is rich enough to express programs that go beyond GKAT; in particular, it can encode traces of programs with **goto** statements that have no equivalent GKAT program [98]. In order to obtain a Kleene Theorem for GKAT, that is, a correspondence between automata and GKAT programs, we identify conditions ensuring that the language accepted by an automaton corresponds to a valid GKAT program.

5.4.1 Automata and Languages

Let G be the functor $GX = (2 + \Sigma \times X)^{\text{At}}$, where $2 = \{0, 1\}$ is the two-element set. A G -coalgebra is a pair $\mathcal{X} = \langle X, \delta^X \rangle$ with *state space* X and *transition map* $\delta^X: X \rightarrow GX$. The outcomes 1 and 0 model immediate acceptance and rejection, respectively. From each state $s \in X$, given an input $\alpha \in \text{At}$, the coalgebra performs exactly one of three possible actions: it either produces an output $p \in \Sigma$ and moves to a new state t , halts and accepts, or halts and rejects; that is, either $\delta^X(s)(\alpha) = (p, t)$, or $\delta^X(s)(\alpha) = 1$, or $\delta^X(s)(\alpha) = 0$.

A G -automaton is a G -coalgebra with a designated start state ι , commonly denoted as a triple $\mathcal{X} = \langle X, \delta^X, \iota \rangle$. We can represent G -coalgebras graphically as in Figure 5.3.

A G -coalgebra $\mathcal{X} = \langle X, \delta^X \rangle$ can be viewed both as an acceptor of finite guarded strings $\text{GS} = \text{At} \cdot (\Sigma \cdot \text{At})^*$, or as an acceptor of finite *and* infinite guarded strings $\text{GS} \cup \omega\text{-GS}$, where $\omega\text{-GS} := (\text{At} \cdot \Sigma)^\omega$. Acceptance for a state s is captured by the following

equivalences:

$$\begin{aligned} \text{accept}(s, \alpha) &\iff \delta^{\mathcal{X}}(s)(\alpha) = 1 \\ \text{accept}(s, \alpha px) &\iff \exists t. \delta^{\mathcal{X}}(s)(\alpha) = (p, t) \wedge \text{accept}(t, x) \end{aligned} \tag{5.2}$$

The language of finite guarded strings $\ell^{\mathcal{X}}(s) \subseteq GS$ accepted from state $s \in X$ is the *least fixpoint* solution of the above system; in other words, we interpret (5.2) inductively. The language of finite and infinite guarded strings $L^{\mathcal{X}}(s) \subseteq GS \cup \omega\text{-GS}$ accepted from state s is the *greatest fixpoint* solution of the above system; in other words, we interpret (5.2) coinductively.³ The two languages are related by the equation $\ell^{\mathcal{X}}(s) = L^{\mathcal{X}}(s) \cap GS$. Our focus will mostly be on the finite-string semantics, $\ell^{\mathcal{X}}(-): X \rightarrow 2^{GS}$, since GKAT expressions denote finite-string languages, $\llbracket - \rrbracket: \text{Exp} \rightarrow 2^{GS}$.

The language accepted by a G -automaton $\mathcal{X} = \langle X, \delta^{\mathcal{X}}, \iota \rangle$ is the language accepted by its initial state ι . Just like the language model for GKAT programs, the language semantics of a G -automaton satisfies the determinacy property (see Definition 5.2.2). In fact, every language that satisfies the determinacy property can be recognized by a G -automaton, possibly with infinitely many states. (We will prove this formally in Theorem 5.5.8.)

5.4.2 Expressions to Automata: a Thompson Construction

We translate expressions to G -coalgebras using a construction reminiscent of Thompson's construction for regular expressions [169], where automata are formed by induction on the structure of the expressions and combined to reflect the various GKAT operations.

We first set some notation. A *pseudostate* is an element $h \in GX$. We let $\mathbf{1} \in GX$ denote the pseudostate $\mathbf{1}(\alpha) = 1$, *i.e.*, the constant function returning 1. Let $\mathcal{X} = \langle X, \delta \rangle$

³The set \mathcal{F} of maps $F: X \rightarrow 2^{GS \cup \omega\text{-GS}}$ ordered pointwise by subset inclusion forms a complete lattice. The monotone map

$$\tau: \mathcal{F} \rightarrow \mathcal{F}, \tau(F) = \lambda s \in X. \{\alpha \in \text{At} \mid \delta^{\mathcal{X}}(s)(\alpha) = 1\} \cup \{\alpha px \mid \exists t. \delta^{\mathcal{X}}(s)(\alpha) = (p, t) \wedge x \in F(t)\}$$

arising from (5.2) has least and greatest fixpoints, $\ell^{\mathcal{X}}$ and $L^{\mathcal{X}}$, by the Knaster-Tarski theorem.

e	X_e	$\delta_e \in X_e \rightarrow GX_e$	$\iota_e(\alpha) \in 2 + \Sigma \times X_e$
b	\emptyset	\emptyset	$[\alpha \leq b]$
p	$\{*\}$	$* \mapsto \mathbf{1}$	$(p, *)$
$f +_b g$	$X_f + X_g$	$\delta_f + \delta_g$	$\begin{cases} \iota_f(\alpha) & \alpha \leq b \\ \iota_g(\alpha) & \alpha \leq \bar{b} \end{cases}$
$f \cdot g$	$X_f + X_g$	$(\delta_f + \delta_g)[X_f, \iota_g]$	$\begin{cases} \iota_f(\alpha) & \iota_f(\alpha) \neq 1 \\ \iota_g(\alpha) & \iota_f(\alpha) = 1 \end{cases}$
$f^{(b)}$	X_f	$\delta_f[X_f, \iota_e]$	$\begin{cases} 1 & \alpha \leq \bar{b} \\ 0 & \alpha \leq b, \iota_f(\alpha) = 1 \\ \iota_f(\alpha) & \alpha \leq b, \iota_f(\alpha) \neq 1 \end{cases}$

Figure 5.4: Construction of the Thompson coalgebra $\mathcal{X}_e = \langle X_e, \delta_e \rangle$ with initial pseudostate ι_e .

be a G -coalgebra. The *uniform continuation* of $Y \subseteq X$ by $h \in GX$ (in \mathcal{X}) is the coalgebra $\mathcal{X}[Y, h] := \langle X, \delta[Y, h] \rangle$, where

$$\delta[Y, h](x)(\alpha) := \begin{cases} h(\alpha) & \text{if } x \in Y, \delta(x)(\alpha) = 1 \\ \delta(x)(\alpha) & \text{otherwise.} \end{cases}$$

Intuitively, uniform continuation replaces termination of states in a region Y of \mathcal{X} by a transition described by $h \in GX$; this construction will be useful for modeling operations that perform some kind of sequencing. Figure 5.5 schematically describes the uniform continuation operation, illustrating different changes to the automaton that can occur as a result; observe that since h may have transitions into Y , uniform continuation can introduce loops.

We will also need coproducts to combine coalgebras. Intuitively, the coproduct of two coalgebras is just the juxtaposition of both coalgebras. Formally, for $\mathcal{X} = \langle X, \delta_1 \rangle$ and $\mathcal{Y} = \langle Y, \delta_2 \rangle$, we write the coproduct as $\mathcal{X} + \mathcal{Y} = \langle X + Y, \delta_1 + \delta_2 \rangle$, where $X + Y$ is the disjoint union of X and Y , and $\delta_1 + \delta_2: X + Y \rightarrow G(X + Y)$ is the map that applies δ_1 to states in X and δ_2 to states in Y .

Figure 5.4 presents our translation from expressions e to coalgebras \mathcal{X}_e using co-

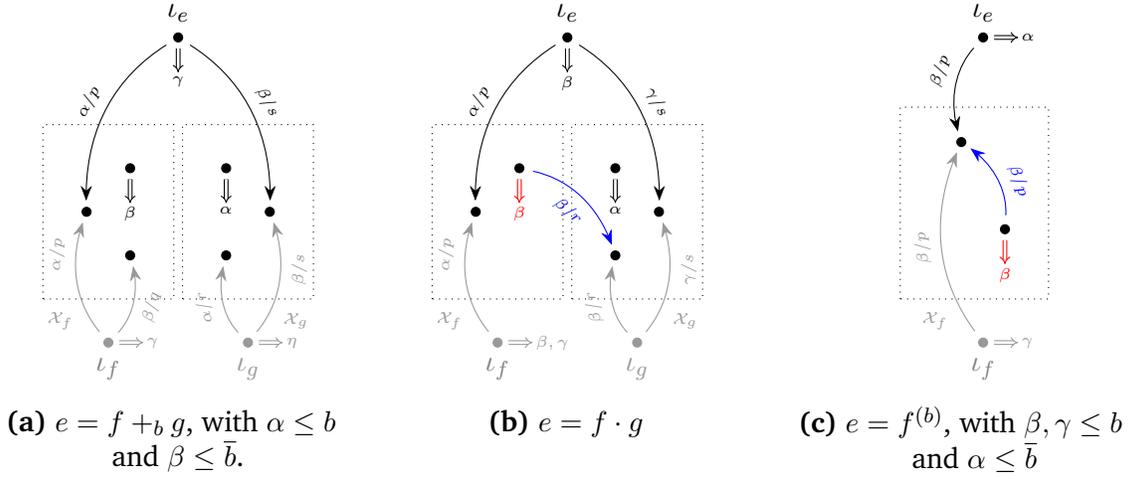


Figure 5.6: Schematic depiction of the Thompson construction for guarded union, sequencing and guarded loop operators. The initial pseudostates of the automata for f and g are depicted in gray. Transitions in red are present in the automata for f and g , but overridden by a uniform extension with the transitions in blue.

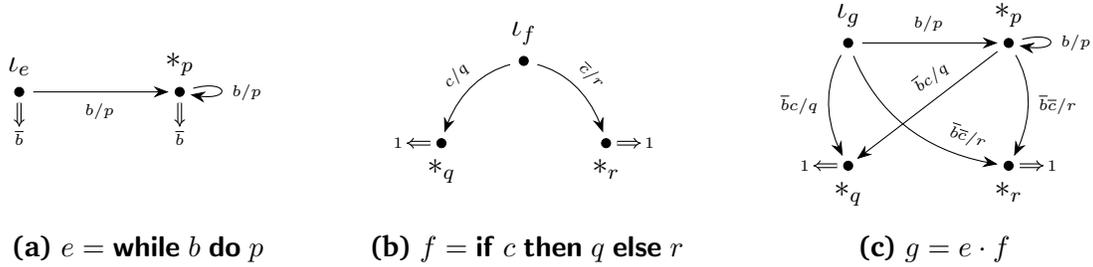


Figure 5.7: Concrete construction of an automaton using the Thompson construction. First, we construct an automaton for e , then an automaton for f , and finally we combine these into an automaton for g .

size of the expression, and $\#_{\Sigma}(e)$ denotes the number of occurrences of actions in e .

5.4.3 Automata to Expressions: Solving Linear Systems

The previous construction shows that every GKAT expression can be translated to an equivalent G-automaton. In this section we consider the reverse direction, from G-automata to GKAT expressions. The main idea is to interpret the coalgebra structure as a system of equations, with one variable and equation per state, and show that there

are GKAT expressions solving the system, modulo equivalence; this idea goes back to Conway [27] and Backhouse [11]. Not all systems arising from G-coalgebras have a solution, and so not all G-coalgebras can be captured by GKAT expressions. However, we identify a subclass of G-coalgebras that can be represented as GKAT terms. Then, by showing that this class contains the coalgebras produced by our expressions-to-automata translation, we obtain an equivalence between GKAT expressions and coalgebras in this class.

We start by defining when a map assigning expressions to coalgebra states is a solution.

Definition 5.4.3 (Solution). Let $\mathcal{X} = \langle X, \delta^{\mathcal{X}} \rangle$ be a G-coalgebra. We say that $s: X \rightarrow \text{Exp}$ is a *solution* to \mathcal{X} if for all $x \in X$ it holds that

$$s(x) \equiv \bigoplus_{\alpha \leq 1} [\delta^{\mathcal{X}}(x)(\alpha)]_s \quad \text{where} \quad [0]_s := 0 \quad [1]_s := 1 \quad [\langle p, x \rangle]_s := p \cdot s(x) \quad \square$$

Example 5.4.4. Consider the examples of Thompson automata in Figure 5.7.

- (a) Solving the first automaton requires, by Definition 5.4.3, finding an expression $s_e(*_p)$ such that $s_e(*_p) \equiv p \cdot s_e(*_p) +_b 1$. By (W1), we know that $s_e(*_p) = p^{(b)}$ is valid; in fact, (W3) tells us that this choice of x is the *only* valid solution up to GKAT-equivalence. If we include ι_e as a state, we can choose $s_e(\iota_e) = p^{(b)}$ as well.
- (b) The second automaton has an easy solution: both $*_q$ and $*_r$ are solved by setting $s_f(*_q) = s_f(*_r) = 1$. If we include ι_f as a state, we can choose $s_f(\iota_f) = q \cdot s_f(*_q) +_b r \cdot s_f(*_r) \equiv q +_b r$.
- (c) The third automaton was constructed from the first two; similarly, we can construct its solution from the solutions to the first two. We set $s_g(*_p) = s_e(*_p) \cdot s_f(\iota_f)$, and $s_g(*_q) = s_f(*_q)$, and $s_g(*_r) = s_f(*_r)$. If we include ι_g as a state, we can choose $s_g(\iota_g) = s_e(\iota_e) \cdot s_f(\iota_f)$. □

Solutions are language-preserving maps from states to expressions in the following sense:

Proposition 5.4.5. *If s solves \mathcal{X} and x is a state, then $\llbracket s(x) \rrbracket = \ell^{\mathcal{X}}(x)$.*

Proof Sketch. We show that $w \in \llbracket s(x) \rrbracket \Leftrightarrow w \in \ell^{\mathcal{X}}(x)$ by induction on the length of $w \in \text{GS}$. □

We would like to build solutions for G -coalgebras, but Kozen and Tseng [98] showed that this is not possible in general: there is a 3-state G -coalgebra that does not correspond to any **while** program, but instead can only be modeled by a program with multi-level breaks. In order to obtain an exact correspondence to GKAT programs, we first identify a sufficient condition for G -coalgebras to permit solutions, and then show that the Thompson coalgebra defined previously meets this condition.

Definition 5.4.6 (Simple Coalgebra). Let $\mathcal{X} = \langle X, \delta^{\mathcal{X}} \rangle$ and $\mathcal{Y} = \langle Y, \delta^{\mathcal{Y}} \rangle$ range over G -coalgebras. The collection of *simple* coalgebras is inductively defined as follows:

(S1) If \mathcal{X} has no transitions, *i.e.*, if $\delta^{\mathcal{X}} \in X \rightarrow 2^{\text{At}}$, then \mathcal{X} is simple.

(S2) If \mathcal{X} and \mathcal{Y} are simple, and $h \in G(X + Y)$ is a pseudostate, then $(\mathcal{X} + \mathcal{Y})[X, h]$ is simple. □

We are now ready to construct solutions to simple coalgebras.

Theorem 5.4.7 (Existence of Solutions). *Any simple coalgebra admits a solution.*

Proof Sketch. Assume \mathcal{X} is simple. We proceed by rule induction on the simplicity derivation.

(S1) Suppose $\delta^{\mathcal{X}} : X \rightarrow 2^{\text{At}}$. Then

$$s^{\mathcal{X}}(x) := \sum \{ \alpha \in \text{At} \mid \delta^{\mathcal{X}}(x)(\alpha) = 1 \}$$

is a solution to \mathcal{X} .

(S2) Let $\mathcal{Y} = \langle Y, \delta^{\mathcal{Y}} \rangle$ and $\mathcal{Z} = \langle Z, \delta^{\mathcal{Z}} \rangle$ be simple G -coalgebras, and let $h \in G(Y + Z)$ be such that $\mathcal{X} = (\mathcal{Y} + \mathcal{Z})[Y, h]$. By induction, \mathcal{Y} and \mathcal{Z} admit solutions $s^{\mathcal{Y}}$ and $s^{\mathcal{Z}}$ respectively; we need to find a solution $s^{\mathcal{X}}$ to $X = Y + Z$. The idea is to retain the solution that we had for states in \mathcal{Z} —whose behavior has not changed under uniform continuation—while modifying the solution to states in \mathcal{Y} in order to account for transitions from h . To this end, we choose the following expressions:

$$b := \sum \{ \alpha \in \text{At} \mid h(\alpha) \in \Sigma \times X \} \quad \ell := \left(\bigoplus_{\alpha \leq b} [h(\alpha)]_{s^{\mathcal{Y}}} \right)^{(b)} \cdot \bigoplus_{\alpha \leq \bar{b}} [h(\alpha)]_{s^{\mathcal{Z}}}$$

We can then define s as follows:

$$s^{\mathcal{X}}(x) := \begin{cases} s^{\mathcal{Y}}(x) \cdot \ell & x \in Y \\ s^{\mathcal{Z}}(x) & x \in Z \end{cases}$$

A detailed argument showing that s is a solution can be found in the appendix. \square

As it turns out, we can do a round-trip, showing that the solution to the (initial state of the) Thompson automaton for an expression is equivalent to the original expression.

Theorem 5.4.8 (Correctness II). *Let $e \in \text{Exp}$. Then \mathcal{X}_e^{ι} admits a solution s such that $e \equiv s(\iota)$.*

Finally, we show that the automata construction of the previous section gives simple automata.

Theorem 5.4.9 (Simplicity of Thompson construction). *\mathcal{X}_e and \mathcal{X}_e^{ι} are simple for all expressions e .*

Proof. We proceed by induction on e . In the base, let $\mathcal{Z} = \langle \emptyset, \emptyset \rangle$ and $\mathcal{I} = \langle \{*\}, * \mapsto \mathbf{1} \rangle$ denote the coalgebras with no states and with a single all-accepting state, respectively. Note that \mathcal{Z} and \mathcal{I} are simple, and that for $b \in \text{BExp}$ and $p \in \Sigma$ we have $\mathcal{X}_b = \mathcal{Z}$ and $X_p = \mathcal{I}$.

All of the operations used to build \mathcal{X}_e , as detailed in Figure 5.4, can be phrased in terms of an appropriate uniform continuation of a coproduct; for instance, when $e = f^{(b)}$ we have that $\mathcal{X}_e = (\mathcal{X}_f + \mathcal{I})[X_f, \iota_e]$. Consequently, the Thompson automaton \mathcal{X}_e is simple by construction. Finally, observe that $\mathcal{X}_e^\iota = (\mathcal{I} + \mathcal{X}_e)[\{*\}, \iota_e]$; hence, \mathcal{X}_e^ι is simple as well. \square

Theorems 5.4.1, 5.4.7, and 5.4.9 now give us the desired Kleene theorem.

Corollary 5.4.10 (Kleene Theorem). *Let $L \subseteq \text{GS}$. The following are equivalent:*

1. $L = \llbracket e \rrbracket$ for a GKAT expression e .
2. $L = \ell^{\mathcal{X}}(\iota)$ for a simple, finite-state G -automaton \mathcal{X} with initial state ι .

5.5 Decision Procedure

We saw in the last section that GKAT expressions can be efficiently converted to equivalent automata with a linear number of states. Equivalence of automata can be established algorithmically, supporting a decision procedure for GKAT that is significantly more efficient than decision procedures for KAT. In this section, we describe our algorithm.

First, we define bisimilarity of automata states in the usual way [98].

Definition 5.5.1 (Bisimilarity). Let \mathcal{X} and \mathcal{Y} be G -coalgebras. A *bisimulation* between \mathcal{X} and \mathcal{Y} is a binary relation $R \subseteq X \times Y$ such that if $x R y$, then the following implications hold:

- (i) if $\delta^{\mathcal{X}}(x)(\alpha) \in 2$, then $\delta^{\mathcal{Y}}(y)(\alpha) = \delta^{\mathcal{X}}(x)(\alpha)$; and
- (ii) if $\delta^{\mathcal{X}}(x)(\alpha) = (p, x')$, then $\delta^{\mathcal{Y}}(y)(\alpha) = (p, y')$ and $x' R y'$ for some y' .

States x and y are called *bisimilar*, denoted $x \sim y$, if there exists a bisimulation relating x and y . \square

As usual, we would like to reduce automata equivalence to bisimilarity. It is easy to see that bisimilar states recognize the same language.

Lemma 5.5.2. *If \mathcal{X} and \mathcal{Y} are G -coalgebras with bisimilar states $x \sim y$, then $\ell^{\mathcal{X}}(x) = \ell^{\mathcal{Y}}(y)$.*

Proof. We verify that $w \in \ell^{\mathcal{X}}(x) \Leftrightarrow w \in \ell^{\mathcal{Y}}(y)$ by induction on the length of $w \in \text{GS}$:

- For $\alpha \in \text{GS}$, we have $\alpha \in \ell^{\mathcal{X}}(x) \Leftrightarrow \delta^{\mathcal{X}}(x)(\alpha) = 1 \Leftrightarrow \delta^{\mathcal{Y}}(y)(\alpha) = 1 \Leftrightarrow \alpha \in \ell^{\mathcal{Y}}(y)$.
- For $\alpha pw \in \text{GS}$, we use bisimilarity and the induction hypothesis to derive

$$\begin{aligned} \alpha pw \in \ell^{\mathcal{X}}(x) &\iff \exists x'. \delta^{\mathcal{X}}(x)(\alpha) = (p, x') \wedge w \in \ell^{\mathcal{X}}(x') \\ &\iff \exists y'. \delta^{\mathcal{Y}}(y)(\alpha) = (p, y') \wedge w \in \ell^{\mathcal{Y}}(y') \iff \alpha pw \in \ell^{\mathcal{Y}}(y). \quad \square \end{aligned}$$

The reverse direction, however, does not hold for G -coalgebras in general. To see the problem, consider the following automaton, where $\alpha \in \text{At}$ is an atom and $p \in \Sigma$ is an action:

$$\begin{array}{ccc} s_1 & \xrightarrow{\alpha/p} & s_2 \\ \bullet & \longrightarrow & \bullet \end{array}$$

Both states recognize the empty language, that is *i.e.*, $\ell(s_1) = \ell(s_2) = \emptyset$; but s_2 rejects immediately, whereas s_1 may first take a transition. As a result, s_1 and s_2 are not bisimilar. Intuitively, the language accepted by a state does not distinguish between early and late rejection, whereas bisimilarity does. We solve this by disallowing late rejection, *i.e.*, transitions that can never lead to an accepting state; we call coalgebras that respect this restriction *normal*.

5.5.1 Normal Coalgebras

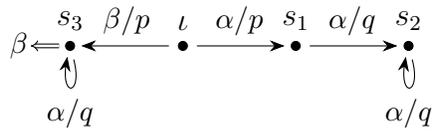
We classify states and coalgebras as follows.

Definition 5.5.3 (Live, Dead, Normal). Let $\mathcal{X} = \langle X, \delta^{\mathcal{X}} \rangle$ denote a G -coalgebra. A state $s \in X$ is *accepting* if $\delta^{\mathcal{X}}(s)(\alpha) = 1$ for some $\alpha \in \text{At}$. A state is *live* if it can transition to an

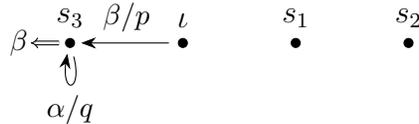
accepting state in a finite number of steps, or *dead* otherwise. A coalgebra is *normal* if it contains no transitions to dead states. \square

Remark 5.5.4. Note that, equivalently, a state is live iff $\ell^X(s) \neq \emptyset$ and dead iff $\ell^X(s) = \emptyset$. Dead states can exist in a normal coalgebra, but they must immediately reject all $\alpha \in \text{At}$, since any successor of a dead state would also be dead. \square

Example 5.5.5. Consider the following automaton.



The state s_3 is accepting. The states l and s_3 are live, since they can reach an accepting state. The states s_1 and s_2 are dead, since they can only reach non-accepting states. The automaton is not normal, since it contains the transitions $l \xrightarrow{\alpha/p} s_1$, $s_1 \xrightarrow{\alpha/q} s_2$, and $s_2 \xrightarrow{\alpha/q} s_1$ to dead states s_1 and s_2 . We can *normalize* the automaton by removing these transitions:



The resulting automaton is normal: the dead states s_1 and s_2 reject all $\alpha \in \text{At}$ immediately. \square

The example shows how coalgebra can be normalized. Formally, let $\mathcal{X} = \langle X, \delta \rangle$ denote a coalgebra with dead states $D \subseteq X$. We define the normalized coalgebra $\widehat{\mathcal{X}} := \langle X, \widehat{\delta} \rangle$ as follows:

$$\widehat{\delta}(s)(\alpha) := \begin{cases} 0 & \text{if } \delta(s)(\alpha) \in \Sigma \times D \\ \delta(s)(\alpha) & \text{otherwise.} \end{cases}$$

Lemma 5.5.6 (Correctness of normalization). *Let \mathcal{X} be a G -coalgebra. Then the following holds:*

(i) \mathcal{X} and $\widehat{\mathcal{X}}$ have the same solutions: that is, $s : X \rightarrow \text{Exp}$ solves \mathcal{X} if and only if s solves $\widehat{\mathcal{X}}$; and

(ii) \mathcal{X} and $\widehat{\mathcal{X}}$ accept the same languages: that is, $\ell^{\mathcal{X}} = \ell^{\widehat{\mathcal{X}}}$; and

(iii) $\widehat{\mathcal{X}}$ is normal.

Proof. For the first claim, suppose s solves \mathcal{X} . It suffices (by Lemma C.1.3) to show that for $x \in X$ and $\alpha \in \text{At}$ we have $\alpha \cdot s(x) \equiv \alpha \cdot \lfloor \delta^{\widehat{\mathcal{X}}}(x)(\alpha) \rfloor_s$. We have two cases.

- If $\delta^{\mathcal{X}}(x)(\alpha) = (p, x')$ with x' dead, then by Proposition 5.4.5 we know that $\llbracket s(x') \rrbracket = \ell^{\mathcal{X}}(x') = \emptyset$. By Corollary 5.3.13, it follows that $s(x') \equiv 0$. Recalling that $\delta^{\widehat{\mathcal{X}}}(x)(\alpha) = 0$ by construction,

$$\alpha \cdot s(x) \equiv \alpha \cdot \lfloor \delta^{\mathcal{X}}(x)(\alpha) \rfloor_s \equiv \alpha \cdot p \cdot s(x') \equiv \alpha \cdot 0 \equiv \alpha \cdot \lfloor \delta^{\widehat{\mathcal{X}}}(x)(\alpha) \rfloor_s$$

- Otherwise, we know that $\delta^{\widehat{\mathcal{X}}}(x)(\alpha) = \delta^{\mathcal{X}}(x)(\alpha)$, and thus

$$\alpha \cdot s(x) \equiv \alpha \cdot \lfloor \delta^{\mathcal{X}}(x)(\alpha) \rfloor_s \equiv \alpha \cdot \lfloor \delta^{\widehat{\mathcal{X}}}(x)(\alpha) \rfloor_s$$

The other direction of the first claim can be shown analogously.

For the second claim, we can establish $x \in \ell^{\mathcal{X}}(s) \Leftrightarrow x \in \ell^{\widehat{\mathcal{X}}}(s)$ for all states s by a straightforward induction on the length of $x \in \text{GS}$, using that dead states accept the empty language.

For the third claim, we note that the dead states of \mathcal{X} and $\widehat{\mathcal{X}}$ coincide by claim two; thus $\widehat{\mathcal{X}}$ has no transition to dead states by construction. \square

5.5.2 Bisimilarity for Normal Coalgebras

We would like to show that, for normal coalgebras, states are bisimilar if and only if they accept the same language. This will allow us to reduce language-equivalence to bisimilarity, which is easy to establish algorithmically. We need to take a slight detour.

Recall the determinacy property satisfied by GKAT languages (Definition 5.2.2): a language $L \subseteq \text{GS}$ is deterministic if, whenever strings $x, y \in L$ agree on the first n atoms, they also agree on the first n actions (or absence thereof). Let $\mathcal{L} \subseteq 2^{\text{GS}}$ denote the set of all such deterministic languages and define, for α an atom and p an action:

$$L_{\alpha p} := \{x \in \text{GS} \mid \alpha p x \in L\}.$$

\mathcal{L} carries a coalgebra structure $\langle \mathcal{L}, \delta^{\mathcal{L}} \rangle$ whose transition map $\delta^{\mathcal{L}}$ is the semantic Brzozowski derivative:

$$\delta^{\mathcal{L}}(L)(\alpha) := \begin{cases} (p, L_{\alpha p}) & \text{if } L_{\alpha p} \neq \emptyset \\ 1 & \text{if } \alpha \in L \\ 0 & \text{otherwise.} \end{cases}$$

Note that the map is well-defined by determinacy: precisely one of the three cases holds.

Next, we define *structure-preserving maps* between G -coalgebras in the usual way:

Definition 5.5.7 (Homomorphism). A homomorphism between G -coalgebras \mathcal{X} and \mathcal{Y} is a map $h: X \rightarrow Y$ from states of \mathcal{X} to states of \mathcal{Y} that respects the transition structures in the following sense:

$$\delta^{\mathcal{Y}} \circ h = (G h) \circ \delta^{\mathcal{X}}.$$

More concretely, for all $\alpha \in \text{At}$, $p \in \Sigma$, and $x, x' \in X$,

- (i) if $\delta^{\mathcal{X}}(x)(\alpha) \in 2$, then $\delta^{\mathcal{Y}}(h(x))(\alpha) = \delta^{\mathcal{X}}(x)(\alpha)$; and
- (ii) if $\delta^{\mathcal{X}}(x)(\alpha) = (p, x')$, then $\delta^{\mathcal{Y}}(h(x))(\alpha) = (p, h(x'))$. □

We can now show that the acceptance map $\ell^{\mathcal{X}}: X \rightarrow 2^{\text{GS}}$ is structure-preserving in the above sense. Moreover, it is *the only* structure-preserving map from states to deterministic languages:

Theorem 5.5.8 (Final Coalgebra). *If \mathcal{X} is normal, then $\ell^{\mathcal{X}}: X \rightarrow 2^{\text{GS}}$ is the unique homomorphism $\mathcal{X} \rightarrow \mathcal{L}$.*

Proof. We need to establish the following claims:

- (1) the language $\ell^{\mathcal{X}}(s)$ is deterministic for all states $s \in X$;
- (2) the map $\ell^{\mathcal{X}}$ is a homomorphism $\mathcal{X} \rightarrow \mathcal{L}$; and
- (3) the map $\ell^{\mathcal{X}}$ is the unique homomorphism $\mathcal{X} \rightarrow \mathcal{L}$.

Before we turn to proving these claims, we establish the following implication:

$$\delta^{\mathcal{X}}(s)(\alpha) = (p, t) \implies \ell^{\mathcal{X}}(s)_{\alpha p} = \ell^{\mathcal{X}}(t). \quad (5.3)$$

To see that it holds, we observe that given the premise, we have

$$w \in \ell^{\mathcal{X}}(s)_{\alpha p} \iff \alpha p w \in \ell^{\mathcal{X}}(s) \iff w \in \ell^{\mathcal{X}}(t).$$

We can now show the main claims:

- (1) We begin by showing that $\ell^{\mathcal{X}}(s)$ is deterministic for $s \in X$. We need to show that

$$\left. \begin{array}{l} x = \alpha_1 p_1 \alpha_2 p_2 \cdots \alpha_n p_n x' \in \ell^{\mathcal{X}}(s) \\ y = \alpha_1 q_1 \alpha_2 q_2 \cdots \alpha_n q_n y' \in \ell^{\mathcal{X}}(s) \end{array} \right\} \implies p_i = q_i \quad (\forall 1 \leq i \leq n),$$

where the final actions may be absent (*i.e.*, $p_n = x' = \varepsilon$ or $q_n = y' = \varepsilon$) but all other actions are present (*i.e.*, $p_i, q_i \in \Sigma$ for $i < n$). We proceed by induction on n . The case $n = 0$ is trivially true. For $n \geq 1$, take x and y as above. We proceed by case distinction:

- If p_1 is absent, *i.e.* $n = 1$ and $p_1 = x' = \varepsilon$, then by Equation (5.2) we must have $\delta^{\mathcal{X}}(s)(\alpha_1) = 1$ and thus cannot have $q_1 \in \Sigma$; hence q_1 is also absent, as required.
- Otherwise $p_1 \in \Sigma$ is a proper action. Then by Equation (5.2), there exist $t, t' \in X$ such that:

$$\begin{aligned} \delta^{\mathcal{X}}(s)(\alpha_1) &= (p_1, t) \wedge \alpha_2 p_2 \cdots \alpha_n p_n x' \in \ell^{\mathcal{X}}(t) \\ \delta^{\mathcal{X}}(s)(\alpha_1) &= (q_1, t') \wedge \alpha_2 q_2 \cdots \alpha_n q_n y' \in \ell^{\mathcal{X}}(t') \end{aligned}$$

This implies $(p_1, t) = (q_1, t')$. Thus $p_1 = q_1$ follows immediately and $p_i = q_i$ for $i > 1$ follows by induction hypothesis.

(2) Next, we show that $\ell^{\mathcal{X}}$ is a homomorphism: $(G \ell^{\mathcal{X}}) \circ \delta^{\mathcal{X}} = \delta^{\mathcal{L}} \circ \ell^{\mathcal{X}}$.

If $\delta^{\mathcal{X}}(x)(\alpha) = 1$, then $\alpha \in \ell^{\mathcal{X}}(x)$ and hence $\delta^{\mathcal{L}}(\ell^{\mathcal{X}}(x))(\alpha) = 1$ by definition of $\delta^{\mathcal{L}}$.

If $\delta^{\mathcal{X}}(x)(\alpha) = 0$, then $\alpha \notin \ell^{\mathcal{X}}(x)$ and for all $p \in \Sigma, w \in \text{GS}$, $\alpha pw \notin \ell^{\mathcal{X}}(x)$ and hence $\ell^{\mathcal{X}}(x)_{\alpha p} = \emptyset$. Thus $\delta^{\mathcal{L}}(\ell^{\mathcal{X}}(x))(\alpha) = 0$ by definition of $\delta^{\mathcal{L}}$.

If $\delta^{\mathcal{X}}(x)(\alpha) = \langle p, y \rangle$, then y is live by normality and thus there exists a word $w_y \in \ell^{\mathcal{X}}(y)$. Thus,

$$\begin{aligned}
& \alpha pw_y \in \ell^{\mathcal{X}}(x) && \text{(def. } \ell^{\mathcal{X}}) \\
\implies & w_y \in \ell^{\mathcal{X}}(x)_{\alpha p} && \text{(def. } L_{\alpha p}) \\
\implies & \delta^{\mathcal{L}}(\ell^{\mathcal{X}}(x))(\alpha) = \langle p, \ell^{\mathcal{X}}(x)_{\alpha p} \rangle && \text{(def. } \delta^{\mathcal{L}}) \\
\implies & \delta^{\mathcal{L}}(\ell^{\mathcal{X}}(x))(\alpha) = \langle p, \ell^{\mathcal{X}}(y) \rangle && \text{(Equation (5.3))}
\end{aligned}$$

(3) For uniqueness, let L denote an arbitrary homomorphism $\mathcal{X} \rightarrow \mathcal{L}$. We will show that

$$w \in L(x) \iff w \in \ell^{\mathcal{X}}(x)$$

by induction on $|w|$.

For $w = \alpha$,

$$\begin{aligned}
\alpha \in L(x) & \iff \delta^{\mathcal{L}}(L(x)) = 1 && \text{(def. } \delta) \\
& \iff \delta^{\mathcal{X}}(x)(\alpha) = 1 && (L \text{ is hom.}) \\
& \iff \alpha \in \ell^{\mathcal{X}}(x) && \text{(def. } \ell^{\mathcal{X}})
\end{aligned}$$

For $w = \alpha pv$,

$$\begin{aligned}
& \alpha pv \in L(x) \\
& \iff \delta^{\mathcal{L}}(L(x))(\alpha) = \langle p, L(x)_{\alpha p} \rangle \wedge v \in L(x)_{\alpha p} && \text{(def. } \delta^{\mathcal{L}}, L_{\alpha p}\text{)} \\
& \iff \exists y. \delta^{\mathcal{X}}(x)(\alpha) = \langle p, y \rangle \wedge v \in L(y) && (L \text{ is hom., Equation (5.3)}) \\
& \iff \exists y. \delta^{\mathcal{X}}(x)(\alpha) = \langle p, y \rangle \wedge v \in \ell^{\mathcal{X}}(y) && \text{(induction)} \\
& \iff \alpha pv \in \ell^{\mathcal{X}}(x) && \text{(def. } \ell^{\mathcal{X}}\text{)}
\end{aligned}$$

This concludes the proof. □

Note that, since the identity function is trivially a homomorphism, Theorem 5.5.8 implies that $\ell^{\mathcal{L}}$ is the identity. That is, in the G -coalgebra \mathcal{L} , the state $L \in \mathcal{L}$ accepts the language L ! This proves that every deterministic language is recognized by a G -coalgebra, though possibly with an infinite number of states.

The property from Theorem 5.5.8 says that \mathcal{L} is *final* for normal G -coalgebras. The desired connection between bisimilarity and language-equivalence is then a standard corollary [149]:

Corollary 5.5.9. *Let \mathcal{X} and \mathcal{Y} be normal G -coalgebras with states s and t . The following are equivalent:*

- (i) $s \sim t$;
- (ii) $\ell^{\mathcal{X}}(s) = \ell^{\mathcal{Y}}(t)$.

Proof. The implication from (i) to (ii) is Lemma 5.5.2. For the implication from (ii) to (i), we observe that the relation $R := \{(s, t) \in X \times Y \mid \ell^{\mathcal{X}}(s) = \ell^{\mathcal{Y}}(t)\}$ is a bisimulation, using that $\ell^{\mathcal{X}}$ and $\ell^{\mathcal{Y}}$ are homomorphisms by Theorem 5.5.8:

- Suppose $s R t$ and $\delta^{\mathcal{X}}(s)(\alpha) \in 2$. Then $\delta^{\mathcal{X}}(s)(\alpha) = \delta^{\mathcal{L}}(\ell^{\mathcal{X}}(s))(\alpha) = \delta^{\mathcal{L}}(\ell^{\mathcal{Y}}(t))(\alpha) = \delta^{\mathcal{Y}}(t)(\alpha)$.

- Suppose $s R t$ and $\delta^x(s)(\alpha) = (p, s')$. Then $(p, \ell^x(s')) = \delta^{\mathcal{L}}(\ell^x(s))(\alpha) = \delta^{\mathcal{L}}(\ell^y(t))(\alpha)$.

This implies that $\delta^y(t)(\alpha) = (p, t')$ for some t' , using that ℓ^y is a homomorphism.

Hence

$$(p, \ell^y(t')) = \delta^{\mathcal{L}}(\ell^y(t))(\alpha) = (p, \ell^x(s'))$$

by the above equation, which implies $s' R t'$ as required. \square

5.5.3 Deciding Equivalence

We now have all the ingredients required for deciding efficiently whether a pair of expressions are equivalent. Given two expressions e_1 and e_2 , the algorithm proceeds as follows:

1. Convert e_1 and e_2 to equivalent Thompson automata \mathcal{X}_1 and \mathcal{X}_2 ;
2. Normalize the automata, obtaining $\widehat{\mathcal{X}}_1$ and $\widehat{\mathcal{X}}_2$;
3. Check bisimilarity of the start states ι_1 and ι_2 using Hopcroft-Karp (see Algorithm 1);
4. Return **true** if $\iota_1 \sim \iota_2$, or return **false** otherwise.

Theorem 5.5.10. *The above algorithm decides whether $\llbracket e_1 \rrbracket = \llbracket e_2 \rrbracket$ in time $\mathcal{O}(n \cdot \alpha(n))$ for $|At|$ constant, where α denotes the inverse Ackermann function and $n = |e_1| + |e_2|$ bounds the size of the expressions.*

Proof. The algorithm is correct by Theorem 5.4.1, Lemma 5.5.6, and Corollary 5.5.9:

$$\llbracket e_1 \rrbracket = \llbracket e_2 \rrbracket \iff \ell^{\mathcal{X}_1}(\iota_1) = \ell^{\mathcal{X}_2}(\iota_2) \iff \ell^{\widehat{\mathcal{X}}_1}(\iota_1) = \ell^{\widehat{\mathcal{X}}_2}(\iota_2) \iff \iota_1 \sim \iota_2$$

For the complexity claim, we analyze the running time of steps (1)–(3) of the algorithm:

1. Recall by Proposition 5.4.2 that the Thompson construction converts e_i to an automaton with $\mathcal{O}(|e_i|)$ states in time $\mathcal{O}(|e_i|)$. Hence this step takes time $\mathcal{O}(n)$.

Algorithm 1: Hopcroft and Karp’s algorithm [65], adapted to G -automata.

Input: G -automata $\mathcal{X} = \langle X, \delta^{\mathcal{X}}, \iota^{\mathcal{X}} \rangle$ and $\mathcal{Y} = \langle Y, \delta^{\mathcal{Y}}, \iota^{\mathcal{Y}} \rangle$, finite and normal; X , Y disjoint.

Output: **true** if \mathcal{X} and \mathcal{Y} are equivalent, **false** otherwise.

```

1 todo ← Queue.singleton( $\iota^{\mathcal{X}}, \iota^{\mathcal{Y}}$ ); // state pairs that need to be checked
2 forest ← UnionFind.disjointForest( $X \uplus Y$ );
3 while not todo.isEmpty() do
4    $x, y \leftarrow$  todo.pop();
5    $r_x, r_y \leftarrow$  forest.find( $x$ ), forest.find( $y$ );
6   if  $r_x = r_y$  then continue; // safe to assume bisimilar
7   for  $\alpha \in \text{At}$  do // check Definition 5.5.1
8     switch  $\delta^{\mathcal{X}}(x)(\alpha), \delta^{\mathcal{Y}}(y)(\alpha)$  do
9       case  $b_1, b_2$  with  $b_1 = b_2$  do // case (i) of Definition 5.5.1
10        | continue
11        case  $(p, x'), (p, y')$  do // case (ii) of Definition 5.5.1
12        | todo.push( $x', y'$ )
13        otherwise do return false; // not bisimilar
14      end
15    end
16    forest.union( $r_x, r_y$ ); // mark as bisimilar
17 end
18 return true;

```

2. Normalizing \mathcal{X}_i amounts to computing its dead states. This requires time $\mathcal{O}(|e_i|)$ using a breadth-first traversal as follows (since there are at most $|\text{At}| \in \mathcal{O}(1)$ transitions per state). We find all states that can reach an accepting state by first marking all accepting states, and then performing a reverse breadth-first search rooted at the accepting states. All marked states are then live; all unmarked states are dead.

3. Since $\widehat{\mathcal{X}}_i$ has $\mathcal{O}(|e_i|)$ states and there are at most $|\text{At}| \in \mathcal{O}(1)$ transitions per state, Hopcroft-Karp requires time $\mathcal{O}(n \cdot \alpha(n))$ by a classic result due to Tarjan [167]. \square

Theorem 5.5.10 establishes a stark complexity gap with KAT—in KAT, the same decision problem is PSPACE-complete [26] even for a constant number of atoms. Intuitively,

the gap arises because GKAT expressions can be translated to linear-size deterministic automata, whereas KAT expressions may be nondeterministic and may require exponential-size deterministic automata.

A shortcoming of Algorithm 1 is that it may scale poorly if the number of atoms $|At|$ is large. It is worth noting that there are symbolic variants [138] of the algorithm that avoid enumerating At explicitly (cf. Line 7 of Algorithm 1), and can often scale to very large alphabets in practice. In the worst case, however, we have the following hardness result:

Proposition 5.5.11. *If $|At|$ is not a constant, GKAT equivalence is co-NP-hard, but in PSPACE.*

Proof. For the hardness result, we observe that Boolean *unsatisfiability* reduces to GKAT equivalence: $b \in \text{BExp}$ is unsatisfiable, interpreting the primitive tests as variables, iff $\llbracket b \rrbracket = \emptyset$. The PSPACE upper bound is inherited from KAT by Remark 5.2.1. \square

5.6 Completeness for the Language Model

In Section 5.3 we presented an axiomatization that is sound with respect to the language model, and put forward the conjecture that it is also complete. We have already proven a partial completeness result (Corollary 5.3.13). In this section, we return to this matter and show we can prove completeness with a generalized version of (W3).

5.6.1 Systems of Left-Affine Equations

A *system of left-affine equations* (or simply, a *system*) is a finite set of equations of the form

$$\begin{aligned}
 \mathbf{x}_1 &= e_{11}\mathbf{x}_1 +_{b_{11}} \cdots +_{b_{1(n-1)}} e_{1n}\mathbf{x}_n +_{b_{1n}} d_1 \\
 &\quad \vdots \\
 \mathbf{x}_n &= e_{n1}\mathbf{x}_1 +_{b_{n1}} \cdots +_{b_{n(n-1)}} e_{nn}\mathbf{x}_n +_{b_{nn}} d_n
 \end{aligned} \tag{5.4}$$

where $+_b$ associates to the right, the \mathbf{x}_i are variables, the e_{ij} are GKAT expressions, and the b_{ij} and d_i are Boolean guards satisfying the following row-wise disjointness property for row $1 \leq i \leq n$:

- for all $j \neq k$, the guards b_{ij} and b_{ik} are disjoint: $b_{ij} \cdot b_{ik} \equiv_{\text{BA}} 0$; and
- for all $1 \leq j \leq n$, the guards b_{ij} and d_i are disjoint: $b_{ij} \cdot d_i \equiv_{\text{BA}} 0$.

Note that by the disjointness property, the ordering of the summands is irrelevant: the system is invariant (up to \equiv) under column permutations. A *solution* to such a system is a function $s : \{\mathbf{x}_1, \dots, \mathbf{x}_n\} \rightarrow \text{Exp}$ assigning expressions to variables such that, for row $1 \leq i \leq n$:

$$s(\mathbf{x}_i) \equiv e_{i1} \cdot s(\mathbf{x}_1) +_{b_{i1}} \cdots +_{b_{i(n-1)}} e_{in} \cdot s(\mathbf{x}_n) +_{b_{in}} d_i$$

Note that any finite G -coalgebra gives rise to a system where each variable represents a state, and the equations define what it means to be a solution to the coalgebra (c.f. Definition 5.4.3); indeed, a solution to a G -coalgebra is precisely a solution to its corresponding system of equations, and vice versa. In particular, for a coalgebra \mathcal{X} with states x_1 to x_n , the parameters for equation i are:

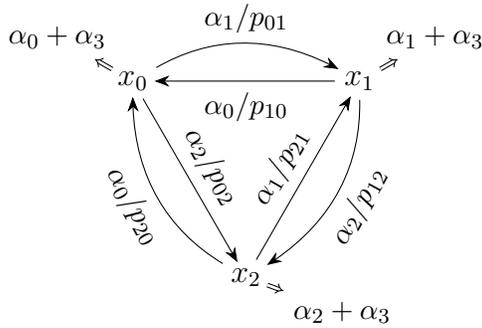
$$\begin{aligned}
 b_{ij} &= \sum \{ \alpha \in \text{At} \mid \delta^{\mathcal{X}}(x_i)(\alpha) \in \Sigma \times \{x_j\} \} \\
 d_i &= \sum \{ \alpha \in \text{At} \mid \delta^{\mathcal{X}}(x_i)(\alpha) = 1 \} & e_{ij} &= \bigoplus_{\alpha : \delta^{\mathcal{X}}(x_i)(\alpha) = (p_\alpha, x_j)} p_\alpha
 \end{aligned}$$

Systems arising from G -coalgebras have a useful property: for all e_{ij} , it holds that $E(e_{ij}) \equiv 0$. This property is analogous to the *empty word property* in Salomaa's axiomatization of regular languages [152]; we call such systems *Salomaa*.

To obtain a general completeness result beyond Section 5.3.4, we assume an additional axiom:

Uniqueness axiom. Any system of left-affine equations that is Salomaa has at most one solution, modulo \equiv . More precisely, whenever s and s' are solutions to a Salomaa system, it holds that $s(x_i) \equiv s'(x_i)$ for all $1 \leq i \leq n$.

Remark 5.6.1. We do not assume that a solution always exists, but only that if it does, then it is unique up to \equiv . It would be unsound to assume that all such systems have solutions; the following automata and its system, due to [98], constitute a counterexample:



$$\begin{aligned} \mathbf{x}_0 &\equiv p_{01}\mathbf{x}_1 + \alpha_1 p_{02}\mathbf{x}_2 + \alpha_2 (\alpha_0 + \alpha_3) \\ \mathbf{x}_1 &\equiv p_{10}\mathbf{x}_0 + \alpha_0 p_{12}\mathbf{x}_2 + \alpha_2 (\alpha_1 + \alpha_3) \\ \mathbf{x}_2 &\equiv p_{20}\mathbf{x}_0 + \alpha_1 p_{21}\mathbf{x}_1 + \alpha_0 (\alpha_2 + \alpha_3) \end{aligned}$$

As shown in [98], no corresponding while program exists for this system. \square

When $n = 1$, a system is a single equation of the form $x = ex +_b d$. In this case, (W1) tells us that a solution does exist, namely $e^{(b)}d$, and (W3) says that this solution is unique up to \equiv under the proviso $E(e) \equiv 0$. In this sense, we can regard the uniqueness axiom as a generalization of (W3) to systems with multiple variables.

Theorem 5.6.2. *The uniqueness axiom is sound in the model of guarded strings: given a system of left-affine equations as in (5.4) that is Salomaa, there exists at most one $R : \{x_1, \dots, x_n\} \rightarrow 2^{\text{GS}}$ s.t.*

$$R(x_i) = \left(\bigcup_{1 \leq j \leq n} \llbracket b_{ij} \rrbracket \diamond \llbracket e_{ij} \rrbracket \diamond R(x_j) \right) \cup \llbracket d_i \rrbracket$$

Proof Sketch. We recast this system as a matrix-vector equation of the form $x = Mx + D$ in the KAT of n -by- n matrices over 2^{GS} ; solutions to x in this equation are in one-to-one correspondence with functions R as above. We then show that the map $\sigma(x) = Mx + D$ on the set of n -dimensional vectors over 2^{GS} is contractive in a certain metric, and therefore has a unique fixpoint by the Banach fixpoint theorem; hence, there can be at most one solution x . \square

5.6.2 General Completeness

Using the generalized version of the fixpoint axiom, we can now establish completeness.

Theorem 5.6.3 (Completeness). *The axioms are complete for all GKAT expressions: given $e_1, e_2 \in \text{Exp}$,*

$$\llbracket e_1 \rrbracket = \llbracket e_2 \rrbracket \implies e_1 \equiv e_2.$$

Proof. Let \mathcal{X}_1 and \mathcal{X}_2 be the Thompson automata corresponding to e_1 and e_2 , with initial states ι_1 and ι_2 , respectively. Theorem 5.4.8 shows there are solutions s_1 and s_2 , with $s_1(\iota_1) \equiv e_1$ and $s_2(\iota_2) \equiv e_2$; and we know from Lemma 5.5.6 that s_1 and s_2 solve the normalized automata $\widehat{\mathcal{X}}_1$ and $\widehat{\mathcal{X}}_2$. By Lemma 5.5.6, Theorem 5.4.1, and the premise, we derive that $\widehat{\mathcal{X}}_1$ and $\widehat{\mathcal{X}}_2$ accept the same language:

$$\ell^{\widehat{\mathcal{X}}_1}(\iota_1) = \ell^{\mathcal{X}_1}(\iota_1) = \llbracket e_1 \rrbracket = \llbracket e_2 \rrbracket = \ell^{\mathcal{X}_2}(\iota_2) = \ell^{\widehat{\mathcal{X}}_2}(\iota_2).$$

This implies, by Corollary 5.5.9, that there is a bisimulation R between $\widehat{\mathcal{X}}_1$ and $\widehat{\mathcal{X}}_2$ relating ι_1 and ι_2 . This bisimulation can be given the following transition structure,

$$\delta^{\mathcal{R}}(x_1, x_2)(\alpha) := \begin{cases} 0 & \text{if } \delta^{\widehat{\mathcal{X}}_1}(x_1)(\alpha) = 0 \text{ and } \delta^{\widehat{\mathcal{X}}_2}(x_2)(\alpha) = 0 \\ 1 & \text{if } \delta^{\widehat{\mathcal{X}}_1}(x_1)(\alpha) = 1 \text{ and } \delta^{\widehat{\mathcal{X}}_2}(x_2)(\alpha) = 1 \\ (p, (x'_1, x'_2)) & \text{if } \delta^{\widehat{\mathcal{X}}_1}(x_1)(\alpha) = (p, x'_1) \text{ and } \delta^{\widehat{\mathcal{X}}_2}(x_2)(\alpha) = (p, x'_2) \end{cases}$$

turning $\mathcal{R} = \langle R, \delta^{\mathcal{R}} \rangle$ into a G -coalgebra; note that $\delta^{\mathcal{R}}$ is well-defined since R is a bisimulation.

Now, define $s'_1, s'_2 : R \rightarrow \text{Exp}$ by $s'_1(x_1, x_2) = s_1(x_1)$ and $s'_2(x_1, x_2) = s_2(x_2)$. We claim that s'_1 and s'_2 are both solutions to \mathcal{R} ; to see this, note that for $\alpha \in \text{At}$, $(x_1, x_2) \in R$, and $i \in \{1, 2\}$, we have that

$$\begin{aligned} \alpha \cdot s'_i(x_i, x_i) &\equiv \alpha \cdot s_i(x_i) && \text{(Def. } s'_i) \\ &\equiv \alpha \cdot [\delta^{\widehat{\mathcal{X}}_i}(x_i)(\alpha)]_{s_i} && (s_i \text{ solves } \widehat{\mathcal{X}}_i) \\ &\equiv \alpha \cdot [\delta^{\mathcal{R}}(x_1, x_2)(\alpha)]_{s'_i} && \text{(Def. } s'_i \text{ and } [-]) \end{aligned}$$

Since the system of left-affine equations induced by \mathcal{R} is Salomaa, the uniqueness axiom then tells us that $s_1(t_1) = s'_1(t_1, t_2) \equiv s'_2(t_1, t_2) = s_2(t_2)$; hence, we can conclude that $e_1 \equiv e_2$. \square

5.7 Related Work

Program schematology is one of the oldest areas of study in the mathematics of computing. It is concerned with questions of translation and representability among and within classes of program schemes, such as flowcharts, while programs, recursion schemes, and schemes with various data structures such as counters, stacks, queues, and dictionaries [47, 66, 107, 131, 160]. A classical pursuit in this area was to find mechanisms to transform unstructured flowcharts to structured form [10, 17, 86, 118, 127, 134, 143, 176]. A seminal result was the *Böhm-Jacopini theorem* [17], which established that all flowcharts can be converted to while programs provided auxiliary variables are introduced. Böhm and Jacopini conjectured that the use of auxiliary variables was necessary in general, and this conjecture was confirmed independently by Ashcroft and Manna [10] and Kosaraju [86].

Early results in program schematology, including those of [10, 17, 86], were typically formulated at the first-order uninterpreted (schematic) level. However, many restructuring operations can be accomplished without reference to first-order constructs. It was

shown in [98] that a purely propositional formulation of the Böhm-Jacopini theorem is false: there is a three-state deterministic propositional flowchart that is not equivalent to any propositional while program. As observed by a number of authors (e.g. [86, 134]), while loops with multi-level breaks are sufficient to represent all deterministic flowcharts without introducing auxiliary variables, and [86] established a strict hierarchy based on the allowed levels of the multi-level breaks. That result was reformulated and proved at the propositional level in [98].

The notions of functions on a domain, variables ranging over that domain, and variable assignment are inherent in first-order logic, but are absent at the propositional level. Moreover, many arguments rely on combinatorial graph restructuring operations, which are difficult to formalize. Thus the value of the propositional view is twofold: it operates at a higher level of abstraction and brings topological and coalgebraic concepts and techniques to bear.

Propositional while programs and their encoding in terms of the regular operators goes back to early work on Propositional Dynamic Logic [41]. GKAT as an independent system and its semantics were introduced in [93, 98] under the name *propositional while programs*, although the succinct form of the program operators is new here. Also introduced in [93, 98] were the functor G and automaton model (Section 5.4), the determinacy property (Definition 5.2.2) (called *strict determinacy* there), and the concept of normality (Section 5.5.1) (called *liveness* there). The linear construction of an automaton from a while program was sketched in [93, 98], based on earlier results for KAT automata [92], but the complexity of deciding equivalence was not addressed. The more rigorous alternative construction given here (Section 5.4.2) is needed to establish simplicity, thereby enabling our Kleene theorem. The existence of a complete axiomatization was not considered in [93, 98].

Guarded strings, which form the basis of our language semantics, were introduced

in [76].

The axiomatization we propose for GKAT is closely related to Salomaa’s axiomatization of regular expressions based on unique fixed points [152] and to Silva’s coalgebraic generalization of Kleene algebra [162]. The proof technique we used for completeness is inspired by the general proof in [162].

The relational semantics is inspired by that for KAT [96], which goes back to work on Dynamic Logic [97]. Because the fixpoint axiom uses a non-algebraic side condition to guarantee soundness, extra care is needed to define the relational interpretation for GKAT.

5.8 Conclusions and Future Directions

We have presented a comprehensive algebraic and coalgebraic study of GKAT, an abstract programming language with uninterpreted actions. Our main contributions include: (i) a new automata construction yielding a nearly linear time decidability result for program equivalence; (ii) a Kleene theorem for GKAT providing an exact correspondence between programs and a well-defined class of automata; and (iii) a set of sound and complete axioms for program equivalence.

We hope this paper is only the beginning of a long and beautiful journey into understanding the (co)algebraic properties of efficient fragments of imperative programming languages. We briefly discuss some limitations of our current development and our vision for future work.

As in Salomaa’s axiomatization of KA, our axiomatization is not fully algebraic: (W3) is only sensible for the language model. As a result, the current completeness proof does not generalize to other natural models of interest—e.g., probabilistic or relational. To overcome this limitation, we would like to extend Kozen’s axiomatization of KA to GKAT by developing a natural order for GKAT programs. In the case of KA we have

$e \leq f \iff e + f = f$, but this natural order is no longer definable in the absence of $+$ and so we need to axiomatize $e \leq f$ for GKAT programs directly. This appears to be the main missing piece to obtain an algebraic axiomatization.

Various extensions of KAT to reason about richer programs (KAT+B!, NetKAT, Prob-NetKAT) have been proposed, and it is natural to ask whether extending GKAT in similar directions will yield interesting algebraic theories and decision procedures for domain-specific applications. For instance, GKAT may be better suited for probabilistic models, as it avoids mixing non-determinism and probabilities. Such models could facilitate reasoning about probabilistic programs, whose complex semantics would make a framework for equational reasoning especially valuable.

In a different direction, a language model containing infinite traces could be interesting in many applications, as it could serve as a model to reason about non-terminating programs—e.g., loops in NetKAT in which packets may be forwarded forever. An interesting open question is whether the infinite language model can be finitely axiomatized.

Finally, another direction would be to extend the GKAT decision procedure to handle extra equations. For instance, both KAT+B! and NetKAT have independently-developed decision procedures, that are similar in flavor, which raises the question of whether the GKAT decision procedure could be extended in a more generic way, similar to the Nelson-Oppen approach [119] for combining decision procedures used in SMT solving.

Part IV

Conclusion

Chapter 6

Conclusion

Computer networks have reached a size and complexity that makes configuring them manually, through low-level interfaces, increasingly challenging and error-prone. This dissertation proposed two instruments to address this: domain-specific programming languages, to reduce the semantic gap between intent and configuration by providing natural abstractions, thereby shifting much of the configuration burden to the compiler; and automated verification tools, to predict network behavior reliably and to rule out bugs before they manifest.

6.1 Thoughts on Practical Impact

The area of software-defined networking has seen exciting academic progress over the last decade, going far beyond the contributions in this dissertation. Industry is showing unusual eagerness to adapt these ideas quickly—*e.g.*, network verification tools have been deployed at Microsoft and Amazon—confirming that there is a practical need for novel approaches to network configuration and management. The transfer of ideas from academia to industry is further catalyzed by several startups [67, 121, 123, 124] in the area.

Why is it that the means of configuring networks have remained essentially un-

changed for decades, but suddenly experience dramatic change? We speculate that the following are two key driving forces:

- **Cloud:** With the shift to cloud computing, there has also been a shift from mostly-static to ever-evolving networks. In other words, the lifetime of network configurations has dramatically decreased. There is a renewed need for reconfiguring networks rapidly, reliably, and inexpensively.
- **Scale:** Many networks, such as data-center networks, have seen rapid growth in terms of size and complexity: they send more packets, at faster speeds, between more machines. This increases the potential for and cost of network misconfigurations. Additionally, traditional approaches to network management are highly manual, but manual reasoning scales poorly beyond a certain complexity.

This is an exciting time for academia to offer novel solutions to network management, and lay a solid foundation upon which future growth in networking can flourish. This dissertation hopes to make a contribution in this direction.

6.2 Future Directions

The development of NetKAT has been an educational case study that suggests some promising directions for future work on the metatheory of programming language design. Chapter 5 can be seen as a first step in this direction.

NetKAT was designed by taking Kleene algebra with Tests (KAT) as a (co)algebraic foundation, and instantiating it with network primitives to obtain a network programming language [6]. This approach is technically attractive: NetKAT inherited KAT's rich mathematical foundation—a compositional semantics, a sound and complete algebraic axiomatization, a coalgebraic automaton model—with comparatively little adaptive effort; and this foundation was the crucial enabler in the development of appealing

applications—*e.g.*, the NetKAT decision procedure [45] and compiler (Chapter 2).

A General Framework. We believe the NetKAT approach to language design is compelling. A natural question is therefore if it could be replicated for other domains, ideally without having to manually adapt KAT’s metatheory each time. We envision a (co)algebraic framework that yields a KAT, or GKAT, (together with a language model, automata model, and sound and complete axiomatization) for a given set of primitives—maybe specified through equations—and is general enough to subsume NetKAT.

On the algorithmic side, such a framework would yield a decision procedure, maybe by combining a generic (G)KAT decision procedure with a user-provided procedure for handling additional equations—specific to the (G)KAT in question—similar to the Nelson-Oppen approach [119] that is used in SMT. Beckett et al. [13] have pursued similar ideas, but their framework imposes a trace semantics. This semantics is too fine for many application of interest: for example, it is not suitable for NetKAT.

Because GKAT has a faster decision procedure and requires less structure (no $+$) than KAT, it may be better suited for certain (*e.g.*, probabilistic) domains and may form an attractive basis for the (co)algebraic framework we envision.

Extensibility. More generally, we lack metatheoretic tools that would help us decide if and how a (co)algebraic system can be extended with additional primitives. For example, extending NetKAT with probabilistic choice proved challenging, and resulted in a system that is—despite its name, ProbNetKAT—not even a KAT. A framework as sketched above would shed some light on this question of extensibility, by showing that a certain class of extensions is always possible.

Probabilistic GKAT. Probabilistic programming languages are currently receiving renewed attention due to their use in machine learning. An intriguing direction for future work is to extend GKAT with probabilistic choice to model such languages. The resulting

system may be used, for example, to reason about program transformations performed by compilers or used in cryptographic proofs.

Bibliography

- [1] Samson Abramsky and Achim Jung. 1994. Domain Theory. In *Handbook of Logic in Computer Science*, S. Abramsky, Dov M. Gabbay, and T.S.E. Maibaum (Eds.). Vol. 3. Clarendon Press, 1–168.
- [2] S. B. Akers. 1978. Binary Decision Diagrams. *IEEE Trans. Comput.* 27, 6 (June 1978), 509–516. <https://doi.org/10.1109/TC.1978.1675141>
- [3] Mohammad Al-Fares, Alex Loukissas, and Amin Vahdat. 2008. A Scalable, Commodity, Data Center Network Architecture. In *SIGCOMM*.
- [4] Mohammad Al-Fares, Sivasankar Radhakrishnan, Barath Raghavan, Nelson Huang, and Amin Vahdat. 2010. Hedera: Dynamic Flow Scheduling for Data Center Networks. In *NSDI*. 19–19.
- [5] Ali Al-Shabibi, Marc De Leenheer, Matteo Gerola, Ayaka Koshibe, Guru Parulkar, Elio Salvadori, and Bill Snow. 2014. OpenVirteX: Make Your Virtual SDNs Programmable. In *HotSDN*.
- [6] Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker. 2014. NetKAT: Semantic Foundations for Networks. In *POPL*. 113–126. <https://doi.org/10.1145/2535838.2535862>
- [7] Allegra Angus and Dexter Kozen. 2001. *Kleene Algebra with Tests and Program*

Schematology. Technical Report TR2001-1844. Computer Science Department, Cornell University.

- [8] Valentin Antimirov. 1996. Partial Derivatives of Regular Expressions and Finite Automaton Constructions. *Theoretical Computer Science* 155, 2 (1996), 291–319.
- [9] David Applegate and Edith Cohen. 2003. Making intra-domain routing robust to changing and uncertain traffic demands: understanding fundamental tradeoffs. In *SIGCOMM*. 313–324. <https://doi.org/10.1145/863955.863991>
- [10] E. Ashcroft and Z. Manna. 1972. The translation of GOTO programs into WHILE programs. In *Proceedings of IFIP Congress 71*, C.V. Freiman, J.E. Griffith, and J.L. Rosenfeld (Eds.), Vol. 1. North-Holland, Amsterdam, 250–255.
- [11] Roland Backhouse. 1975. *Closure algorithms and the star-height problem of regular languages*. Ph.D. Dissertation. University of London. <http://ethos.bl.uk/OrderDetails.do?uin=uk.bl.ethos.448525>
- [12] Adam Barth and Dexter Kozen. 2002. *Equational Verification of Cache Blocking in LU Decomposition using Kleene Algebra with Tests*. Technical Report TR2002-1865. Computer Science Department, Cornell University.
- [13] Ryan Beckett, Eric Campbell, and Michael Greenberg. 2017. Kleene Algebra Modulo Theories. *CoRR* abs/1707.02894 (2017). arXiv:1707.02894 <http://arxiv.org/abs/1707.02894>
- [14] Ryan Beckett, Ratul Mahajan, Todd D. Millstein, Jitendra Padhye, and David Walker. 2016. Don't Mind the Gap: Bridging Network-wide Objectives and Device-level Configurations. In *SIGCOMM*. 328–341. <https://doi.org/10.1145/2934872.2934909>

- [15] Manav Bhatia, Mach Chen, Sami Boutros, Marc Binderberger, and Jeffrey Haas. 2014. Bidirectional Forwarding Detection (BFD) on Link Aggregation Group (LAG) Interfaces. RFC 7130. <https://doi.org/10.17487/RFC7130>
- [16] Garrett Birkhoff and Thomas C. Bartee. 1970. *Modern applied algebra*. McGraw-Hill, New York.
- [17] C. Böhm and G. Jacopini. 1966. Flow Diagrams, Turing Machines and Languages with only Two Formation Rules. *Commun. ACM* (May 1966), 366–371. <https://doi.org/10.1145/355592.365646>
- [18] Filippo Bonchi and Damien Pous. 2013. Checking NFA equivalence with bisimulations up to congruence. In *Proc. Principles of Programming Languages (POPL)*. ACM, New York, 457–468. <https://doi.org/10.1145/2429069.2429124>
- [19] Johannes Borgström, Andrew D. Gordon, Michael Greenberg, James Margetson, and Jurgen Van Gael. 2011. Measure Transformer Semantics for Bayesian Machine Learning. In *ESOP*. https://doi.org/10.1007/978-3-642-19718-5_5
- [20] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. 2014. P4: Programming Protocol-Independent Packet Processors. *SIGCOMM CCR* 44, 3 (July 2014), 87–95.
- [21] Randal E. Bryant. 1986. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Trans. Comput.* 35, 8 (August 1986), 677–691. <https://doi.org/10.1109/TC.1986.1676819>
- [22] Martin Casado, Nate Foster, and Arjun Guha. 2014. Abstractions for Software-Defined Networks. *CACM* 57, 10 (October 2014), 86–95.

- [23] Martin Casado, Teemu Koponen, Rajiv Ramanathan, and Scott Shenker. 2010. Virtualizing the Network Forwarding Plane. In *PRESTO*.
- [24] Ernie Cohen. 1994. Lazy Caching in Kleene Algebra. <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.57.5074>
- [25] Ernie Cohen. 1994. *Using Kleene algebra to reason about concurrency control*. Technical Report. Telcordia, Morristown, NJ.
- [26] Ernie Cohen, Dexter Kozen, and Frederick Smith. 1996. *The complexity of Kleene algebra with tests*. Technical Report TR96-1598. Computer Science Department, Cornell University.
- [27] John Horton Conway. 1971. *Regular Algebra and Finite Machines*. Chapman and Hall, London.
- [28] R. Cruz. 1991. A calculus for network delay, parts I and II. *IEEE Transactions on Information Theory* 37, 1 (January 1991), 114–141. <https://doi.org/10.1109/18.61110>
- [29] Emilie Danna, Subhasree Mandal, and Arjun Singh. 2012. A practical algorithm for balancing the max-min fairness and throughput objectives in traffic engineering. In *INFOCOM*. 846–854. <https://doi.org/10.1109/INFOCOM.2012.6195833>
- [30] Timothy A. Davis. 2004. Algorithm 832: UMFPACK V4.3—an Unsymmetric-pattern Multifrontal Method. *ACM Trans. Math. Softw.* 30, 2 (June 2004), 196–199. <https://doi.org/10.1145/992200.992206>
- [31] Alessandra Di Pierro, Chris Hankin, and Herbert Wiklicky. 2005. Probabilistic λ -calculus and quantitative program analysis. *Journal of Logic and Computation* 15, 2 (2005), 159–179. <https://doi.org/10.1093/logcom/exi008>

- [32] Roger Dingledine, Nick Mathewson, and Paul Syverson. 2004. Tor: The Second-generation Onion Router. In *USENIX Security Symposium (SSYM)*. 21–21.
- [33] A. Dixit, P. Prakash, Y. C. Hu, and R. R. Kompella. 2013. On the impact of packet spraying in data center networks. In *IEEE INFOCOM*. 2130–2138.
- [34] Ernst-Erich Doberkat. 2007. *Stochastic Relations: Foundations for Markov Transition Systems*. Chapman Hall. <https://doi.org/10.1201/9781584889427>
- [35] Rick Durrett. 2010. *Probability: Theory and Examples*. Cambridge University Press. <https://doi.org/10.1017/CB09780511779398>
- [36] Abbas Edalat. 1994. Domain theory and integration. In *LICS*. 115–124.
- [37] Abbas Edalat. 1996. The Scott topology induces the weak topology. In *LICS*. 372–381.
- [38] Abbas Edalat and Reinhold Heckmann. 1998. A computational model for metric spaces. *Theoretical Computer Science* 193, 1 (1998), 53–73. [https://doi.org/10.1016/S0304-3975\(96\)00243-5](https://doi.org/10.1016/S0304-3975(96)00243-5)
- [39] Andrew D. Ferguson, Arjun Guha, Chen Liang, Rodrigo Fonseca, and Shriram Krishnamurthi. 2012. Hierarchical Policies for Software Defined Networks. In *HotSDN*.
- [40] Andrew D. Ferguson, Arjun Guha, Chen Liang, Rodrigo Fonseca, and Shriram Krishnamurthi. 2013. Participatory Networking: An API for Application Control of SDNs. In *SIGCOMM*.
- [41] Michael J. Fischer and Richard E. Ladner. 1979. Propositional dynamic logic of regular programs. *J. Comput. Syst. Sci.* 18, 2 (1979), 194–211.

- [42] B. Fortz, J. Rexford, and M. Thorup. 2002. Traffic Engineering with Traditional IP Routing Protocols. *IEEE Communications Magazine* 40, 10 (October 2002), 118–124. <https://doi.org/10.1109/MCOM.2002.1039866>
- [43] Nate Foster, Rob Harrison, Michael J. Freedman, Christopher Monsanto, Jennifer Rexford, Alec Story, and David Walker. 2011. Frenetic: A Network Programming Language. In *ICFP*. 279–291. <https://doi.org/10.1145/2034773.2034812>
- [44] Nate Foster, Dexter Kozen, Konstantinos Mamouras, Mark Reitblatt, and Alexandra Silva. 2016. Probabilistic NetKAT. In *ESOP*. 282–309. https://doi.org/10.1007/978-3-662-49498-1_12
- [45] Nate Foster, Dexter Kozen, Matthew Milano, Alexandra Silva, and Laure Thompson. 2015. A Coalgebraic Decision Procedure for NetKAT. In *POPL*. ACM, 343–355. <https://doi.org/10.1145/2775051.2677011>
- [46] M. Fujita, P. C. McGeer, and J. C.-Y. Yang. 1997. Multi-Terminal Binary Decision Diagrams: An Efficient DataStructure for Matrix Representation. *Form. Methods Syst. Des.* 10, 2-3 (April 1997), 149–169. <https://doi.org/10.1023/A:1008647823331>
- [47] Stephen J. Garland and David C. Luckham. 1973. Program schemes, recursion schemes, and formal languages. *J. Comput. System Sci.* 7, 2 (1973), 119 – 160. [https://doi.org/10.1016/S0022-0000\(73\)80040-6](https://doi.org/10.1016/S0022-0000(73)80040-6)
- [48] Timon Gehr, Sasa Misailovic, Petar Tsankov, Laurent Vanbever, Pascal Wiesmann, and Martin T. Vechev. 2018. Bayonet: Probabilistic Computer Network Analysis. Available at <https://github.com/eth-sri/bayonet/>.
- [49] Timon Gehr, Sasa Misailovic, Petar Tsankov, Laurent Vanbever, Pascal Wiesmann,

- and Martin T. Vechev. 2018. Bayonet: probabilistic inference for networks. In *ACM SIGPLAN PLDI*. 586–602.
- [50] Timon Gehr, Sasa Misailovic, and Martin T. Vechev. 2016. PSI: Exact Symbolic Inference for Probabilistic Programs. 62–83.
- [51] Phillipa Gill, Navendu Jain, and Nachiappan Nagappan. 2011. Understanding Network Failures in Data Centers: Measurement, Analysis, and Implications. In *SIGCOMM*. 350–361.
- [52] Michele Giry. 1982. A categorical approach to probability theory. In *Categorical aspects of topology and analysis*. Springer, 68–85. <https://doi.org/10.1007/BFb0092872>
- [53] Andrew D. Gordon, Thomas A. Henzinger, Aditya V. Nori, and Sriram K. Rajamani. 2014. Probabilistic Programming. In *FOSE*. <https://doi.org/10.1145/2593882.2593900>
- [54] Andrew D Gordon, Thomas A Henzinger, Aditya V Nori, and Sriram K Rajamani. 2014. Probabilistic programming. In *Proceedings of the on Future of Software Engineering*. ACM, 167–181. <https://doi.org/10.1145/2593882.2593900>
- [55] S. Graham. 1988. Closure properties of a probabilistic powerdomain construction. In *MFPS*. 213–233. https://doi.org/10.1007/3-540-19020-1_11
- [56] Friedrich Gretz, Nils Jansen, Benjamin Lucien Kaminski, Joost-Pieter Katoen, Annabelle McIver, and Federico Olmedo. 2015. Conditioning in Probabilistic Programming. *CoRR* abs/1504.00198 (2015).
- [57] Arjun Guha, Mark Reitblatt, and Nate Foster. 2013. Machine-Verified Network Controllers. In *PLDI*.

- [58] Arpit Gupta, Laurent Vanbever, Muhammad Shahbaz, Sean Donovan, Brandon Schlinker, Nick Feamster, Jennifer Rexford, Scott Shenker, Russ Clark, and Ethan Katz-Bassett. 2014. SDX: A Software Defined Internet Exchange. In *SIGCOMM*.
- [59] Stephen Gutz, Alec Story, Cole Schlesinger, and Nate Foster. 2012. Splendid Isolation: A Slice Abstraction for Software-Defined Networks. In *HotSDN*.
- [60] P. R. Halmos. 1950. *Measure Theory*. Van Nostrand. <https://doi.org/10.1007/978-1-4684-9440-2>
- [61] Jiayue He and Jennifer Rexford. 2008. Toward internet-wide multipath routing. *IEEE Network Magazine* 22, 2 (2008), 16–21. <https://doi.org/10.1109/MNET.2008.4476066>
- [62] R. Heckmann. 1994. Probabilistic power domains, information systems, and locales. In *MFPS*, Vol. 802. 410–437. https://doi.org/10.1007/3-540-58027-1_20
- [63] Laurie J. Hendren, C. Donawa, Maryam Emami, Guang R. Gao, Justiani, and B. Sridharan. 1992. Designing the McCAT Compiler Based on a Family of Structured Intermediate Representations. In *LCPC '15: Proceedings of the 5th International Workshop on Languages and Compilers for Parallel Computing*. Springer, Berlin, Heidelberg, 406–420.
- [64] Chi-Yao Hong, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Vijay Gill, Mohan Nanduri, and Roger Wattenhofer. 2013. Achieving high utilization with software-driven WAN. In *SIGCOMM*. 15–26. <https://doi.org/10.1145/2534169.2486012>
- [65] John E. Hopcroft and Richard M. Karp. 1971. *A linear algorithm for testing equivalence of finite automata*. Technical Report TR 71-114. Cornell University.

- [66] I. Ianov. 1960. The Logical Schemes of Algorithms. *Problems of Cybernetics* (1960), 82–140.
- [67] Intentionet. 2019. Intentionet. Retrieved September 6, 2019 from <https://www.intentionet.com>
- [68] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, et al. 2013. B4: Experience with a globally-deployed software defined WAN. In *SIGCOMM*. 3–14. <https://doi.org/10.1145/2534169.2486019>
- [69] Vimalkumar Jeyakumar, Mohammad Alizadeh, David Mazières, Balaji Prabhakar, Albert Greenberg, and Changhoon Kim. 2013. EyeQ: Practical network performance isolation at the edge. In *NSDI*. 297–311.
- [70] Xin Jin, Jennifer Gossels, Jennifer Rexford, and David Walker. 2015. CoVisor: A Compositional Hypervisor for Software-Defined Networks. In *NSDI*.
- [71] Claire Jones. 1989. *Probabilistic Non-determinism*. Ph.D. Dissertation. University of Edinburgh.
- [72] C. Jones and G. Plotkin. 1989. A probabilistic powerdomain of evaluations. In *LICS*. 186–195.
- [73] Lavanya Jose, Lisa Yan, George Varghese, and Nick McKeown. 2015. Compiling Packet Programs to Reconfigurable Switches. In *NSDI*.
- [74] Achim Jung and Regina Tix. 1998. The Troublesome Probabilistic Powerdomain. *ENTCS* 13 (1998), 70–91. [https://doi.org/10.1016/S1571-0661\(05\)80216-6](https://doi.org/10.1016/S1571-0661(05)80216-6)
- [75] Srikanth Kandula, Dina Katabi, Bruce Davie, and Anna Charny. 2005. Walking

the tightrope: Responsive yet stable traffic engineering. In *SIGCOMM*. 253–264.
<https://doi.org/10.1145/1090191.1080122>

- [76] Donald M. Kaplan. 1969. Regular Expressions and the Equivalence of Programs. *J. Comput. Syst. Sci.* 3 (1969), 361–386.
- [77] Peyman Kazemian, George Varghese, and Nick McKeown. 2012. Header Space Analysis: Static Checking for Networks. In *NSDI*.
- [78] John G Kemeny and James Laurie Snell. 1960. *Finite markov chains*. Vol. 356. van Nostrand Princeton, NJ.
- [79] Ahmed Khurshid, Xuan Zou, Wenxuan Zhou, Matthew Caesar, and P. Brighten Godfrey. 2013. VeriFlow: Verifying Network-Wide Invariants in Real Time. In *NSDI*. <https://doi.org/10.1145/2342441.2342452>
- [80] Changhoon Kim, Anirudh Sivaraman, Naga Katta, Antonin Bas, Advait Dixit, and Lawrence J Wobker. 2015. In-band network telemetry via programmable dataplanes. In *SIGCOMM*.
- [81] Stephen C. Kleene. 1956. Representation of Events in Nerve Nets and Finite Automata. *Automata Studies* (1956), 3–41.
- [82] Simon Knight, Hung X. Nguyen, Nickolas Falkner, Rhys Bowden, and Matthew Roughan. 2011. The Internet Topology Zoo. *IEEE Journal on Selected Areas in Communications* (2011).
- [83] Murali Kodialam, TV Lakshman, James B Orlin, and Sudipta Sengupta. 2009. Oblivious routing of highly variable traffic in service overlays and IP backbones. *IEEE/ACM Transactions on Networking (TON)* 17, 2 (2009), 459–472. <https://doi.org/10.1109/TNET.2008.927257>

- [84] A. N. Kolmogorov and S. V. Fomin. 1970. *Introductory Real Analysis*. Prentice Hall.
- [85] Teemu Koponen, Keith Amidon, Peter Balland, Martín Casado, Anupam Chanda, Bryan Fulton, Jesse Gross Igor Ganichev, Natasha Gude, Paul Ingram, Ethan Jackson, Andrew Lambeth, Romain Lenglet, Shih-Hao Li, Amar Padmanabhan, Justin Pettit, Ben Pfaff, , Rajiv Ramanathan, Scott Shenker, Alan Shieh, Jeremy Stribling, Pankaj Thakkar, Dan Wendlandt, Alexander Yip, and Ronghua Zhang. 2014. Network Virtualization in Multi-tenant Datacenters. In *NSDI*.
- [86] S. Rao Kosaraju. 1973. Analysis of structured programs. In *Proc. 5th ACM Symp. Theory of Computing (STOC'73)*. ACM, New York, NY, USA, 240–252. <https://doi.org/10.1145/800125.804055>
- [87] Dexter Kozen. 1981. Semantics of probabilistic programs. *J. Comput. Syst. Sci.* 22 (1981), 328–350.
- [88] Dexter Kozen. 1985. A probabilistic PDL. *J. Comput. Syst. Sci.* 30, 2 (April 1985), 162–178.
- [89] Dexter Kozen. 1996. Kleene algebra with tests and commutativity conditions. In *Proc. Second Int. Workshop Tools and Algorithms for the Construction and Analysis of Systems (TACAS'96) (Lecture Notes in Computer Science)*, T. Margaria and B. Steffen (Eds.), Vol. 1055. Springer-Verlag, Passau, Germany, 14–33.
- [90] Dexter Kozen. 1997. Kleene algebra with tests. *ACM TOPLAS* 19, 3 (May 1997), 427–443. <https://doi.org/10.1145/256167.256195>
- [91] Dexter Kozen. 1997. Kleene algebra with tests. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 19, 3 (May 1997), 427–443. <https://doi.org/10.1145/256167.256195>

- [92] Dexter Kozen. 2003. Automata on Guarded Strings and Applications. *Matématica Contemporânea* 24 (2003), 117–139.
- [93] Dexter Kozen. 2008. Nonlocal Flow of Control and Kleene Algebra with Tests. In *Proc. 23rd IEEE Symp. Logic in Computer Science (LICS'08)* (Pittsburgh). 105–117.
- [94] Dexter Kozen, Radu Mardare, and Prakash Panangaden. 2013. Strong Completeness for Markovian Logics. In *MFCS*. 655–666. https://doi.org/10.1007/978-3-642-40313-2_58
- [95] Dexter Kozen and Maria-Cristina Patron. 2000. Certification of compiler optimizations using Kleene algebra with tests. In *Proc. 1st Int. Conf. Computational Logic (CL2000)* (London) (*Lecture Notes in Artificial Intelligence*), John Lloyd, Veronica Dahl, Ulrich Furbach, Manfred Kerber, Kung-Kiu Lau, Catuscia Palamidessi, Luis Moniz Pereira, Yehoshua Sagiv, and Peter J. Stuckey (Eds.), Vol. 1861. Springer-Verlag, London, 568–582.
- [96] Dexter Kozen and Frederick Smith. 1996. Kleene algebra with tests: Completeness and decidability. In *Proc. 10th Int. Workshop Computer Science Logic (CSL'96)* (*Lecture Notes in Computer Science*), D. van Dalen and M. Bezem (Eds.), Vol. 1258. Springer-Verlag, Utrecht, The Netherlands, 244–259.
- [97] Dexter Kozen and Jerzy Tiuryn. 1990. Logics of Programs. In *Handbook of Theoretical Computer Science*, J. van Leeuwen (Ed.). Vol. B. North Holland, Amsterdam, 789–840.
- [98] Dexter Kozen and Wei-Lung (Dustin) Tseng. 2008. The Böhm-Jacopini Theorem is False, Propositionally. In *Proc. 9th Int. Conf. Mathematics of Program Construction (MPC'08)* (*Lecture Notes in Computer Science*), P. Audebaud and C. Paulin-Mohring (Eds.), Vol. 5133. Springer, 177–192.

- [99] Praveen Kumar, Yang Yuan, Chris Yu, Nate Foster, Robert Kleinberg, Petr Lapukhov, Chiun Lin Lim, and Robert Soulé. 2018. Semi-Oblivious Traffic Engineering: The Road Not Taken. In *USENIX NSDI*.
- [100] M. Kwiatkowska, G. Norman, and D. Parker. 2011. PRISM 4.0: Verification of Probabilistic Real-time Systems. In *Proc. 23rd International Conference on Computer Aided Verification (CAV'11) (LNCS)*, G. Gopalakrishnan and S. Qadeer (Eds.), Vol. 6806. Springer, 585–591. https://doi.org/10.1007/978-3-642-22110-1_47
- [101] Marta Z. Kwiatkowska, Gethin Norman, and David Parker. 2011. PRISM 4.0: Verification of Probabilistic Real-Time Systems. In *CAV*. 585–591. https://doi.org/10.1007/978-3-642-22110-1_47
- [102] Kim G. Larsen, Radu Mardare, and Prakash Panangaden. 2012. Taking it to the limit: Approximate reasoning for Markov Processes. In *MFCS*. https://doi.org/10.1007/978-3-642-32589-2_59
- [103] Alex X. Liu, Fei Chen, JeeHyun Hwang, and Tao Xie. 2008. XEngine: A Fast and Scalable XACML Policy Evaluation Engine. In *International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*.
- [104] Alex X. Liu, Chad R. Meiners, and Eric Torng. 2010. TCAM Razor: A systematic approach towards minimizing packet classifiers in TCAMs. *TON* 18, 2 (April 2010), 490–500.
- [105] Jed Liu, William Hallahan, Cole Schlesinger, Milad Sharif, Jeongkeun Lee, Robert Soulé, Han Wang, Calin Cascaval, Nick McKeown, and Nate Foster. 2018. p4v: Practical Verification for Programmable Data Planes. In *SIGCOMM*. 490–503.

- [106] Vincent Liu, Daniel Halperin, Arvind Krishnamurthy, and Thomas E Anderson. 2013. F10: A Fault-Tolerant Engineered Network. In *USENIX NSDI*. 399–412.
- [107] D.C. Luckham, D.M.R. Park, and M.S. Paterson. 1970. On formalised computer programs. *J. Comput. System Sci.* 4, 3 (1970), 220–249. [https://doi.org/10.1016/S0022-0000\(70\)80022-8](https://doi.org/10.1016/S0022-0000(70)80022-8)
- [108] Haohui Mai, Ahmed Khurshid, Rachit Agarwal, Matthew Caesar, P. Brighten Godfrey, and Samuel Talmadge King. 2011. Debugging the Data Plane with Anteater. In *ACM SIGCOMM*. 290–301.
- [109] Radu Mardare, Prakash Panangaden, and Gordon Plotkin. 2016. Quantitative Algebraic Reasoning. In *LICS*. <https://doi.org/10.1145/2933575.2934518>
- [110] Jedidiah McClurg, Hossein Hojjat, Nate Foster, and Pavol Cerny. 2016. Event-Driven Network Programming. In *PLDI*. <https://doi.org/10.1145/2908080.2908097>
- [111] Annabelle McIver and Carroll Morgan. 2004. *Abstraction, Refinement And Proof For Probabilistic Systems*. Springer.
- [112] A. K. McIver, E. Cohen, C. Morgan, and C. Gonzalia. 2008. Using Probabilistic Kleene Algebra pKA for Protocol Verification. *J. Logic and Algebraic Programming* 76, 1 (2008), 90–111. <https://doi.org/10.1016/j.jlap.2007.10.005>
- [113] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. 2008. OpenFlow: Enabling Innovation in Campus Networks. *SIGCOMM CCR* 38, 2 (2008), 69–74.
- [114] M.W. Mislove. 2006. On Combining Probability and Nondeterminism. *Electronic Notes in Theoretical Computer Science* 162 (2006), 261 – 265. <https://doi.org/10.1016/j.entcs.2006.08.011>

- [org/10.1016/j.entcs.2005.12.113](https://doi.org/10.1016/j.entcs.2005.12.113) Proceedings of the Workshop: Essays on Algebraic Process Calculi (APC 25).
- [115] Christopher Monsanto, Nate Foster, Rob Harrison, and David Walker. 2012. A Compiler and Run-time System for Network Programming Languages. In *POPL*.
- [116] Christopher Monsanto, Joshua Reich, Nate Foster, Jennifer Rexford, and David Walker. 2013. Composing Software-Defined Networks. In *NSDI*.
- [117] Carroll Morgan, Annabelle McIver, and Karen Seidel. 1996. Probabilistic Predicate Transformers. *ACM TOPLAS* 18, 3 (May 1996), 325–353. <https://doi.org/10.1145/229542.229547>
- [118] Paul H. Morris, Ronald A. Gray, and Robert E. Filman. 1997. GOTO Removal Based on Regular Expressions. *J. Software Maintenance: Research and Practice* 9, 1 (1997), 47–66. [https://doi.org/DOI:10.1002/\(SICI\)1096-908X\(199701\)9:1<47::AID-SMR142>3.0.CO;2-V](https://doi.org/DOI:10.1002/(SICI)1096-908X(199701)9:1<47::AID-SMR142>3.0.CO;2-V)
- [119] Greg Nelson and Derek C. Oppen. 1979. Simplification by Cooperating Decision Procedures. *ACM Trans. Program. Lang. Syst.* 1, 2 (1979), 245–257. <https://doi.org/10.1145/357073.357079>
- [120] Tim Nelson, Andrew D. Ferguson, Michael J. G. Scheer, and Shriram Krishnamurthi. 2014. Tierless Programming and Reasoning for Software-Defined Networks. In *NSDI*.
- [121] Barefoot Networks. 2019. Barefoot Networks. Retrieved September 6, 2019 from <https://barefootnetworks.com/>
- [122] Barefoot Networks. 2019. Tofino - World’s fastest P4-programmable Ethernet switch ASICs. Retrieved August 20, 2019 from <https://www.barefootnetworks.com/products/brief-tofino/>

- [123] Forward Networks. 2019. Forward Networks. Retrieved September 6, 2019 from <https://www.forwardnetworks.com/>
- [124] Forward Networks. 2019. Veriflow. Retrieved September 6, 2019 from <https://www.veriflow.net/>
- [125] Mogens Nielsen, Gordon D. Plotkin, and Glynn Winskel. 1979. Petri Nets, Event Structures and Domains. In *Semantics of Concurrent Computation*. 266–284. <https://doi.org/10.1007/BFb0022474>
- [126] Maciej Olejnik, Herbert Wiklicky, and Mahdi Cheraghchi. 2016. Probabilistic Programming and Discrete Time Markov Chains. <http://www.imperial.ac.uk/media/imperial-college/faculty-of-engineering/computing/public/MaciejOlejnik.pdf>
- [127] G. Oulsnam. 1982. Unraveling unstructured programs. *Comput. J.* 25, 3 (1982), 379–387.
- [128] Prakash Panangaden. 1998. Probabilistic Relations. In *PROBMIV*. 59–74.
- [129] Prakash Panangaden. 2009. *Labelled Markov Processes*. Imperial College Press. <https://doi.org/10.1142/9781848162891>
- [130] Sungwoo Park, Frank Pfenning, and Sebastian Thrun. 2008. A Probabilistic Language Based on Sampling Functions. *ACM TOPLAS* 31, 1, Article 4 (December 2008), 46 pages. <https://doi.org/10.1145/1452044.1452048>
- [131] M.S. Paterson and C.E. Hewitt. 1970. Comparative schematology. In *Record of Project MAC Conference on Concurrent Systems and Parallel Computation*. ACM, New York, 119–127.
- [132] A. Paz. 1971. *Introduction to Probabilistic Automata*. Academic Press.

- [133] Jonathan Perry, Amy Ousterhout, Hari Balakrishnan, Devavrat Shah, and Hans Fugal. 2014. Fastpass: A Centralized “Zero-Queue” Datacenter Network. In *SIGCOMM*. <https://doi.org/10.1145/2619239.2626309>
- [134] W. Peterson, T. Kasami, and N. Tokura. 1973. On the Capabilities of while, repeat, and exit Statements. *Comm. Assoc. Comput. Mach.* 16, 8 (1973), 503–512.
- [135] G. D. Plotkin. 1982. Probabilistic powerdomains. In *CAAP*. 271–287.
- [136] Gordon D. Plotkin, Nikolaj Bjørner, Nuno P. Lopes, Andrey Rybalchenko, and George Varghese. 2016. Scaling network verification using symmetry and surgery. In *POPL*. 69–83. <https://doi.org/10.1145/2837614.2837657>
- [137] Damien Pous. 2015. Symbolic Algorithms for Language Equivalence and Kleene Algebra with Tests. In *POPL*.
- [138] Damien Pous. 2015. Symbolic Algorithms for Language Equivalence and Kleene Algebra with Tests. In *Proc. Principles of Programming Languages (POPL)*. ACM, New York, 357–368. <https://doi.org/10.1145/2676726.2677007>
- [139] ONOS Project. 2014. Intent Framework. Available at <http://onos.wpengine.com/wp-content/uploads/2014/11/ONOS-Intent-Framework.pdf>.
- [140] Open Daylight Project. 2014. Group Policy. Available at https://wiki.opendaylight.org/view/Group_Policy:Main.
- [141] Harald Räcke. 2008. Optimal hierarchical decompositions for congestion minimization in networks. In *STOC*. 255–264.
- [142] Norman Ramsey and Avi Pfeffer. 2002. Stochastic lambda calculus and monads of probability distributions. In *POPL*. 154–165. <https://doi.org/10.1145/565816.503288>

- [143] L. Ramshaw. 1988. Eliminating goto's while preserving program structure. *Journal of the ACM* 35, 4 (1988), 893–920.
- [144] L. H. Ramshaw. 1979. *Formalizing the Analysis of Algorithms*. Ph.D. Dissertation. Stanford University.
- [145] M. M. Rao. 1987. *Measure Theory and Integration*. Wiley-Interscience.
- [146] Mark Reitblatt, Nate Foster, Jennifer Rexford, Cole Schlesinger, and David Walker. 2012. Abstractions for Network Update. In *SIGCOMM*. 323–334. <https://doi.org/10.1145/2377677.2377748>
- [147] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C. Snoeren. 2015. Inside the Social Network's (Datacenter) Network. In *ACM SIGCOMM*. 123–137.
- [148] Daniel M. Roy. 2011. *Computability, inference and modeling in probabilistic programming*. Ph.D. Dissertation. Massachusetts Institute of Technology.
- [149] Jan J. M. M. Rutten. 2000. Universal coalgebra: a theory of systems. *Theor. Comput. Sci.* 249, 1 (2000), 3–80. [https://doi.org/10.1016/S0304-3975\(00\)00056-6](https://doi.org/10.1016/S0304-3975(00)00056-6)
- [150] N. Saheb-Djahromi. 1978. Probabilistic LCF. In *MFCS*. 442–451. https://doi.org/10.1007/3-540-08921-7_92
- [151] N. Saheb-Djahromi. 1980. CPOs of measures for nondeterminism. *Theoretical Computer Science* 12 (1980), 19–37. [https://doi.org/10.1016/0304-3975\(80\)90003-1](https://doi.org/10.1016/0304-3975(80)90003-1)
- [152] Arto Salomaa. 1966. Two complete axiom systems for the algebra of regular events. *J. Assoc. Comput. Mach.* 13, 1 (January 1966), 158–169.

- [153] Sriram Sankaranarayanan, Aleksandar Chakarov, and Sumit Gulwani. 2013. Static analysis for probabilistic programs: inferring whole program properties from finitely many paths. In *PLDI*. 447–458. <https://doi.org/10.1145/2499370.2462179>
- [154] Cole Schlesinger, Michael Greenberg, and David Walker. 2014. Concurrent Net-Core: From Policies to Pipelines. In *ICFP*.
- [155] D. S. Scott. 1972. Continuous lattices. In *Toposes, Algebraic Geometry and Logic*. Springer, 97–136. <https://doi.org/10.1007/BFb0073967>
- [156] R. Segala. 2006. Probability and nondeterminism in operational models of concurrency. In *CONCUR*. 64–78. https://doi.org/10.1007/11817949_5
- [157] R. Segala and N. A. Lynch. 1995. Probabilistic simulations for probabilistic processes. In *NJC*. 250–273.
- [158] Vyas Sekar, Michael K. Reiter, Walter Willinger, Hui Zhang, Ramana Rao Kompella, and David G. Andersen. 2008. CSAMP: A System for Network-wide Flow Monitoring. In *USENIX NSDI*. 233–246.
- [159] Micha Sharir, Amir Pnueli, and Sergiu Hart. 1984. Verification of probabilistic programs. *SIAM J. Comput.* 13, 2 (1984), 292–314. <https://doi.org/10.1137/0213021>
- [160] J.C. Shepherdson and H.E. Sturgis. 1963. Computability of Recursive Functions. *Journal of the ACM (JACM)* 10, 2 (1963), 217–255. <https://doi.org/10.1145/321160.321170>
- [161] Alan Shieh, Srikanth Kandula, Albert G Greenberg, and Changhoon Kim. 2010. Seawall: Performance Isolation for Cloud Datacenter Networks.. In *HotCloud*.

- [162] Alexandra Silva. 2010. *Kleene Coalgebra*. Ph.D. Dissertation. Radboud University.
- [163] Rachee Singh, Manya Ghobadi, Klaus-Tycho Foerster, Mark Filer, and Phillipa Gill. 2018. RADWAN: Rate Adaptive Wide Area Network. In *ACM SIGCOMM*.
- [164] Steffen Smolka, Spiros Eliopoulos, Nate Foster, and Arjun Guha. 2015. A Fast Compiler for NetKAT. In *ICFP*. <https://doi.org/10.1145/2784731.2784761>
- [165] Steffen Smolka, Praveen Kumar, Nate Foster, Dexter Kozen, and Alexandra Silva. 2017. Cantor Meets Scott: Semantic Foundations for Probabilistic Networks. In *POPL 2017*. <https://doi.org/10.1145/3009837.3009843>
- [166] Martin Suchara, Dahai Xu, Robert Doverspike, David Johnson, and Jennifer Rexford. 2011. Network architecture for joint failure recovery and traffic engineering. *ACM SIGMETRICS* (2011), 97–108. <https://doi.org/10.1145/2007116.2007128>
- [167] Robert Endre Tarjan. 1975. Efficiency of a Good But Not Linear Set Union Algorithm. *J. ACM* 22, 2 (1975), 215–225. <https://doi.org/10.1145/321879.321884>
- [168] David E. Taylor and Jonathan S. Turner. 2007. ClassBench: A Packet Classification Benchmark. *TON* 15 (June 2007), 499–511. Issue 3.
- [169] Ken Thompson. 1968. Regular Expression Search Algorithm. *Commun. ACM* 11, 6 (1968), 419–422. <https://doi.org/10.1145/363347.363387>
- [170] R. Tix, K. Keimel, and G. Plotkin. 2009. Semantic domains for combining probability and nondeterminism. *ENTCS* 222 (2009), 3–99. <https://doi.org/10.1016/j.entcs.2009.01.002>

- [171] L. Valiant. 1982. A Scheme for Fast Parallel Communication. *SIAM J. Comput.* 11, 2 (1982), 350–361.
- [172] Daniele Varacca, Hagen Völzer, and Glynn Winskel. 2006. Probabilistic event structures and domains. *TCS* 358, 2-3 (2006), 173–199. <https://doi.org/10.1016/j.tcs.2006.01.015>
- [173] D. Varacca and G. Winskel. 2006. Distributing probability over non-determinism. *Mathematical Structures in Computer Science* 16, 1 (2006), 87–113. <https://doi.org/10.1017/S0960129505005074>
- [174] Andreas Voellmy, Junchang Wang, Y. Richard Yang, Bryan Ford, and Paul Hudak. 2013. Maple: Simplifying SDN Programming Using Algorithmic Policies. In *SIGCOMM*. <https://doi.org/10.1145/2486001.2486030>
- [175] Di Wang, Jan Hoffmann, and Thomas Reps. 2018. PMAF: An Algebraic Framework for Static Analysis of Probabilistic Programs. In *POPL 2018*. <https://www.cs.cmu.edu/~janh/papers/WangHR17.pdf>
- [176] M. Williams and H. Ossher. 1978. Conversion of unstructured flow diagrams into structured form. *Comput. J.* 21, 2 (1978), 161–167.
- [177] Geoffrey G. Xie, Jibin Zhan, David A. Maltz, Hui Zhang, Albert G. Greenberg, Gísli Hjálmtýsson, and Jennifer Rexford. 2005. On static reachability analysis of IP networks. In *INFOCOM*.
- [178] Hongkun Yang and Simon S. Lam. 2016. Real-time Verification of Network Properties Using Atomic Predicates. *IEEE/ACM ToN* 24, 2 (April 2016), 887–900.
- [179] Zhiyong Zhang, Ovidiu Mara, and Katerina Argyraki. 2014. Network Neutrality Inference. In *ACM SIGCOMM*. 63–74.

- [180] Rui Zhang-Shen and Nick McKeown. 2005. Designing a Predictable Internet Backbone with Valiant Load-Balancing. In *International Workshop on Quality of Service (IWQoS)*. 178–192. https://doi.org/10.1007/11499169_15

Part V

Appendix

Chapter A

Appendix to Chapter 3

A.1 $(\mathcal{M}(2^H), \sqsubseteq)$ is not a Semilattice

Despite the fact that $(\mathcal{M}(2^H), \sqsubseteq)$ is a directed set (Lemma 3.6.2), it is not a semilattice. Here is a counterexample.

Let $b = \{\pi, \sigma, \tau\}$, where π, σ, τ are distinct packets. Let

$$\begin{aligned}\mu_1 &= \frac{1}{2}\delta_{\{\pi\}} + \frac{1}{2}\delta_{\{\sigma\}} & \mu_2 &= \frac{1}{2}\delta_{\{\sigma\}} + \frac{1}{2}\delta_{\{\tau\}} \\ \mu_3 &= \frac{1}{2}\delta_{\{\tau\}} + \frac{1}{2}\delta_{\{\pi\}}.\end{aligned}$$

The measures μ_1, μ_2, μ_3 would be the output measures of the programs $\pi! \oplus \sigma!, \sigma! \oplus \tau!, \tau! \oplus \pi!$, respectively.

We claim that $\mu_1 \sqcap \mu_2$ does not exist. To see this, define

$$\begin{aligned}\nu_1 &= \frac{1}{2}\delta_{\{\tau\}} + \frac{1}{2}\delta_{\{\pi, \sigma\}} & \nu_2 &= \frac{1}{2}\delta_{\{\pi\}} + \frac{1}{2}\delta_{\{\sigma, \tau\}} \\ \nu_3 &= \frac{1}{2}\delta_{\{\sigma\}} + \frac{1}{2}\delta_{\{\tau, \pi\}}.\end{aligned}$$

All ν_i are \sqsubseteq -upper bounds for all μ_j . (In fact, any convex combination $r\nu_1 + s\nu_2 + t\nu_3$ for $0 \leq r, s, t$ and $r + s + t = 1$ is an upper bound for any convex combination $u\mu_1 + v\mu_2 + w\mu_3$ for $0 \leq u, v, w$ and $u + v + w = 1$.) But we show by contradiction that there cannot exist a measure that is both \sqsubseteq -above μ_1 and μ_2 and \sqsubseteq -below ν_1 and ν_2 . Suppose ρ was such a

measure. Since $\rho \sqsubseteq \nu_1$ and $\rho \sqsubseteq \nu_2$, we have

$$\begin{aligned}\rho(B_{\sigma\tau}) &\leq \nu_1(B_{\sigma\tau}) = 0 & \rho(B_{\tau\pi}) &\leq \nu_1(B_{\tau\pi}) = 0 \\ \rho(B_{\pi\sigma}) &\leq \nu_2(B_{\pi\sigma}) = 0.\end{aligned}$$

Since $\mu_1 \sqsubseteq \rho$ and $\mu_2 \sqsubseteq \rho$, we have

$$\begin{aligned}\rho(B_\pi) &\geq \mu_1(B_\pi) = \frac{1}{2} & \rho(B_\sigma) &\geq \mu_1(B_\sigma) = \frac{1}{2} \\ \rho(B_\tau) &\geq \mu_2(B_\tau) = \frac{1}{2}.\end{aligned}$$

But then

$$\begin{aligned}\rho(A_{\pi b}) &= \rho(B_\pi) - \rho(B_{\pi\sigma} \cup B_{\tau\pi}) \geq \frac{1}{2} \\ \rho(A_{\sigma b}) &= \rho(B_\sigma) - \rho(B_{\sigma\tau} \cup B_{\pi\sigma}) \geq \frac{1}{2} \\ \rho(A_{\tau b}) &= \rho(B_\tau) - \rho(B_{\tau\pi} \cup B_{\sigma\tau}) \geq \frac{1}{2},\end{aligned}$$

which is impossible, because ρ would have total weight at least $\frac{3}{2}$.

A.2 Non-Algebraicity

Here is a counterexample to the conjecture that the elements continuous DCPO of continuous kernels is algebraic with finite elements $b \cdot P \cdot d$. Let σ, τ be packets and let $\sigma!$ and $\tau!$ be the programs that set the current packet to σ or τ , respectively. For $r \in [\frac{1}{2}, 1]$, let $P_r = (\sigma! \oplus_r \tau!) \& (\tau! \oplus_r \sigma!)$. On any nonempty input, P_r produces $\{\sigma\}$ with probability $r(1-r)$, $\{\tau\}$ with probability $r(1-r)$, and $\{\sigma, \tau\}$ with probability $r^2 + (1-r)^2$. In particular, P_1 produces $\{\sigma, \tau\}$ with probability 1. The kernels P_r for $1/2 \leq r < 1$ form a directed set whose supremum is P_1 , yet $\{\sigma\} \cdot P_1 \cdot \{\sigma, \tau\}$ is not \sqsubseteq -bounded by any P_r for $r < 1$, therefore the up-closure of $\{\sigma\} \cdot P_1 \cdot \{\sigma, \tau\}$ is not an open set.

A.3 Cantor Meets Scott

This appendix contains proofs omitted from Section 3.5.

Proof of Lemma 3.5.5. For any $a \sqsubseteq b$,

$$\begin{aligned}
 X_a &= \mu(B_a) = \sum_{a \sqsubseteq c \sqsubseteq b} \mu(A_{cb}) \\
 &= \sum_c [a \subseteq c] \cdot [c \subseteq b] \cdot \mu(A_{cb}) \\
 &= \sum_c E[b]_{ac} \cdot Y_c = (E[b] \cdot Y)_a. \quad \square
 \end{aligned}$$

Proof of Theorem 3.5.6. Given a probability measure μ , certainly (i) and (ii) hold of the matrices M and N formed from μ by the rule (3.6). For (iii), we calculate:

$$\begin{aligned}
 (E^{-1}ME)_{ab} &= \sum_{c,d} E_{ac}^{-1} M_{cd} E_{db} = \sum_{c,d} E_{ac}^{-1} M_{cd} E_{db} \\
 &= \sum_{c,d} [a \subseteq c] \cdot (-1)^{|c-a|} \cdot [c = d] \cdot \mu(M_{cd}) \cdot [d \subseteq b] \\
 &= \sum_{a \sqsubseteq c \sqsubseteq b} (-1)^{|c-a|} \cdot \mu(B_c) = \mu(A_{ab}) = N_{ab}.
 \end{aligned}$$

That the correspondence is one-to-one is immediate from Theorem 3.5.4. □

A.4 A DCPO on Markov Kernels

This appendix contains proofs omitted from Section 3.6.

Proof of Theorem 3.6.1. We prove the theorem for our concrete instance $(2^H, \mathcal{B})$. The relation \sqsubseteq is a partial order. Reflexivity and transitivity are clear, and antisymmetry follows from Lemma 3.5.3.

To show that suprema of directed sets exist, let \mathcal{D} be a directed set of measures, and define

$$\left(\bigsqcup \mathcal{D}\right)(B) := \sup_{\mu \in \mathcal{D}} \mu(B), \quad B \in \mathcal{O}.$$

This is clearly the supremum of \mathcal{D} , provided it defines a valid measure.¹ To show this, choose a countable chain $\mu_0 \sqsubseteq \mu_1 \sqsubseteq \dots$ in \mathcal{D} such that $\mu_m \sqsubseteq \mu_n$ for all $m < n$ and $(\bigsqcup \mathcal{D})(B_c) - \mu_n(B_c) \leq 1/n$ for all c such that $|c| \leq n$. Then for all finite $c \in 2^H$, $(\bigsqcup \mathcal{D})(B_c) = \sup_n \mu_n(B_c)$.

Then $\bigsqcup \mathcal{D}$ is a measure by Theorem 3.5.4 because for all finite b and $a \subseteq b$,

$$\begin{aligned} \sum_{a \subseteq c \subseteq b} (-1)^{|c-a|} (\bigsqcup \mathcal{D})(B_c) &= \sum_{a \subseteq c \subseteq b} (-1)^{|c-a|} \sup_n \mu_n(B_c) \\ &= \lim_n \sum_{a \subseteq c \subseteq b} (-1)^{|c-a|} \mu_n(B_c) \\ &\geq 0. \end{aligned}$$

To show that δ_\emptyset is \sqsubseteq -minimum, observe that for all $B \in \mathcal{O}$,

$$\delta_\emptyset(B) = [\emptyset \in B] = [B = B_\emptyset = 2^H]$$

as $B_\emptyset = 2^H$ is the only up-closed set containing \emptyset . Thus for all measures μ , $\delta_\emptyset(2^H) = 1 = \mu(2^H)$, and for all $B \in \mathcal{O}$, $B \neq 2^H$, $\delta_\emptyset(B) = 0 \leq \mu(B)$.

Finally, to show that δ_H is \sqsubseteq -maximum, observe that every nonempty $B \in \mathcal{O}$ contains H because it is up-closed. Therefore, δ_H is the constant function 1 on $\mathcal{O} - \{\emptyset\}$, making it \sqsubseteq -maximum. □

¹This is actually quite subtle. One might be tempted to define

$$\left(\bigsqcup \mathcal{D}\right)(B) := \sup_{\mu \in \mathcal{D}} \mu(B), \quad B \in \mathcal{B}$$

However, this definition would not give a valid probability measure in general. In particular, an increasing chain of measures does not generally converge to its supremum pointwise. However, it *does* converge pointwise on \mathcal{O} .

Proof of Lemma 3.6.2. For any up-closed measurable set B ,

$$\begin{aligned}\mu(B) &= \mu(B) \cdot \nu(2^{\mathbb{H}}) = (\mu \times \nu)(B \times 2^{\mathbb{H}}) \\ &= (\mu \times \nu)(\{(b, c) \mid b \in B\}) \\ &\leq (\mu \times \nu)(\{(b, c) \mid b \cup c \in B\}) = (\mu \& \nu)(B).\end{aligned}$$

and similarly for ν . □

Proof of Lemma 3.6.3. To show that (i), (ii), and (iv) are equivalent,

$$\begin{aligned}\forall a \in 2^{\mathbb{H}} \forall B \in \mathcal{O} \ P(a, B) \leq Q(a, B) \\ \Leftrightarrow \forall a \in 2^{\mathbb{H}} (\forall B \in \mathcal{O} \ P(a, B) \leq Q(a, B)) \\ \Leftrightarrow \forall a \in 2^{\mathbb{H}} \ P(a, -) \sqsubseteq Q(a, -) \\ \Leftrightarrow \forall a \in 2^{\mathbb{H}} \ (\text{curry } P)(a) \sqsubseteq (\text{curry } Q)(a) \\ \Leftrightarrow \text{curry } P \sqsubseteq \text{curry } Q.\end{aligned}$$

To show that (i) and (iii) are equivalent,

$$\begin{aligned}\forall a \in 2^{\mathbb{H}} \forall B \in \mathcal{O} \ P(a, B) \leq Q(a, B) \\ \Leftrightarrow \forall B \in \mathcal{O} (\forall a \in 2^{\mathbb{H}} \ P(a, B) \leq Q(a, B)) \\ \Leftrightarrow \forall B \in \mathcal{O} \ P(-, B) \sqsubseteq Q(-, B).\end{aligned}$$

□

Proof of Theorem 3.6.4. We must show that the supremum of any directed set of continuous Markov kernels is a continuous Markov kernel. In general, the supremum of a directed set of continuous functions between DCPOs is continuous. Given a directed set \mathcal{D} of continuous kernels, we apply this to the directed set $\{\text{curry } P : 2^{\mathbb{H}} \rightarrow \mathcal{M}(2^{\mathbb{H}}) \mid P \in \mathcal{D}\}$ to derive that $\bigsqcup_{P \in \mathcal{D}} \text{curry } P$ is continuous, then use the fact that curry is continuous to infer that $\bigsqcup_{P \in \mathcal{D}} \text{curry } P = \text{curry } \bigsqcup \mathcal{D}$, therefore $\text{curry } \bigsqcup \mathcal{D}$ is continuous. This says that the function $P : 2^{\mathbb{H}} \times \mathcal{B} \rightarrow [0, 1]$ is continuous in its first argument.

We must still argue that the supremum $\bigsqcup \mathcal{D}$ is a Markov kernel, that is, a measurable function in its first argument and a probability measure in its second argument. The first statement follows from the fact that any continuous function is measurable with respect to the Borel sets generated by the topologies of the two spaces. For the second statement, we appeal to Theorem 3.6.1 and the continuity of curry:

$$(\text{curry } \bigsqcup \mathcal{D})(a) = (\bigsqcup_{P \in \mathcal{D}} \text{curry } P)(a) = \bigsqcup_{P \in \mathcal{D}} (\text{curry } P)(a),$$

which is a supremum of a directed set of probability measures, therefore by Theorem 3.6.1 is itself a probability measure.

To show that it is a continuous DCPO with basis of the indicated form, we note that for any $a \in 2^{\mathbb{H}}$ and $B \in \mathcal{O}$,

$$(b \cdot P \cdot d)(a, B) = P(a \cap b, \{c \mid c \cap d \in B\}). \quad (\text{A.1})$$

Every element of the space is the supremum of a directed set of such elements. Given a continuous kernel P , consider the directed set \mathcal{D} of all elements $b \cdot P \cdot d$ for b, d finite. Then for any $a \in 2^{\mathbb{H}}$ and $B \in \mathcal{O}$,

$$(\bigsqcup \mathcal{D})(a, B) = \sup_{b, d \in \wp_{\omega}(\mathbb{H})} P(a \cap b, \{c \mid c \cap d \in B\}) \quad (\text{A.2})$$

$$= \sup_{d \in \wp_{\omega}(\mathbb{H})} P(a, \{c \mid c \cap d \in B\}) \quad (\text{A.3})$$

$$= P(a, B), \quad (\text{A.4})$$

the inference (A.2) from (A.1), the inference (A.3) from the fact that P is continuous in its first argument, and the inference (A.4) from the fact that the sets $\{c \mid c \cap d \in B\}$ for $d \in \wp_{\omega}(\mathbb{H})$ form a directed set of Scott-open sets whose union is B and that P is a measure in its second argument. \square

A.5 Continuity of Kernels and Program Operators and a Least-Fixpoint Characterization of Iteration

This appendix contains lemmas and proofs omitted from Section 3.7.

A.5.1 Products and Integration

This section develops some properties of products and integration needed for from the point of view of Scott topology.

As pointed out by Jones [71, §3.6], the product σ -algebra of the Borel sets of two topological spaces X, Y is in general not the same as the Borel sets of the topological product $X \times Y$, although this property does hold for the Cantor space, as its basic open sets are clopen. More importantly, as also observed in [71, §3.6], the Scott topology on the product of DCPOs with the componentwise order is not necessarily the same as the product topology. However, in our case, the two topologies coincide.

Theorem A.5.1. *Let D_α , $\alpha < \kappa$, be a collection of algebraic DCPOs with F_α the finite elements of D_α . Then the product $\prod_{\alpha < \kappa} D_\alpha$ with the componentwise order is an algebraic DCPO with finite elements*

$$F = \{c \in \prod_{\alpha} F_{\alpha} \mid \pi_{\alpha}(c) = \perp \text{ for all but finitely many } \alpha\}.$$

Proof. The projections $\pi_{\beta} : \prod_{\alpha} D_{\alpha} \rightarrow D_{\beta}$ are easily shown to be continuous with respect to the componentwise order. For any $d \in \prod_{\alpha < \kappa} D_{\alpha}$, the set $\{d\}_{\downarrow} \cap F$ is directed, and $d = \bigsqcup(\{d\}_{\downarrow} \cap F)$: for any α , the set $\pi_{\alpha}(\{d\}_{\downarrow} \cap F) = \{\pi_{\alpha}(d)\}_{\downarrow} \cap F_{\alpha}$ is directed, thus

$$\begin{aligned} \pi_{\alpha}(d) &= \bigsqcup(\{\pi_{\alpha}(d)\}_{\downarrow} \cap F_{\alpha}) = \bigsqcup(\pi_{\alpha}(\{d\}_{\downarrow} \cap F)) \\ &= \pi_{\alpha}(\bigsqcup(\{d\}_{\downarrow} \cap F)), \end{aligned}$$

and as α was arbitrary, $d = \bigsqcup(\{d\}_{\downarrow} \cap F)$.

It remains to show that $\{c\}^\uparrow = \prod_{\alpha < \kappa} \{\pi_\alpha(c)\}^\uparrow$ is open for $c \in F$. Let A be a directed set with $\bigsqcup A \in \{c\}^\uparrow$. For each α , $\{\pi_\alpha(a) \mid a \in A\}$ is directed, and

$$\bigsqcup_{a \in A} \pi_\alpha(a) = \pi_\alpha(\bigsqcup A) \in \pi_\alpha(\{c\}^\uparrow) = \{\pi_\alpha(c)\}^\uparrow,$$

so there exists $a_\alpha \in A$ such that $\pi_\alpha(a_\alpha) \in \{\pi_\alpha(c)\}^\uparrow$. Since A is directed, there is a single $a \in A$ that majorizes the finitely many a_α such that $\pi_\alpha(c) \neq \perp$. Then $\pi_\alpha(a) \in \{\pi_\alpha(c)\}^\uparrow$ for all α , thus $a \in \{c\}^\uparrow$. \square

Corollary A.5.2. *The Scott topology on a product of algebraic DCPOs with respect to the componentwise order coincides with the product topology induced by the Scott topology on each component.*

Proof. Let $\prod_{\alpha < \kappa} D_\alpha$ be a product of algebraic DCPOs with \mathcal{O}_0 the product topology and \mathcal{O}_1 the Scott topology. As noted in the proof of Theorem A.5.1, the projections $\pi_\beta : \prod_\alpha D_\alpha \rightarrow D_\beta$ are continuous with respect to \mathcal{O}_1 . By definition, \mathcal{O}_0 is the weakest topology on the product such that the projections are continuous, so $\mathcal{O}_0 \subseteq \mathcal{O}_1$.

For the reverse inclusion, we use the observation that the sets $\{c\}^\uparrow$ for finite elements $c \in F$ as defined in Theorem A.5.1 form a base for the topology \mathcal{O}_1 . These sets are also open in \mathcal{O}_0 , since they are finite intersections of sets of the form $\pi_\alpha^{-1}(\{\pi_\alpha(c)\}^\uparrow)$, and $\{\pi_\alpha(c)\}^\uparrow$ is open in D_α since $\pi_\alpha(c) \in F_\alpha$. As \mathcal{O}_1 is the smallest topology containing its basic open sets, $\mathcal{O}_1 \subseteq \mathcal{O}_0$. \square

A function $g : 2^{\mathcal{H}} \rightarrow \mathbb{R}_+$ is \mathcal{O} -simple if it is a finite linear combination of the form $\sum_{A \in F} r_A \mathbf{1}_A$, where F is a finite subset of \mathcal{O} . Let $S_{\mathcal{O}}$ denote the set of \mathcal{O} -simple functions.

Theorem A.5.3. *Let f be a bounded Scott-continuous function $f : 2^{\mathcal{H}} \rightarrow \mathbb{R}_+$. Then*

$$\sup_{\substack{g \in S_{\mathcal{O}} \\ g \leq f}} \int g \, d\mu = \int f \, d\mu = \inf_{\substack{g \in S_{\mathcal{O}} \\ f \leq g}} \int g \, d\mu$$

under Lebesgue integration.

Proof. Let $\varepsilon > 0$ and $r_N = \sup_{a \in 2^H} f(a)$. Let

$$0 = r_0 < r_1 < \cdots < r_N$$

such that $r_{i+1} - r_i < \varepsilon$, $0 \leq i \leq N - 1$, and set

$$A_i = \{a \mid f(a) > r_i\} = f^{-1}((r_i, \infty)) \in \mathcal{O}, \quad 0 \leq i \leq N.$$

Then $A_{i+1} \subseteq A_i$ and

$$A_i - A_{i+1} = \{a \mid r_i < f(a) \leq r_{i+1}\} = f^{-1}((r_i, r_{i+1}]).$$

Let

$$f_{\bullet} = \sum_{i=0}^{N-1} r_i \mathbf{1}_{A_i - A_{i+1}} \qquad f^{\bullet} = \sum_{i=0}^{N-1} r_{i+1} \mathbf{1}_{A_i - A_{i+1}}.$$

For $a \in A_i - A_{i+1}$,

$$\begin{aligned} f_{\bullet}(a) &= \sum_{i=0}^{N-1} r_i \mathbf{1}_{A_i - A_{i+1}}(a) = r_i < f(a) \\ &\leq r_{i+1} = \sum_{i=0}^{N-1} r_{i+1} \mathbf{1}_{A_i - A_{i+1}}(a) = f^{\bullet}(a), \end{aligned}$$

and as a was arbitrary, $f_{\bullet} \leq f \leq f^{\bullet}$ pointwise. Thus

$$\int f_{\bullet} d\mu \leq \int f d\mu \leq \int f^{\bullet} d\mu.$$

Moreover,

$$\begin{aligned} \int f^{\bullet} d\mu - \int f_{\bullet} d\mu &= \sum_{i=0}^{N-1} r_{i+1} \mu(A_i - A_{i+1}) \\ &\quad - \sum_{i=0}^{N-1} r_i \mu(A_i - A_{i+1}) \\ &= \sum_{i=0}^{N-1} (r_{i+1} - r_i) \mu(A_i - A_{i+1}) \\ &< \varepsilon \cdot \sum_{i=0}^{N-1} \mu(A_i - A_{i+1}) = \varepsilon \cdot \mu(2^H) = \varepsilon, \end{aligned}$$

so the integral is approximated arbitrarily closely from above and below by the f^\bullet and f_\bullet . Finally, we argue that f_\bullet and f^\bullet are \mathcal{O} -simple. Using the fact that $r_0 = 0$ and $A_N = \emptyset$ to reindex,

$$\begin{aligned}
f_\bullet &= \sum_{i=0}^{N-1} r_i \mathbf{1}_{A_i - A_{i+1}} = \sum_{i=0}^{N-1} r_i \mathbf{1}_{A_i} - \sum_{i=0}^{N-1} r_i \mathbf{1}_{A_{i+1}} \\
&= \sum_{i=0}^{N-1} r_{i+1} \mathbf{1}_{A_{i+1}} - \sum_{i=0}^{N-1} r_i \mathbf{1}_{A_{i+1}} = \sum_{i=0}^{N-1} (r_{i+1} - r_i) \mathbf{1}_{A_{i+1}}, \\
f^\bullet &= \sum_{i=0}^{N-1} r_{i+1} \mathbf{1}_{A_i - A_{i+1}} = \sum_{i=0}^{N-1} r_{i+1} \mathbf{1}_{A_i} - \sum_{i=0}^{N-1} r_{i+1} \mathbf{1}_{A_{i+1}} \\
&= \sum_{i=0}^{N-1} r_{i+1} \mathbf{1}_{A_i} - \sum_{i=0}^{N-1} r_i \mathbf{1}_{A_i} = \sum_{i=0}^{N-1} (r_{i+1} - r_i) \mathbf{1}_{A_i},
\end{aligned}$$

and both functions are \mathcal{O} -simple since all A_i are in \mathcal{O} . \square

We can prove a stronger version of Theorem A.5.3 that also works for functions taking on infinite value. A function g is simple if it is a finite linear combination of indicator functions of the form $g = \sum_{i=1}^k r_i \mathbf{1}_{A_i}$, where $k \in \mathbb{N}$ and the A_i are measurable. Let S denote the set of all simple functions.

Theorem A.5.4. *Let $f : 2^H \rightarrow [0, \infty]$ be Scott-continuous and let μ be a probability measure.*

Then

$$\int f d\mu = \sup_{\substack{g \in S_{\mathcal{O}} \\ g \leq f}} \int g d\mu$$

Proof. It suffices to show that

$$\sup_{\substack{g \in S \\ g \leq f}} \int g d\mu = \sup_{\substack{g \in S_{\mathcal{O}} \\ g \leq f}} \int g d\mu \tag{A.5}$$

since the left side of this equation defines the integral of f . We trivially have

$$\sup_{\substack{g \in S \\ g \leq f}} \int g d\mu \geq \sup_{\substack{g \in S_{\mathcal{O}} \\ g \leq f}} \int g d\mu \tag{A.6}$$

because $S_{\mathcal{O}} \subseteq S$. To show the reverse inequality, let $g \in S$ with $g \leq f$ be arbitrary. We will show that there exists a family of functions $g_\varepsilon \in S_{\mathcal{O}}$, $\varepsilon > 0$ with $g_\varepsilon \leq f$ such that $\int g d\mu - \int g_\varepsilon d\mu \leq \varepsilon$. Together with (A.6), this proves (A.5) because it implies that

$$\sup_{\substack{g \in S \\ g \leq f}} \int g d\mu \leq \sup_{\substack{g \in S \\ g \leq f}} \sup_{\varepsilon > 0} \int g_\varepsilon d\mu \leq \sup_{\substack{g \in S_{\mathcal{O}} \\ g \leq f}} \int g d\mu$$

Let's turn to constructing the family of functions $g_\varepsilon \in S_{\mathcal{O}}$. Since g is simple, we may w.l.o.g. assume that it has the form $g = \sum_{i=1}^k r_i \mathbf{1}_{A_i}$ with disjoint $A_i \in \mathcal{B}$ and $r_1 < r_2 < \dots < r_k$. Define

$$r_0 := \varepsilon$$

$$B_{i,\varepsilon} := f^{-1}((r_i - \varepsilon, \infty]) \in \mathcal{O}$$

$$\beta_i := r_i - r_{i-1}$$

$$g_\varepsilon := \sum_{i=1}^k \beta_i \cdot \mathbf{1}_{B_{i,\varepsilon}} \in S_{\mathcal{O}}$$

Then we have $g_\varepsilon \leq f$ because for all $a \in 2^H$

$$\begin{aligned} \left(\sum_{i=1}^k \beta_i \cdot \mathbf{1}_{B_{i,\varepsilon}} \right)(a) &= \sum_{i=1}^k \beta_i \cdot [a \in B_{i,\varepsilon}] \\ &= \sum_{i=1}^k (r_i - r_{i-1}) \cdot [f(a) > r_i - \varepsilon] \\ &= \max\{r_i \mid 1 \leq i \leq k \text{ and } f(a) > r_i - \varepsilon\} - r_0 \\ &< f(a) \end{aligned}$$

Moreover, we have that $g - g_\varepsilon \leq \varepsilon$ because

$$\begin{aligned} \left(\sum_{i=1}^k \beta_i \cdot \mathbf{1}_{B_{i,\varepsilon}} \right)(a) &= \max\{r_i \mid 1 \leq i \leq k \text{ and } f(a) > r_i - \varepsilon\} - r_0 \\ &\geq \max\{r_i \mid 1 \leq i \leq k \text{ and } f(a) \geq r_i\} - \varepsilon \\ &\geq \max\{r_i \mid 1 \leq i \leq k \text{ and } g(a) = r_i\} - \varepsilon \\ &= g(a) - \varepsilon \end{aligned}$$

Thus it follows that

$$\int g d\mu - \int g_\varepsilon d\mu = \int (g - g_\varepsilon) d\mu \leq \int \varepsilon d\mu = \varepsilon$$

□

Proof of Theorem 3.7.1. (i) We prove the result first for \mathcal{O} -simple functions. If $\mu \sqsubseteq \nu$, then for any \mathcal{O} -simple function $g = \sum_A r_A \mathbf{1}_A$,

$$\begin{aligned} \int g d\mu &= \int \sum_A r_A \mathbf{1}_A d\mu = \sum_A r_A \mu(A) \\ &\leq \sum_A r_A \nu(A) = \int \sum_A r_A \mathbf{1}_A d\nu = \int g d\nu. \end{aligned}$$

Thus the map (3.7) is monotone. If \mathcal{D} is a directed set of measures with respect to \sqsubseteq , then

$$\begin{aligned} \int g d(\bigsqcup \mathcal{D}) &= \int \sum_A r_A \mathbf{1}_A d(\bigsqcup \mathcal{D}) = \sum_A r_A (\bigsqcup \mathcal{D})(A) \\ &= \sup_{\mu \in \mathcal{D}} \sum_A r_A \mu(A) = \sup_{\mu \in \mathcal{D}} \int \sum_A r_A \mathbf{1}_A d\mu \\ &= \sup_{\mu \in \mathcal{D}} \int g d\mu. \end{aligned}$$

Now consider an arbitrary Scott-continuous function $f : 2^H \rightarrow [0, \infty]$. Let $S_{\mathcal{O}}$ be the family of \mathcal{O} -simple functions. By Theorem A.5.4, if $\mu \sqsubseteq \nu$, we have

$$\int f d\mu = \sup_{\substack{g \in S_{\mathcal{O}} \\ g \leq f}} \int g d\mu \leq \sup_{\substack{g \in S_{\mathcal{O}} \\ g \leq f}} \int g d\nu = \int f d\nu,$$

and if \mathcal{D} is a directed set of measures with respect to \sqsubseteq , then

$$\begin{aligned} \int f d(\bigsqcup \mathcal{D}) &= \sup_{\substack{g \in S_{\mathcal{O}} \\ g \leq f}} \int g d(\bigsqcup \mathcal{D}) = \sup_{\substack{g \in S_{\mathcal{O}} \\ g \leq f}} \sup_{\mu \in \mathcal{D}} \int g d\mu \\ &= \sup_{\mu \in \mathcal{D}} \sup_{\substack{g \in S_{\mathcal{O}} \\ g \leq f}} \int g d\mu = \sup_{\mu \in \mathcal{D}} \int f d\mu. \end{aligned}$$

(ii) This just the monotone convergence theorem for Lebesgue Integration. □

A.5.2 Continuous Operations on Measures

In this section we show that certain operations on measures are continuous. These properties will be lifted to kernels as required.

Lemma A.5.5. *For any probability measure μ on an algebraic DCPO and open set B , the value $\mu(B)$ is approximated arbitrarily closely from below by $\mu(C)$ for compact-open sets C .*

Proof. Since the sets $\{a\}^\uparrow$ for finite a form a base for the topology, and every compact-open set is a finite union of such sets, the set $\mathcal{K}(B)$ of compact-open subsets of B is a directed set whose union is B . Then

$$\mu(B) = \mu(\bigcup \mathcal{K}(B)) = \sup\{\mu(C) \mid C \in \mathcal{K}(B)\}.$$

□

Lemma A.5.6. *The product operator on measures in algebraic DCPOs is Scott-continuous in each argument.*

Proof. The difficult part of the argument is monotonicity. Once we have that, then for any $B, C \in \mathcal{O}$, we have $(\mu \times \nu)(B \times C) = \mu(B) \cdot \nu(C)$. Thus for any directed set D of measures,

$$\begin{aligned} (\bigsqcup D \times \nu)(B \times C) &= (\bigsqcup D)(B) \cdot \nu(C) = \left(\sup_{\mu \in D} \mu(B)\right) \cdot \nu(C) \\ &= \sup_{\mu \in D} (\mu(B) \cdot \nu(C)) = \sup_{\mu \in D} ((\mu \times \nu)(B \times C)) \\ &= \left(\bigsqcup_{\mu \in D} (\mu \times \nu)\right)(B \times C). \end{aligned}$$

By Theorem A.5.1, the sets $B \times C$ for $B, C \in \mathcal{O}$ form a basis for the Scott topology on the product space $2^H \times 2^H$, thus $\bigsqcup D \times \nu = \bigsqcup_{\mu \in D} (\mu \times \nu)$.

To show monotonicity, we use approximability by compact-open sets (Lemma A.5.5). We wish to show that if $\mu_1 \sqsubseteq \mu_2$, then $\mu_1 \times \nu \sqsubseteq \mu_2 \times \nu$. By Lemma A.5.5, it suffices to show that

$$(\mu_1 \times \nu)\left(\bigcup_n B_n \times C_n\right) \leq (\mu_2 \times \nu)\left(\bigcup_n B_n \times C_n\right),$$

where the index n ranges over a finite set, and B_n and C_n are open sets of the component spaces. Consider the collection of all atoms A of the Boolean algebra generated by the C_n . For each such atom A , let

$$N(A) = \{n \mid C_n \text{ occurs positively in } A\}.$$

Then

$$\bigcup_n B_n \times C_n = \bigcup_A \left(\bigcup_{n \in N(A)} B_n \right) \times A.$$

The right-hand side is a disjoint union, since the A are pairwise disjoint. Then

$$\begin{aligned} (\mu_1 \times \nu)\left(\bigcup_n B_n \times C_n\right) &= (\mu_1 \times \nu)\left(\bigcup_A \left(\bigcup_{n \in N(A)} B_n \right) \times A\right) \\ &= \sum_A (\mu_1 \times \nu)\left(\left(\bigcup_{n \in N(A)} B_n \right) \times A\right) \\ &= \sum_A \mu_1\left(\bigcup_{n \in N(A)} B_n\right) \cdot \nu(A) \\ &\leq \sum_A \mu_2\left(\bigcup_{n \in N(A)} B_n\right) \cdot \nu(A) \\ &= (\mu_2 \times \nu)\left(\bigcup_n B_n \times C_n\right). \end{aligned}$$

□

Let S and T be measurable spaces and $f : S \rightarrow T$ a measurable function. For a measure μ on S , the *push-forward measure* $f_*(\mu)$ is the measure $\mu \circ f^{-1}$ on T .

Lemma A.5.7. *If $f : (2^H)^\kappa \rightarrow 2^H$ is Scott-continuous with respect to the subset order, then the push-forward operator $f_* : \mathcal{M}((2^H)^\kappa) \rightarrow \mathcal{M}(2^H)$ is Scott-continuous with respect to \sqsubseteq .*

Proof. Let $\mu, \nu \in \mathcal{M}((2^H)^\kappa)$, $\mu \sqsubseteq \nu$. If $B \in \mathcal{O}$, then $f^{-1}(B)$ is Scott-open in $(2^H)^\kappa$, so $f_*(\mu)(B) = \mu(f^{-1}(B)) \leq \nu(f^{-1}(B)) = f_*(\nu)(B)$. As $B \in \mathcal{O}$ was arbitrary, $f_*(\mu) \sqsubseteq f_*(\nu)$. Similarly, if D is any \sqsubseteq -directed set in $\mathcal{M}((2^H)^\kappa)$, then so is $\{f_*(\mu) \mid \mu \in D\}$, and

$$\begin{aligned} f_*(\bigsqcup D)(B) &= (\bigsqcup D)(f^{-1}(B)) = \sup_{\mu \in D} \mu(f^{-1}(B)) \\ &= \sup_{\mu \in D} f_*(\mu)(B) = (\bigsqcup_{\mu \in D} f_*(\mu))(B) \end{aligned}$$

for any $B \in \mathcal{O}$, thus $f_*(\bigsqcup D) = \bigsqcup_{\mu \in D} f_*(\mu)$. \square

Lemma A.5.8. *Parallel composition of measures ($\&$) is Scott-continuous in each argument.*

Proof. By definition, $\mu \& \nu = (\mu \times \nu) \cdot \bigcup^{-1}$, where $\bigcup : 2^H \times 2^H \rightarrow 2^H$ is the set union operator. The set union operator is easily shown to be continuous with respect to the Scott topologies on $2^H \times 2^H$ and the 2^H . By Lemma A.5.7, the push-forward operator with respect to union is Scott-continuous with respect to \sqsubseteq . By Lemma A.5.6, the product operator is Scott-continuous in each argument with respect to \sqsubseteq . The operator $\&$ is the composition of these two Scott continuous operators, therefore is itself Scott-continuous. \square

A.5.3 Continuous Kernels

Lemma A.5.9. *The deterministic kernel associated with any Scott-continuous function $f : D \rightarrow E$ is a continuous kernel.*

Proof. Recall from [44] that deterministic kernels are those whose output measures are Dirac measures (point masses). Any measurable function $f : D \rightarrow E$ uniquely determines a deterministic kernel P_f such that $P_f(a, -) = \delta_{f(a)}$ (or equivalently, $P = \eta \circ f$) and vice versa (this was shown in [44] for $D = E = 2^H$). We show that if in addition f is Scott-continuous, then the kernel P_f is continuous.

Let $f : D \rightarrow E$ be Scott-continuous. For any open B , if $a \sqsubseteq b$, then $f(a) \sqsubseteq f(b)$ since f is monotone. Since B is up-closed, if $f(a) \in B$, then $f(b) \in B$. Thus

$$P_f(a, B) = [f(a) \in B] \leq [f(b) \in B] = P_f(b, B).$$

If $A \subseteq D$ is a directed set, then $f(\bigsqcup A) = \bigsqcup_{a \in A} f(a)$. Since B is open, $\bigsqcup_{a \in A} f(a) \in B$ iff there exists $a \in A$ such that $f(a) \in B$. Then

$$\begin{aligned} P_f(\bigsqcup A, B) &= [f(\bigsqcup A) \in B] = [\bigsqcup_{a \in A} f(a) \in B] \\ &= \sup_{a \in A} [f(a) \in B] = \sup_{a \in A} P_f(a, B). \end{aligned}$$

□

Lemma A.5.10. *All atomic ProbNetKAT programs (including predicates) denote deterministic and Scott-continuous kernels.*

Proof. By Lemma 3.4.2, all atomic programs denote kernels of the form $a \mapsto \eta(\{f(h) \mid h \in a\})$, where f is a partial function $H \rightarrow H$. Hence they are deterministic. Using Lemma A.5.9, we see that they are also Scott-continuous:

- If $a \subseteq b$, then $\{f(h) \mid h \in a\} \subseteq \{f(h) \mid h \in b\}$; and
- If $D \subseteq 2^H$ is a directed set, then $\{f(h) \mid h \in \bigcup D\} = \bigcup_{a \in D} \{f(h) \mid h \in a\}$.

□

Lemma A.5.11. *Let P be a continuous Markov kernel and $f : 2^H \rightarrow \mathbb{R}_+$ a Scott-continuous function. Then the map*

$$a \mapsto \int_{c \in 2^H} f(c) \cdot P(a, dc) \tag{A.7}$$

is Scott-continuous.

Proof. The map (A.7) is the composition of the maps

$$a \mapsto P(a, -) \qquad P(a, -) \mapsto \int_{c \in 2^H} P(a, dc) \cdot f(c),$$

which are Scott-continuous by Lemmas A.5.19 and 3.7.1, respectively, and the composition of Scott-continuous maps is Scott-continuous. \square

Lemma A.5.12. *Product preserves continuity of Markov kernels: If P and Q are continuous, then so is $P \times Q$.*

Proof. We wish to show that if $a \subseteq b$, then $(P \times Q)(a, -) \sqsubseteq (P \times Q)(b, -)$, and if A is a directed subset of 2^H , then $(P \times Q)(\bigcup A) = \sup_{a \in A} (P \times Q)(a, -)$. For the first statement, using Lemma A.5.6 twice,

$$\begin{aligned} (P \times Q)(a, -) &= P(a, -) \times Q(a, -) \sqsubseteq P(b, -) \times Q(a, -) \\ &\sqsubseteq P(b, -) \times Q(b, -) = (P \times Q)(b, -). \end{aligned}$$

For the second statement, for A a directed subset of 2^H ,

$$\begin{aligned} (P \times Q)(\bigcup A, -) &= P(\bigcup A, -) \times Q(\bigcup A, -) \\ &= (\bigsqcup_{a \in A} P(a, -)) \times (\bigsqcup_{b \in A} Q(b, -)) \\ &= \bigsqcup_{a \in A} \bigsqcup_{b \in A} P(a, -) \times Q(b, -) \\ &= \bigsqcup_{a \in A} P(a, -) \times Q(a, -) \\ &= \bigsqcup_{a \in A} (P \times Q)(a, -). \end{aligned}$$

\square

Lemma A.5.13. *Sequential composition preserves continuity of Markov kernels: If P and Q are continuous, then so is $P \cdot Q$.*

Proof. We have

$$(P \cdot Q)(a, A) = \int_{c \in 2^H} P(a, dc) \cdot Q(c, A).$$

Since Q is a continuous kernel, it is Scott-continuous in its first argument, thus so is $P \cdot Q$ by Lemma A.5.11. \square

Lemma A.5.14. *Parallel composition preserves continuity of Markov kernels: If P and Q are continuous, then so is $P \& Q$.*

Proof. Suppose P and Q are continuous. By definition, $P \& Q = (P \times Q) \cdot \bigcup$. By Lemma A.5.12, $P \times Q$ is continuous, and $\bigcup : 2^H \times 2^H \rightarrow 2^H$ is continuous. Thus their composition is continuous by Lemma A.5.13. \square

Lemma A.5.15. *The probabilistic choice operator (\oplus_r) preserves continuity of kernels.*

Proof. If P and Q are continuous, then $P \oplus_r Q = rP + (1 - r)Q$. If $a \subseteq b$, then

$$\begin{aligned} (P \oplus_r Q)(a, -) &= rP(a, -) + (1 - r)Q(a, -) \\ &\leq rP(b, -) + (1 - r)Q(b, -) \\ &= (P \oplus_r Q)(b, -). \end{aligned}$$

If $A \subseteq 2^H$ is a directed set, then

$$\begin{aligned} (P \oplus_r Q)(\bigcup A, -) &= rP(\bigcup A, -) + (1 - r)Q(\bigcup A, -) \\ &= \bigsqcup_{a \in A} (rP(a, -) + (1 - r)Q(a, -)) \\ &= \bigsqcup_{a \in A} (P \oplus_r Q)(a, -). \end{aligned}$$

\square

Lemma A.5.16. *The iteration operator $(*)$ preserves continuity of kernels.*

Proof. Suppose P is continuous. It follows inductively using Lemmas A.5.14 and A.5.13 that $P^{(n)}$ is continuous. Since $P^* = \bigsqcup_n P^{(n)}$ and since the supremum of a directed set of continuous kernels is continuous by Theorem 3.6.4, P^* is continuous. \square

Proof of Theorem 3.7.2. The result follows from Lemmas A.5.9, A.5.11, A.5.12, A.5.13, A.5.14, A.5.15, and A.5.16. \square

Proof of Corollary 3.7.3. This follows from Theorem 3.7.2. All primitive programs are deterministic, thus give continuous kernels, and continuity is preserved by all the program operators. \square

A.5.4 Continuous Operations on Kernels

Lemma A.5.17. *The product operation on kernels (\times) is Scott-continuous in each argument.*

Proof. We use Lemma A.5.6. If $P_1 \sqsubseteq P_2$, then for all $a \in 2^H$,

$$\begin{aligned} (P_1 \times Q)(a, -) &= P_1(a, -) \times Q(a, -) \\ &\sqsubseteq P_2(a, -) \times Q(a, -) = (P_2 \times Q)(a, -). \end{aligned}$$

Since a was arbitrary, $P_1 \times Q \sqsubseteq P_2 \times Q$. For a directed set \mathcal{D} of kernels,

$$\begin{aligned} (\bigsqcup \mathcal{D} \times Q)(a, -) &= (\bigsqcup \mathcal{D})(a, -) \times Q(a, -) \\ &= \bigsqcup_{P \in \mathcal{D}} P(a, -) \times Q(a, -) \\ &= \bigsqcup_{P \in \mathcal{D}} (P(a, -) \times Q(a, -)) \\ &= \bigsqcup_{P \in \mathcal{D}} (P \times Q)(a, -) \\ &= (\bigsqcup_{P \in \mathcal{D}} (P \times Q))(a, -). \end{aligned}$$

Since a was arbitrary, $\bigsqcup \mathcal{D} \times Q = \bigsqcup_{P \in \mathcal{D}} (P \times Q)$. \square

Lemma A.5.18. *Parallel composition of kernels ($\&$) is Scott-continuous in each argument.*

Proof. By definition, $P \& Q = (P \times Q) \cdot \cup$. By Lemmas A.5.17 and A.5.20, the product operation and sequential composition are continuous in both arguments, thus their composition is. \square

Lemma A.5.19. *Let P be a continuous Markov kernel. The map $\text{curry } P$ is Scott-continuous with respect to the subset order on 2^H and the order \sqsubseteq on $\mathcal{M}(2^H)$.*

Proof. We have $(\text{curry } P)(a) = P(a, -)$. Since P is monotone in its first argument, if $a \subseteq b$ and $B \in \mathcal{O}$, then $P(a, B) \leq P(b, B)$. As $B \in \mathcal{O}$ was arbitrary,

$$(\text{curry } P)(a) = P(a, -) \sqsubseteq P(b, -) = (\text{curry } P)(b).$$

This shows that $\text{curry } P$ is monotone.

Let $D \subseteq 2^H$ be a directed set. By the monotonicity of $\text{curry } P$, so is the set $\{(\text{curry } P)(a) \mid a \in D\}$. Then for any $B \in \mathcal{O}$,

$$\begin{aligned} (\text{curry } P)(\bigcup D)(B) &= P(\bigcup D, B) = \sup_{a \in D} P(a, B) \\ &= \sup_{a \in D} (\text{curry } P)(a)(B) \\ &= (\bigsqcup_{a \in D} (\text{curry } P)(a))(B), \end{aligned}$$

thus $(\text{curry } P)(\bigcup D) = \bigsqcup_{a \in D} (\text{curry } P)(a)$. □

Lemma A.5.20. *Sequential composition of kernels is Scott-continuous in each argument.*

Proof. To show that \cdot is continuous in its first argument, we wish to show that if P_1, P_2, Q are any continuous kernels with $P_1 \sqsubseteq P_2$, and if \mathcal{D} is any directed set of continuous kernels, then

$$P_1 \cdot Q \leq P_2 \cdot Q \qquad (\bigsqcup \mathcal{D}) \cdot Q = \bigsqcup_{P \in \mathcal{D}} (P \cdot Q).$$

We must show that for all $a \in 2^H$ and $B \in \mathcal{O}$,

$$\begin{aligned} \int_c P_1(a, dc) \cdot Q(c, B) &\leq \int_c P_2(a, dc) \cdot Q(c, B) \\ \int_c (\bigsqcup \mathcal{D})(a, dc) \cdot Q(c, B) &= \sup_{P \in \mathcal{D}} \int_c P(a, dc) \cdot Q(c, B). \end{aligned}$$

By Lemma 3.6.3, for all $a \in 2^H$, $P_1(a, -) \sqsubseteq P_2(a, -)$ and $(\bigsqcup \mathcal{D})(a, -) = \bigsqcup_{P \in \mathcal{D}} P(a, -)$, and $Q(-, B)$ is a Scott-continuous function by assumption. The result follows from Lemma 3.7.1(i).

The argument that \cdot is continuous in its second argument is similar, using Lemma 3.7.1(ii). We wish to show that if P, Q_1, Q_2 are any continuous kernels with $Q_1 \sqsubseteq Q_2$, and if \mathcal{D} is any directed set of continuous kernels, then

$$P \cdot Q_1 \leq P \cdot Q_2 \qquad P \cdot \bigsqcup \mathcal{D} = \bigsqcup_{Q \in \mathcal{D}} (P \cdot Q).$$

We must show that for all $a \in 2^H$ and $B \in \mathcal{O}$,

$$\begin{aligned} \int_c P(a, dc) \cdot Q_1(c, B) &\leq \int_c P(a, dc) \cdot Q_2(c, B) \\ \int_c P(a, dc) \cdot (\bigsqcup \mathcal{D})(c, B) &= \sup_{Q \in \mathcal{D}} \int_c P(a, dc) \cdot Q(c, B). \end{aligned}$$

By Lemma 3.6.3, for all $B \in \mathcal{O}$, $Q_1(-, B) \sqsubseteq Q_2(-, B)$ and $(\bigsqcup \mathcal{D})(-, B) = \bigsqcup_{Q \in \mathcal{D}} Q(-, B)$.

The result follows from Lemma 3.7.1(ii). \square

Lemma A.5.21. *The probabilistic choice operator applied to kernels (\oplus_r) is continuous in each argument.*

Proof. If P and Q are continuous, then $P \oplus_r Q = rP + (1 - r)Q$. If $P_1 \sqsubseteq P_2$, then for any $a \in 2^H$ and $B \in \mathcal{O}$,

$$\begin{aligned} (P_1 \oplus_r Q)(a, B) &= rP_1(a, B) + (1 - r)Q(a, B) \\ &\leq rP_2(a, B) + (1 - r)Q(a, B) \\ &= (P_2 \oplus_r Q)(a, B), \end{aligned}$$

so $P_1 \oplus_r Q \sqsubseteq P_2 \oplus_r Q$. If \mathcal{D} is a directed set of kernels and $B \in \mathcal{O}$, then

$$\begin{aligned} (\bigsqcup \mathcal{D} \oplus_r Q)(a, B) &= r(\bigsqcup \mathcal{D})(a, B) + (1 - r)Q(a, B) \\ &= \sup_{P \in \mathcal{D}} (rP(a, B) + (1 - r)Q(a, B)) \\ &= \sup_{P \in \mathcal{D}} (P \oplus_r Q)(a, B). \end{aligned}$$

\square

Lemma A.5.22. *If $P \sqsubseteq Q$ then $P^{(n)} \sqsubseteq Q^{(n)}$.*

Proof. By induction on $n \in \mathbb{N}$. The claim is trivial for $n = 0$. For $n > 0$, we assume that $P^{(n-1)} \sqsubseteq Q^{(n-1)}$ and deduce

$$P^{(n)} = \text{skip} \ \& \ P \cdot P^{(n-1)} \sqsubseteq \text{skip} \ \& \ Q \cdot Q^{(n-1)} = Q^{(n)}$$

by monotonicity of sequential and parallel composition (Lemmas A.5.20 and A.5.18, respectively). \square

Lemma A.5.23. *If $m \leq n$ then $P^{(m)} \sqsubseteq P^{(n)}$.*

Proof. We have $P^{(0)} \sqsubseteq P^{(1)}$ by Lemmas 3.6.2 and 3.6.3. Proceeding by induction using Lemma A.5.22, we have $P^{(n)} \sqsubseteq P^{(n+1)}$ for all n . The result follows from transitivity. \square

Lemma A.5.24. *The iteration operator applied to kernels ($*$) is continuous.*

Proof. It is a straightforward consequence of Lemma A.5.22 and Theorem 3.7.7 that if $P \sqsubseteq Q$, then $P^* \sqsubseteq Q^*$. Now let \mathcal{D} be a directed set of kernels. It follows by induction using Lemmas A.5.18 and A.5.20 that the operator $P \mapsto P^{(n)}$ is continuous, thus

$$\begin{aligned} (\bigsqcup \mathcal{D})^* &= \bigsqcup_n (\bigsqcup \mathcal{D})^{(n)} = \bigsqcup_n \bigsqcup_{P \in \mathcal{D}} P^{(n)} \\ &= \bigsqcup_{P \in \mathcal{D}} \bigsqcup_n P^{(n)} = \bigsqcup_{P \in \mathcal{D}} P^*. \end{aligned}$$

\square

Proof of Theorem 3.7.4. The result follows from Lemmas A.5.17, A.5.18, A.5.19, A.5.20, A.5.21, and A.5.24. \square

A.5.5 Iteration as Least Fixpoint

In this section we show that the semantics of iteration presented in [44], defined in terms of an infinite process, coincides with the least fixpoint semantics presented here.

In this section, we use the notation P^* refers to the semantics of [44]. For the iterate introduced here, we use $\bigsqcup_n P^{(n)}$.

Recall from [44] the approximants

$$P^{(0)} = \text{skip} \qquad P^{(m+1)} = \text{skip} \ \& \ P \cdot P^{(m)}.$$

It was shown in [44] that for any $c \in 2^H$, the measures $P^{(m)}(c, -)$ converge weakly to $P^*(c, -)$; that is, for any bounded (Cantor-)continuous real-valued function f on 2^H , the expected values of f with respect to the measures $P^{(m)}(c, -)$ converge to the expected value of f with respect to $P^*(c, -)$:

$$\lim_{m \rightarrow \infty} \int_{a \in 2^H} f(a) \cdot P^{(m)}(c, da) = \int_{a \in 2^H} f(a) \cdot P^*(c, da).$$

Theorem A.5.25. *The kernel $Q = \bigsqcup_{n \in \mathbb{N}} P^{(n)}$ is the unique fixpoint of $(\lambda Q. \text{skip} \ \& \ P \cdot Q)$ such that $P^{(n)}(a)$ weakly converges to $Q(a)$ (with respect to the Cantor topology) for all $a \in 2^H$.*

Proof. Let P^* denote any fixpoint of $(\lambda Q. \text{skip} \ \& \ P \cdot Q)$ such that the measure $\mu_n = P^{(n)}(a)$ weakly converges to the measure $\mu = P^*(a)$, i.e. such that for all (Cantor-)continuous bounded functions $f : 2^H \rightarrow \mathbb{R}$

$$\lim_{n \rightarrow \infty} \int f d\mu_n = \int f d\mu$$

for all $a \in 2^H$. Let $\nu = Q(a)$. Fix an arbitrary Scott-open set V . Since 2^H is a Polish space under the Cantor topology, there exists an increasing chain of compact sets

$$C_1 \subseteq C_2 \subseteq \dots \subseteq V \quad \text{such that} \quad \sup_{n \in \mathbb{N}} \mu(C_n) = \mu(V).$$

By Urysohn's lemma (see [84, 145]), there exist continuous functions $f_n : 2^H \rightarrow [0, 1]$

such that $f_n(x) = 1$ for $x \in C_n$ and $f(x) = 0$ for $x \in \sim V$. We thus have

$$\begin{aligned}
\mu(C_n) &= \int \mathbf{1}_{C_n} d\mu \\
&\leq \int f_n d\mu && \text{by monotonicity of } \int \\
&= \lim_{m \rightarrow \infty} \int f_n d\mu_m && \text{by weak convergence} \\
&\leq \lim_{m \rightarrow \infty} \int \mathbf{1}_V d\mu_m && \text{by monotonicity of } \int \\
&= \lim_{m \rightarrow \infty} \mu_m(V) \\
&= \nu(V) && \text{by pointwise convergence on } \mathcal{O}
\end{aligned}$$

Taking the supremum over n , we get that $\mu(V) \leq \nu(V)$. Since ν is the \sqsubseteq -least fixpoint, the measures must therefore agree on V , which implies that they are equal by Theorem 3.5.4. Thus, any fixpoint of $(\lambda Q. \text{skip} \& P \cdot Q)$ with the weak convergence property must be equal to Q . But the fixpoint P^* defined in previous work *does* enjoy the weak convergence property, and therefore so does $Q = P^*$. \square

Proof of Lemma 3.7.6. Let A be a Borel set. Since we are in a Polish space, $\mu(A)$ is approximated arbitrarily closely from below by $\mu(C)$ for compact sets $C \subseteq A$ and from above by $\mu(U)$ for open sets $U \supseteq A$. By Urysohn's lemma (see [84, 145]), there exists a continuous function $f : D \rightarrow [0, 1]$ such that $f(a) = 1$ for all $a \in C$ and $f(a) = 0$ for all $a \notin U$. We thus have

$$\begin{aligned}
\mu(C) &= \int_{a \in C} f(a) \cdot \mu(da) \leq \int_{a \in D} f(a) \cdot \mu(da) \\
&= \int_{a \in U} f(a) \cdot \mu(da) \leq \mu(U), \\
\mu(C) &\leq \mu(A) \leq \mu(U),
\end{aligned}$$

thus

$$\left| \mu(A) - \int_{a \in D} f(a) \cdot \mu(da) \right| \leq \mu(U) - \mu(C),$$

and the right-hand side can be made arbitrarily small. \square

By Lemma 3.7.6, if P, Q are two Markov kernels and

$$\int_{a \in 2^H} f(a) \cdot P(c, da) = \int_{a \in 2^H} f(a) \cdot Q(c, da)$$

for all Cantor-continuous $f : 2^H \rightarrow [0, 1]$, then $P(c, -) = Q(c, -)$. If this holds for all $c \in 2^H$, then $P = Q$.

Proof of Theorem 3.7.5. Let $\varepsilon > 0$. Since all continuous functions on a compact space are uniformly continuous, for sufficiently large finite b and for all $a \subseteq b$, the value of f does not vary by more than ε on A_{ab} ; that is, $\sup_{c \in A_{ab}} f(c) - \inf_{c \in A_{ab}} f(c) < \varepsilon$. Then for any μ ,

$$\begin{aligned} & \int_{c \in A_{ab}} f(c) \cdot \mu(dc) - \int_{c \in A_{ab}} \inf_{c \in A_{ab}} f(c) \cdot \mu(dc) \\ & \leq \int_{c \in A_{ab}} (\sup_{c \in A_{ab}} f(c) - \inf_{c \in A_{ab}} f(c)) \cdot \mu(dc) < \varepsilon \cdot \mu(A_{ab}). \end{aligned}$$

Moreover,

$$\begin{aligned} (\bigsqcup A)(A_{ab}) &= \sum_{a \subseteq c \subseteq b} (-1)^{|c-a|} (\bigsqcup A)(B_c) \\ &= \sum_{a \subseteq c \subseteq b} (-1)^{|c-a|} \sup_{\mu \in A} \mu(B_c) \\ &= \lim_{\mu \in A} \sum_{a \subseteq c \subseteq b} (-1)^{|c-a|} \mu(B_c) = \lim_{\mu \in A} \mu(A_{ab}), \end{aligned}$$

so for sufficiently large $\mu \in A$, $\mu(A_{ab})$ does not differ from $(\bigsqcup A)(A_{ab})$ by more than $\varepsilon \cdot 2^{-|b|}$. Then for any constant $r \in [0, 1]$,

$$\begin{aligned} & \left| \int_{c \in A_{ab}} r \cdot (\bigsqcup A)(dc) - \int_{c \in A_{ab}} r \cdot \mu(dc) \right| \\ &= r \cdot |(\bigsqcup A)(A_{ab}) - \mu(A_{ab})| \\ &\leq |(\bigsqcup A)(A_{ab}) - \mu(A_{ab})| < \varepsilon \cdot 2^{-|b|}. \end{aligned}$$

Combining these observations,

$$\begin{aligned}
& \left| \int_{c \in 2^{\mathbb{H}}} f(c) \cdot (\bigsqcup A)(dc) - \int_{c \in 2^{\mathbb{H}}} f(c) \cdot \mu(dc) \right| \\
&= \left| \sum_{a \subseteq b} \int_{c \in A_{ab}} f(c) \cdot (\bigsqcup A)(dc) - \sum_{a \subseteq b} \int_{c \in A_{ab}} f(c) \cdot \mu(dc) \right| \\
&\leq \sum_{a \subseteq b} \left(\left| \int_{c \in A_{ab}} f(c) \cdot (\bigsqcup A)(dc) - \int_{c \in A_{ab}} \inf_{c \in A_{ab}} f(c) \cdot (\bigsqcup A)(dc) \right| \right. \\
&\quad + \left| \int_{c \in A_{ab}} \inf_{c \in A_{ab}} f(c) \cdot (\bigsqcup A)(dc) - \int_{c \in A_{ab}} \inf_{c \in A_{ab}} f(c) \cdot \mu(dc) \right| \\
&\quad \left. + \left| \int_{c \in A_{ab}} \inf_{c \in A_{ab}} f(c) \cdot \mu(dc) - \int_{c \in A_{ab}} f(c) \cdot \mu(dc) \right| \right) \\
&\leq \sum_{a \subseteq b} (\varepsilon \cdot (\bigsqcup A)(A_{ab}) + \varepsilon \cdot 2^{-|b|} + \varepsilon \cdot \mu(A_{ab})) \\
&= 3\varepsilon.
\end{aligned}$$

As $\varepsilon > 0$ was arbitrary,

$$\lim_{\mu \in A} \int_{c \in 2^{\mathbb{H}}} f(c) \cdot \mu(dc) = \int_{c \in 2^{\mathbb{H}}} f(c) \cdot (\bigsqcup A)(dc). \quad \square$$

Proof of Theorem 3.7.7. Consider the continuous transformation

$$T_P(Q) := \text{skip} \ \& \ P \cdot Q$$

on the DCPO of continuous Markov kernels. The continuity of T_P follows from Lemmas A.5.18 and A.5.20. The bottom element \perp is drop in this space, and

$$T_P(\perp) = \text{skip} = P^{(0)} \quad T_P(P^{(n)}) = \text{skip} \ \& \ P \cdot P^{(n)} = P^{(n+1)},$$

thus $T_P^{n+1}(\perp) = P^{(n)}$, so $\bigsqcup T_P^n(\perp) = \bigsqcup_n P^{(n)}$, and this is the least fixpoint of T_P . As shown in [44], P^{\otimes} is also a fixpoint of T_P , so it remains to show that $P^{\otimes} = \bigsqcup_n P^{(n)}$.

Let $c \in 2^{\mathbb{H}}$. As shown in [44], the measures $P^{(n)}(c, -)$ converge weakly to $P^{\otimes}(c, -)$; that is, for any Cantor-continuous function $f : 2^{\mathbb{H}} \rightarrow [0, 1]$, the expected values of f relative to $P^{(n)}$ converge to the expected value of f relative to P^{\otimes} :

$$\lim_n \int f(a) \cdot P^{(n)}(c, da) = \int f(a) \cdot P^{\otimes}(c, da).$$

But by Theorem 3.7.5, we also have

$$\lim_n \int f(a) \cdot P^{(n)}(c, da) = \int f(a) \cdot (\bigsqcup_n P^{(n)})(c, da),$$

thus

$$\int f(a) \cdot P^{\otimes}(c, da) = \int f(a) \cdot (\bigsqcup_n P^{(n)})(c, da).$$

As f was arbitrary, we have $P^{\otimes}(c, -) = (\bigsqcup_n P^{(n)})(c, -)$ by Lemma 3.7.6, and as c was arbitrary, we have $P^{\otimes} = \bigsqcup_n P^{(n)}$. \square

A.6 Approximation and Discrete Measures

This section contains the proofs of Section 3.8. We need the following auxiliary lemma to prove Theorem 3.8.3.

Lemma A.6.1.

- (i) For any Borel set B , $(\mu \upharpoonright b)(B) = \mu(\{c \mid c \cap b \in B\})$.
- (ii) $(\mu \upharpoonright b) \upharpoonright d = \mu \upharpoonright (b \cap d)$.
- (iii) If $a, b \in \wp_\omega(H)$ and $a \subseteq b$, then $\mu \upharpoonright a \sqsubseteq \mu \upharpoonright b \sqsubseteq \mu$.
- (iv) $\mu \sqsubseteq \delta_b$ iff $\mu = \mu \upharpoonright b$.
- (v) The function $\mu \mapsto \mu \upharpoonright b$ is continuous.

Proof. (i)

$$\begin{aligned} (\mu \upharpoonright b)(B) &= \sum_{a \subseteq b} \mu(A_{ab}) \delta_a(B) = \sum_{a \subseteq b} \mu(\{c \mid c \cap b = a\}) [a \in B] \\ &= \sum_{\substack{a \subseteq b \\ a \in B}} \mu(\{c \mid c \cap b = a\}) = \mu\left(\bigcup_{\substack{a \subseteq b \\ a \in B}} \{c \mid c \cap b = a\}\right) \\ &= \mu(\{c \mid c \cap b \in B\}). \end{aligned}$$

(ii) For any Borel set B ,

$$\begin{aligned}
((\mu \upharpoonright b) \upharpoonright d)(B) &= (\mu \upharpoonright b)(\{c \mid c \cap d \in B\}) \\
&= \mu(\{c \mid c \cap b \in \{c \mid c \cap d \in B\}\}) \\
&= \mu(\{c \mid c \cap b \cap d \in B\}) \\
&= (\mu \upharpoonright (b \cap d))(B).
\end{aligned}$$

(iii) If $a \subseteq b$, then for any up-closed Borel set B ,

$$\begin{aligned}
\{c \mid c \cap a \in B\} &\subseteq \{c \mid c \cap b \in B\} \subseteq B, \\
\mu(\{c \mid c \cap a \in B\}) &\leq \mu(\{c \mid c \cap b \in B\}) \leq \mu(B), \\
(\mu \upharpoonright a)(B) &\leq (\mu \upharpoonright b)(B) \leq \mu(B).
\end{aligned}$$

As this holds for all $B \in \mathcal{O}$, we have $\mu \upharpoonright a \sqsubseteq \mu \upharpoonright b \sqsubseteq \mu$.

(iv) First we show that $\mu \upharpoonright b \sqsubseteq \delta_b$. For any up-closed Borel set B ,

$$\begin{aligned}
(\mu \upharpoonright b)(B) &= \sum_{a \subseteq b} \mu(A_{ab})[a \in B] \\
&\leq \sum_{a \subseteq b} \mu(A_{ab})[b \in B] = [b \in B] = \delta_b(B).
\end{aligned}$$

Now we show that if $\mu \sqsubseteq \delta_b$, then $\mu = \mu \upharpoonright b$. From

$$d \subseteq b \wedge d \subseteq c \Leftrightarrow d \subseteq c \cap b \qquad c \in B_d \Leftrightarrow d \subseteq c$$

we have

$$(\exists d \in F \ d \subseteq b \wedge c \in B_d) \Leftrightarrow (\exists d \in F \ c \cap b \in B_d)$$

$$c \in \bigcup_{\substack{d \in F \\ d \subseteq b}} B_d \Leftrightarrow c \cap b \in \bigcup_{d \in F} B_d$$

$$(\mu \upharpoonright b)\left(\bigcup_{d \in F} B_d\right) = \mu(\{c \mid c \cap b \in \bigcup_{d \in F} B_d\}) = \mu\left(\bigcup_{\substack{d \in F \\ d \subseteq b}} B_d\right). \tag{A.8}$$

Now if $\mu \sqsubseteq \delta_b$, then

$$\mu(\bigsqcup_{\substack{d \in F \\ d \sqsubseteq b}} B_d) \leq \delta_b(\bigsqcup_{\substack{d \in F \\ d \sqsubseteq b}} B_d) = [b \in \bigsqcup_{\substack{d \in F \\ d \sqsubseteq b}} B_d] = 0,$$

so

$$\mu(\bigsqcup_{d \in F} B_d) \leq \mu(\bigsqcup_{\substack{d \in F \\ d \subseteq b}} B_d) + \mu(\bigsqcup_{\substack{d \in F \\ d \not\subseteq b}} B_d) = \mu(\bigsqcup_{d \in F} B_d).$$

Combining this with (A.8), we have that μ and $\mu \upharpoonright b$ agree on all $B \in \mathcal{O}$, therefore they agree everywhere.

(v) If $\mu \sqsubseteq \nu$, then for all $B \in \mathcal{O}$,

$$\begin{aligned} (\mu \upharpoonright b)(B) &= \mu(\{c \mid c \cap b \in B\}) \\ &\leq \nu(\{c \mid c \cap b \in B\}) = (\nu \upharpoonright b)(B). \end{aligned}$$

Also, for any directed set D of measures and $B \in \mathcal{O}$,

$$\begin{aligned} ((\bigsqcup D) \upharpoonright b)(B) &= (\bigsqcup D)(\{c \mid c \cap b \in B\}) \\ &= \sup_{\mu \in D} \mu(\{c \mid c \cap b \in B\}) = \sup_{\mu \in D} (\mu \upharpoonright b)(B) \\ &= (\bigsqcup_{\mu \in D} (\mu \upharpoonright b))(B), \end{aligned}$$

therefore $(\bigsqcup D) \upharpoonright b = \bigsqcup_{\mu \in D} (\mu \upharpoonright b)$. □

Proof of Theorem 3.8.3. The set $\{\mu \upharpoonright b \mid b \in \wp_\omega(H)\}$ is a directed set below μ by Lemma A.6.1(iii), and for any up-closed Borel set B ,

$$\begin{aligned} (\bigsqcup_{b \in \wp_\omega(H)} \mu \upharpoonright b)(B) &= \sup_{b \in \wp_\omega(H)} \mu(\{c \mid c \cap b \in B\}) \\ &= \mu(\bigcup_{b \in \wp_\omega(H)} \{c \mid c \cap b \in B\}) = \mu(B). \end{aligned}$$

An approximating set for μ is the set

$$L = \left\{ \sum_{a \subseteq b} r_a \delta_a \mid b \in \wp_\omega(H), r_a < \mu(A_{ab}) \text{ for all } a \neq \emptyset \right\}.$$

If L is empty, then $\mu(A_{\emptyset b}) = 1$ for all finite b , in which case $\mu = \delta_{\emptyset}$ and there is nothing to prove. Otherwise, L is a nonempty directed set whose supremum is μ .

Now we show that $\nu \ll \mu$ for any $\nu \in L$. Suppose D is a directed set and $\mu \sqsubseteq \bigsqcup D$. By Lemma A.6.1(iii) and (v),

$$\mu \upharpoonright b \sqsubseteq \left(\bigsqcup D \right) \upharpoonright b = \bigsqcup_{\rho \in D} \rho \upharpoonright b.$$

Moreover, for any $B \in \mathcal{O}$, $B \neq B_{\emptyset}$, and $\sum_{a \subseteq b} r_a \delta_a \in L$,

$$\begin{aligned} (\nu \upharpoonright b)(B) &= \sum_{a \in B} \nu(A_{ab})[a \in B] \\ &< \sum_{a \in B} \mu(A_{ab})[a \in B] = (\mu \upharpoonright b)(B). \end{aligned}$$

Then $\nu(B_{\emptyset}) = \rho(B_{\emptyset}) = 1$ for all $\rho \in D$, and for any $B \in \mathcal{O}$, $B \neq B_{\emptyset}$,

$$(\nu \upharpoonright b)(B) < (\mu \upharpoonright b)(B) \leq \sup_{\rho \in D} (\rho \upharpoonright b)(B) \tag{A.9}$$

so there exists $\rho \in D$ such that $(\nu \upharpoonright b)(B) \leq (\rho \upharpoonright b)(B)$. But since B can intersect 2^H in only finitely many ways and D is directed, a single $\rho \in D$ can be found such that (A.9) holds uniformly for all $B \in \mathcal{O}$, $B \neq B_{\emptyset}$. Then $\nu \upharpoonright b \sqsubseteq \rho \in D$. \square

Proof of Corollary 3.8.4. Let $f : 2^H \rightarrow 2^H$ map a to $a \cap b$. This is a continuous function that gives rise to a deterministic kernel. Then for any $B \in \mathcal{O}$,

$$\begin{aligned} (P \cdot b)(a, B) &= P(a, f^{-1}(B)) = P(a, \{c \mid c \cap b \in B\}) \\ &= (P(a, -) \upharpoonright b)(B). \end{aligned} \quad \square$$

Chapter B

Appendix to Chapter 4

B.1 ProbNetKAT Denotational Semantics for History-free Fragment

In the original ProbNetKAT language (Chapter 3), programs manipulate sets of *packet histories*—non-empty, finite sequences of packets modeling trajectories through the network [44, 165]. The resulting state space is uncountable and modeling the semantics properly requires full-blown measure theory as some programs generate continuous distributions (Chapter 3). In the history-free fragment considered in Chapter 4, programs manipulate sets of packets and the state space is finite, permitting a much simpler semantics using only discrete probability measures (Figure B.1). This section is concerned with formalizing this simpler semantics. We will then prove (in Appendix B.2) that the simpler semantics is equivalent to the original semantics from Chapter 3 in the following sense:

Proposition B.1.1. *Let p be a dup-free program, let $\llbracket p \rrbracket \in 2^H \rightarrow \mathcal{D}(2^H)$ denote the semantics defined in Chapter 3, and let $\llbracket p \rrbracket \in 2^{P^k} \rightarrow \mathcal{D}(2^{P^k})$ denote the semantics defined in Figure B.1. Then for all inputs $a \in 2^{P^k}$, we have $\llbracket p \rrbracket(a) = \llbracket p \rrbracket(a)$, where we identify packets and histories of length one.*

Before we turn to proving this claim in Appendix B.2, we formalize the discrete semantics in detail.

Semantics $\boxed{\llbracket p \rrbracket \in 2^{\text{Pk}} \rightarrow \mathcal{D}(2^{\text{Pk}})}$

$$\begin{aligned} \llbracket \text{drop} \rrbracket(a) &:= \delta(\emptyset) \\ \llbracket \text{skip} \rrbracket(a) &:= \delta(a) \\ \llbracket f = n \rrbracket(a) &:= \delta(\{\pi \in a \mid \pi.f = n\}) \\ \llbracket f \leftarrow n \rrbracket(a) &:= \delta(\{\pi[f := n] \mid \pi \in a\}) \\ \llbracket \neg t \rrbracket(a) &:= \mathcal{D}(\lambda b. a - b)(\llbracket t \rrbracket(a)) \\ \llbracket p \ \& \ q \rrbracket(a) &:= \mathcal{D}(\cup)(\llbracket p \rrbracket(a) \times \llbracket q \rrbracket(a)) \\ \llbracket p \cdot q \rrbracket(a) &:= \llbracket q \rrbracket^\dagger(\llbracket p \rrbracket(a)) \\ \llbracket p \oplus_r q \rrbracket(a) &:= r \cdot \llbracket p \rrbracket(a) + (1 - r) \cdot \llbracket q \rrbracket(a) \\ \llbracket p^* \rrbracket(a) &:= \bigsqcup_{n \in \mathbb{N}} \llbracket p^{(n)} \rrbracket(a) \end{aligned}$$

where $p^{(0)} := \text{skip}$, $p^{(n+1)} := \text{skip} \ \& \ p \cdot p^{(n)}$

(Discrete) Probability Monad \mathcal{D}

$$\begin{aligned} \text{Unit} \quad \delta &: X \rightarrow \mathcal{D}(X) \quad \delta(x) := \delta_x \\ \text{Bind} \quad -^\dagger &: (X \rightarrow \mathcal{D}(Y)) \rightarrow \mathcal{D}(X) \rightarrow \mathcal{D}(Y) \\ &f^\dagger(\mu)(A) := \sum_{x \in X} f(x)(A) \cdot \mu(x) \end{aligned}$$

Figure B.1: Discrete ProbNetKAT semantics for history-free fragment (without dup).

We work in the discrete space 2^{Pk} , *i.e.*, the set of sets of packets. An *outcome* (denoted by lowercase variables a, b, c, \dots) is a set of packets and an *event* (denoted by uppercase variables A, B, C, \dots) is a set of outcomes. Given a discrete probability measure on this space, the probability of an event is the sum of the probabilities of its outcomes.

ProbNetKAT programs are interpreted as *Markov kernels* on the space 2^{Pk} . A Markov kernel is a function $2^{\text{Pk}} \rightarrow \mathcal{D}(2^{\text{Pk}})$ where \mathcal{D} is the probability (or Giry) monad [52, 87]. Thus, a program p maps an input set of packets $a \in 2^{\text{Pk}}$ to a *distribution* $\llbracket p \rrbracket(a) \in \mathcal{D}(2^{\text{Pk}})$ over output sets of packets. The semantics uses the following probabilistic constructions:

- For a discrete measurable space X , $\mathcal{D}(X)$ denotes the set of probability measures over X ; that is, the set of countably additive functions $\mu: 2^X \rightarrow [0, 1]$ with $\mu(X) = 1$.
- For a measurable function $f: X \rightarrow Y$, $\mathcal{D}(f): \mathcal{D}(X) \rightarrow \mathcal{D}(Y)$ denotes the *pushforward* along f ; that is, the function that maps a measure μ on X to

$$\mathcal{D}(f)(\mu) := \mu \circ f^{-1} = \lambda A \in \Sigma_Y. \mu(\{x \in X \mid f(x) \in A\})$$

which is called the *pushforward measure* on Y .

- The *unit* $\delta: X \rightarrow \mathcal{D}(X)$ of the monad maps a point $x \in X$ to the point mass (or *Dirac measure*) $\delta_x \in \mathcal{D}(X)$. The Dirac measure is given by

$$\delta_x(A) := [x \in A]$$

That is, the Dirac measure is 1 if $x \in A$ and 0 otherwise.

- The *bind* operation of the monad,

$$-\dagger: (X \rightarrow \mathcal{D}(Y)) \rightarrow \mathcal{D}(X) \rightarrow \mathcal{D}(Y),$$

lifts a function $f: X \rightarrow \mathcal{D}(Y)$ with deterministic inputs to a function $f^\dagger: \mathcal{D}(X) \rightarrow \mathcal{D}(Y)$ that takes random inputs. Intuitively, this is achieved by averaging the output of f when the inputs are randomly distributed according to μ . Formally,

$$f^\dagger(\mu)(A) := \sum_{x \in X} f(x)(A) \cdot \mu(x).$$

- Given two measures $\mu \in \mathcal{D}(X)$ and $\nu \in \mathcal{D}(Y)$, $\mu \times \nu \in \mathcal{D}(X \times Y)$ denotes their *product measure*. This is the unique measure satisfying

$$(\mu \times \nu)(A \times B) = \mu(A) \cdot \nu(B)$$

Intuitively, it models distributions over pairs of independent values.

Using these primitives, we can now make our operational intuitions precise (see Figure B.1 for formal definitions). A predicate t maps the set of input packets $a \in 2^{\text{Pk}}$ to the subset of packets $b \subseteq a$ satisfying the predicate (with probability 1). Hence, *drop* drops all packets (i.e., it returns the empty set) while *skip* keeps all packets (i.e., it returns the input set). The test $f=n$ returns the subset of input packets whose f -field is n . Negation $\neg t$ filters out the packets returned by t .

Parallel composition $p \& q$ executes p and q independently on the input set, then returns the union of their results. Note that packet sets do *not* model nondeterminism,

unlike the usual situation in Kleene algebras—rather, they model collections of packets traversing possibly different portions of the network simultaneously. In particular, the union operation is *not* idempotent: $p \& p$ need not have the same semantics as p . Probabilistic choice $p \oplus_r q$ feeds the input to both p and q and returns a convex combination of the output distributions according to r . Sequential composition $p \cdot q$ can be thought of as a two-stage probabilistic process: it first executes p on the input set to obtain a random intermediate result, then feeds that into q to obtain the final distribution over outputs. The outcome of q is averaged over the distribution of intermediate results produced by p .

We say that two programs are *equivalent*, denoted $p \equiv q$, if they denote the same Markov kernel, *i.e.* if $\llbracket p \rrbracket = \llbracket q \rrbracket$. As usual, we expect Kleene star p^* to satisfy the characteristic fixed point equation $p^* \equiv \text{skip} \& p \cdot p^*$, which allows it to be unrolled ad infinitum. Thus we define it as the supremum of its finite unrollings $p^{(n)}$; see Figure B.1. This supremum is taken in a CPO $(\mathcal{D}(2^{\text{Pk}}), \sqsubseteq)$ of distributions that is described in more detail in the next section (Appendix B.1.1). The partial ordering \sqsubseteq on packet set distributions gives rise to a partial ordering on programs: we write $p \leq q$ iff $\llbracket p \rrbracket(a) \sqsubseteq \llbracket q \rrbracket(a)$ for all inputs $a \in 2^{\text{Pk}}$. Intuitively, $p \leq q$ iff p produces any particular output packet π with probability at most that of q for any fixed input— q has a larger probability of delivering more output packets.

B.1.1 The CPO $(\mathcal{D}(2^{\text{Pk}}), \sqsubseteq)$

The space 2^{Pk} with the subset order forms a CPO $(2^{\text{Pk}}, \subseteq)$. Following Saheb-Djahromi [151], this CPO can be lifted to a CPO $(\mathcal{D}(2^{\text{Pk}}), \sqsubseteq)$ on distributions over 2^{Pk} . Because 2^{Pk} is a finite space, the resulting ordering \sqsubseteq on distributions takes a particularly easy form:

$$\mu \sqsubseteq \nu \iff \mu(\{a\}^\uparrow) \leq \nu(\{a\}^\uparrow) \text{ for all } a \subseteq \text{Pk}$$

where $\{a\}^\uparrow := \{b \mid a \subseteq b\}$ denotes upward closure. Intuitively, the probability of observing a particular packet in a set sampled from μ is smaller than if sampling from ν . As shown in Chapter 3, ProbNetKAT satisfies various monotonicity (and continuity) properties with respect to this ordering, including

$$a \subseteq a' \implies \llbracket p \rrbracket(a) \sqsubseteq \llbracket p \rrbracket(a') \quad \text{and} \quad n \leq m \implies \llbracket p^{(n)} \rrbracket(a) \sqsubseteq \llbracket p^{(m)} \rrbracket(a).$$

As a result, the semantics of p^* as the supremum of its finite unrollings $p^{(n)}$ is well-defined.

While the semantics of full ProbNetKAT requires more domain theory to give a satisfactory characterization of Kleene star, a simpler characterization suffices for the history-free fragment.

Lemma B.1.2 (Pointwise Convergence). *Let $A \subseteq 2^{\text{Pk}}$. Then for all programs p and inputs $a \in 2^{\text{Pk}}$,*

$$\llbracket p^* \rrbracket(a)(A) = \lim_{n \rightarrow \infty} \llbracket p^{(n)} \rrbracket(a)(A).$$

B.2 Omitted Proofs

Lemma B.2.1. *Let A be a finite boolean combination of basic open sets, i.e. sets of the form $B_a = \{a\}^\uparrow$ for $a \in \wp_\omega(\text{H})$, and let $\llbracket - \rrbracket$ denote the semantics from Chapter 3. Then for all programs p and inputs $a \in 2^{\text{H}}$,*

$$\llbracket p^* \rrbracket(a)(A) = \lim_{n \rightarrow \infty} \llbracket p^{(n)} \rrbracket(a)(A)$$

Proof. Using topological arguments, the claim follows directly from previous results: A is a Cantor-clopen set by Chapter 3 (i.e., both A and \bar{A} are Cantor-open), so its indicator function $\mathbf{1}_A$ is Cantor-continuous. But $\mu_n := \llbracket p^{(n)} \rrbracket(a)$ converges weakly to $\mu := \llbracket p^* \rrbracket(a)$ in the Cantor topology [44, Theorem 4], so

$$\lim_{n \rightarrow \infty} \llbracket p^{(n)} \rrbracket(a)(A) = \lim_{n \rightarrow \infty} \int \mathbf{1}_A d\mu_n = \int \mathbf{1}_A d\mu = \llbracket p^* \rrbracket(a)(A)$$

(To see why A and \overline{A} are open in the Cantor topology, note that they can be written in disjunctive normal form over atoms $B_{\{h\}}$.) \square

Predicates in ProbNetKAT form a Boolean algebra.

Lemma B.2.2. *Every predicate t satisfies $\llbracket t \rrbracket(a) = \delta_{a \cap b_t}$, for $b_t \subseteq \text{Pk}$ chosen as follows:*

$$\begin{array}{lll} b_{\text{drop}} := \emptyset & b_{t \& u} := b_t \cup b_u & b_{f=n} := \{\pi \in \text{Pk} \mid \pi.f = n\} \\ b_{\text{skip}} := \text{Pk} & b_{t \cdot u} := b_t \cap b_u & b_{\neg t} := \text{Pk} - b_t \end{array}$$

Proof. For drop, skip, and $f=n$, the claim holds trivially. For $\neg t$, $t \& u$, and $t \cdot u$, the claim follows inductively, using that $\mathcal{D}(f)(\delta_b) = \delta_{f(b)}$, $\delta_b \times \delta_c = \delta_{(b,c)}$, and that $f^\dagger(\delta_b) = f(b)$. The first and last equations hold because $\langle \mathcal{D}, \delta, -^\dagger \rangle$ is a monad. \square

Proof of Proposition B.1.1. We only need to show that for dup-free programs p and history-free inputs $a \in 2^{\text{Pk}}$, $\llbracket p \rrbracket(a)$ is a distribution on packets (where we identify packets and singleton histories). We proceed by structural induction on p . All cases are straightforward except perhaps the case of p^* . For this case, by the induction hypothesis, all $\llbracket p^{(n)} \rrbracket(a)$ are discrete probability distributions on packet sets, therefore vanish outside 2^{Pk} . By Lemma B.2.1, this is also true of the limit $\llbracket p^* \rrbracket(a)$, as its value on 2^{Pk} must be 1, therefore it is also a discrete distribution on packet sets. \square

Proof of Lemma B.1.2. This follows directly from Lemma B.2.1 and Proposition B.1.1 by noticing that any set $A \subseteq 2^{\text{Pk}}$ is a finite boolean combination of basic open sets. \square

Proof of Theorem 4.3.1. It suffices to show the equality $\mathcal{B}\llbracket p \rrbracket_{ab} = \llbracket p \rrbracket(a)(\{b\})$; the remaining claims then follow by well-definedness of $\llbracket - \rrbracket$. The equality is shown using Lemma B.1.2 and a routine induction on p :

For $p = \text{drop}$, skip, $f=n$, $f \leftarrow n$ we have

$$\llbracket p \rrbracket(a)(\{b\}) = \delta_c(\{b\}) = [b = c] = \mathcal{B}\llbracket p \rrbracket_{ab}$$

for $c = \emptyset, a, \{\pi \in a \mid \pi.f = n\}, \{\pi[f := n] \mid \pi \in a\}$, respectively.

For $\neg t$ we have,

$$\begin{aligned}
\mathcal{B}[\neg t]_{ab} &= [b \subseteq a] \cdot \mathcal{B}[t]_{a, a-b} \\
&= [b \subseteq a] \cdot \llbracket t \rrbracket(a)(\{a - b\}) && \text{(IH)} \\
&= [b \subseteq a] \cdot [a - b = a \cap b_t] && \text{(Lemma B.2.2)} \\
&= [b \subseteq a] \cdot [a - b = a - (H - b_t)] \\
&= [b = a \cap (H - b_t)] \\
&= \llbracket \neg t \rrbracket(a)(b) && \text{(Lemma B.2.2)}
\end{aligned}$$

For $p \& q$, letting $\mu = \llbracket p \rrbracket(a)$ and $\nu = \llbracket q \rrbracket(a)$ we have

$$\begin{aligned}
\llbracket p \& q \rrbracket(a)(\{b\}) &= (\mu \times \nu)(\{(b_1, b_2) \mid b_1 \cup b_2 = b\}) \\
&= \sum_{b_1, b_2} [b_1 \cup b_2 = b] \cdot (\mu \times \nu)(\{(b_1, b_2)\}) \\
&= \sum_{b_1, b_2} [b_1 \cup b_2 = b] \cdot \mu(\{b_1\}) \cdot \nu(\{b_2\}) \\
&= \sum_{b_1, b_2} [b_1 \cup b_2 = b] \cdot \mathcal{B}[p]_{ab_1} \cdot \mathcal{B}[q]_{ab_2} && \text{(IH)} \\
&= \mathcal{B}[p \& q]_{ab}
\end{aligned}$$

where we use in the second step that $b \subseteq \text{Pk}$ is finite, thus $\{(b_1, b_2) \mid b_1 \cup b_2 = b\}$ is finite.

For $p \cdot q$, let $\mu = \llbracket p \rrbracket(a)$ and $\nu_c = \llbracket q \rrbracket(c)$ and recall that μ is a discrete distribution on 2^{Pk} . Thus

$$\begin{aligned}
\llbracket p \cdot q \rrbracket(a)(\{b\}) &= \sum_{c \in 2^{\text{Pk}}} \nu_c(\{b\}) \cdot \mu(\{c\}) \\
&= \sum_{c \in 2^{\text{Pk}}} \mathcal{B}[q]_{c, b} \cdot \mathcal{B}[p]_{a, c} \\
&= \mathcal{B}[p \cdot q]_{ab}.
\end{aligned}$$

For $p \oplus_r q$, the claim follows directly from the induction hypotheses.

Finally, for p^* , we know that $\mathcal{B}[p^{(n)}]_{ab} = \llbracket p^{(n)} \rrbracket(a)(\{b\})$ by induction hypothesis. The key to proving the claim is Lemma B.1.2, which allows us to take the limit on both sides and deduce

$$\mathcal{B}[p^*]_{ab} = \lim_{n \rightarrow \infty} \mathcal{B}[p^{(n)}]_{ab} = \lim_{n \rightarrow \infty} \llbracket p^{(n)} \rrbracket(a)(\{b\}) = \llbracket p^* \rrbracket(a)(\{b\}). \quad \square$$

Proof of Lemma 4.4.1. For arbitrary $a, b \subseteq \text{Pk}$, we have

$$\begin{aligned}
\sum_{a', b'} \mathcal{S}[[p]]_{(a,b),(a',b')} &= \sum_{a', b'} [b' = a \cup b] \cdot \mathcal{B}[[p]]_{a,a'} \\
&= \sum_{a'} \left(\sum_{b'} [b' = a \cup b] \right) \cdot \mathcal{B}[[p]]_{a,a'} \\
&= \sum_{a'} \mathcal{B}[[p]]_{a,a'} = 1
\end{aligned}$$

where in the last step, we use that $\mathcal{B}[[p]]$ is stochastic (Theorem 4.3.1). □

Proof of Lemma 4.4.3. By induction on $n \geq 0$. For $n = 0$, we have

$$\begin{aligned}
\sum_{a'} [b' = a' \cup b] \cdot \mathcal{B}[[p^{(n)}]]_{a,a'} &= \sum_{a'} [b' = a' \cup b] \cdot \mathcal{B}[[\text{skip}]]_{a,a'} \\
&= \sum_{a'} [b' = a' \cup b] \cdot [a = a'] \\
&= [b' = a \cup b] \\
&= [b' = a \cup b] \cdot \sum_{a'} \mathcal{B}[[p]]_{a,a'} \\
&= \sum_{a'} \mathcal{S}[[p]]_{(a,b),(a',b')}
\end{aligned}$$

In the induction step ($n > 0$),

$$\begin{aligned}
& \sum_{a'} [b' = a' \cup b] \cdot \mathcal{B}[[p^{(n)}]]_{a,a'} \\
&= \sum_{a'} [b' = a' \cup b] \cdot \mathcal{B}[\text{skip} \ \& \ p \cdot p^{(n-1)}]_{a,a'} \\
&= \sum_{a'} [b' = a' \cup b] \cdot \sum_c [a' = a \cup c] \cdot \mathcal{B}[p \cdot p^{(n-1)}]_{a,c} \\
&= \sum_c \left(\sum_{a'} [b' = a' \cup b] \cdot [a' = a \cup c] \right) \cdot \sum_k \mathcal{B}[[p]]_{a,k} \cdot \mathcal{B}[[p^{(n-1)}]]_{k,c} \\
&= \sum_{c,k} [b' = a \cup c \cup b] \cdot \mathcal{B}[[p]]_{a,k} \cdot \mathcal{B}[[p^{(n-1)}]]_{k,c} \\
&= \sum_k \mathcal{B}[[p]]_{a,k} \cdot \sum_{a'} [b' = a' \cup (a \cup b)] \cdot \mathcal{B}[[p^{(n-1)}]]_{k,a'} \\
&= \sum_k \mathcal{B}[[p]]_{a,k} \cdot \sum_{a'} \mathcal{S}[[p]]_{(k,a \cup b),(a',b')}^n \\
&= \sum_{a'} \sum_{k_1, k_2} [k_2 = a \cup b] \cdot \mathcal{B}[[p]]_{a,k_1} \cdot \mathcal{S}[[p]]_{(k_1, k_2),(a',b')}^n \\
&= \sum_{a'} \sum_{k_1, k_2} \mathcal{S}[[p]]_{(a,b)(k_1, k_2)} \cdot \mathcal{S}[[p]]_{(k_1, k_2),(a',b')}^n \\
&= \sum_{a'} \mathcal{S}[[p]]_{(a,b),(a',b')}^{n+1} \quad \square
\end{aligned}$$

Lemma B.2.3. *The matrix $X = I - Q$ in Equation (4.2) of Section 4.4 is invertible.*

Proof. Let S be a finite set of states, $|S| = n$, M an $S \times S$ substochastic matrix ($M_{st} \geq 0$, $M\mathbf{1} \leq \mathbf{1}$). A state s is *defective* if $(M\mathbf{1})_s < 1$. We say M is *stochastic* if $M\mathbf{1} = \mathbf{1}$, *irreducible* if $(\sum_{i=0}^{n-1} M^i)_{st} > 0$ (that is, the support graph of M is strongly connected), and *aperiodic* if all entries of some power of M are strictly positive.

We show that if M is substochastic such that every state can reach a defective state via a path in the support graph, then the spectral radius of M is strictly less than 1. Intuitively, all weight in the system eventually drains out at the defective states.

Let e_s , $s \in S$, be the standard basis vectors. As a distribution, e_s^T is the unit point mass on s . For $A \subseteq S$, let $e_A = \sum_{s \in A} e_s$. The L_1 -norm of a substochastic vector is its

total weight as a distribution. Multiplying on the right by M never increases total weight, but will strictly decrease it if there is nonzero weight on a defective state. Since every state can reach a defective state, this must happen after n steps, thus $\|e_s^T M^n\|_1 < 1$. Let $c = \max_s \|e_s^T M^n\|_1 < 1$. For any $y = \sum_s a_s e_s$,

$$\begin{aligned} \|y^T M^n\|_1 &= \left\| \left(\sum_s a_s e_s \right)^T M^n \right\|_1 \\ &\leq \sum_s |a_s| \cdot \|e_s^T M^n\|_1 \leq \sum_s |a_s| \cdot c = c \cdot \|y^T\|_1. \end{aligned}$$

Then M^n is contractive in the L_1 norm, so $|\lambda| < 1$ for all eigenvalues λ . Thus $I - M$ is invertible because 1 is not an eigenvalue of M . \square

Proof of Proposition 4.4.6.

1. It suffices to show that $USU = SU$. Suppose that

$$\Pr[(a, b) \xrightarrow{USU}_1 (a', b')] = p > 0.$$

It suffices to show that this implies

$$\Pr[(a, b) \xrightarrow{SU}_1 (a', b')] = p.$$

If (a, b) is saturated, then we must have $(a', b') = (\emptyset, b)$ and

$$\Pr[(a, b) \xrightarrow{USU}_1 (\emptyset, b)] = 1 = \Pr[(a, b) \xrightarrow{SU}_1 (\emptyset, b)]$$

If (a, b) is not saturated, then $(a, b) \xrightarrow{U}_1 (a, b)$ with probability 1 and therefore

$$\Pr[(a, b) \xrightarrow{USU}_1 (a', b')] = \Pr[(a, b) \xrightarrow{SU}_1 (a', b')]$$

2. Since S and U are stochastic, clearly SU is a MC. Since SU is finite state, any state can reach an absorbing communication class. (To see this, note that the reachability relation \xrightarrow{SU} induces a partial order on the communication classes of SU . Its maximal elements are necessarily absorbing, and they must exist because the state space is finite.) It thus suffices to show that a state set $C \subseteq 2^{\text{Pk}} \times 2^{\text{Pk}}$ in SU is an absorbing communication class iff $C = \{(\emptyset, b)\}$ for some $b \subseteq \text{Pk}$.

“ \Leftarrow ”: Observe that $\emptyset \xrightarrow{B}_1 a'$ iff $a' = \emptyset$. Thus $(\emptyset, b) \xrightarrow{S}_1 (a', b')$ iff $a' = \emptyset$ and $b' = b$, and likewise $(\emptyset, b) \xrightarrow{U}_1 (a', b')$ iff $a' = \emptyset$ and $b' = b$. Thus (\emptyset, b) is an absorbing state in SU as required.

“ \Rightarrow ”: First observe that by monotonicity of SU (Lemma 4.4.5), we have $b = b'$ whenever $(a, b) \xleftrightarrow{SU} (a', b')$; thus there exists a fixed b_C such that $(a, b) \in C$ implies $b = b_C$.

Now pick an arbitrary state $(a, b_C) \in C$. It suffices to show that $(a, b_C) \xrightarrow{SU} (\emptyset, b_C)$, because that implies $(a, b_C) \xleftrightarrow{SU} (\emptyset, b_C)$, which in turn implies $a = \emptyset$. But the choice of $(a, b_C) \in C$ was arbitrary, so that would mean $C = \{(\emptyset, b_C)\}$ as claimed.

To show that $(a, b_C) \xrightarrow{SU} (\emptyset, b_C)$, pick arbitrary states such that

$$(a, b_C) \xrightarrow{S} (a', b') \xrightarrow{U}_1 (a'', b'')$$

and recall that this implies $(a, b_C) \xrightarrow{SU} (a'', b'')$ by claim (1). Then $(a'', b'') \xrightarrow{SU} (a, b_C)$ because C is absorbing, and thus $b_C = b' = b''$ by monotonicity of S , U , and SU . But (a', b') was chosen as an arbitrary state S -reachable from (a, b_C) , so (a, b) and by transitivity (a', b') must be saturated. Thus $a'' = \emptyset$ by the definition of U . \square

Proof of Theorem 4.4.7. Using Proposition 4.4.6.1 in the second step and equation (4.3) in the last step,

$$\begin{aligned} \lim_{n \rightarrow \infty} \sum_{a'} S_{(a,b),(a',b')}^n &= \lim_{n \rightarrow \infty} \sum_{a'} (S^n U)_{(a,b),(a',b')} \\ &= \lim_{n \rightarrow \infty} \sum_{a'} (SU)_{(a,b),(a',b')}^n \\ &= \sum_{a'} (SU)_{(a,b),(a',b')}^\infty = (SU)_{(a,b),(\emptyset,b')}^\infty \end{aligned}$$

$(SU)^\infty$ is computable because S and U are matrices over \mathbb{Q} and hence so is $(I - Q)^{-1}R$. \square

Corollary B.2.4. For programs p and q , it is decidable whether $p \equiv q$.

Proof of Corollary B.2.4. Recall from Corollary 4.3.2 that it suffices to compute the finite rational matrices $\mathcal{B}[[p]]$ and $\mathcal{B}[[q]]$ and check them for equality. But Theorem 4.4.7 together with Proposition 4.4.2 gives us an effective mechanism to compute $\mathcal{B}[[\text{---}]]$ in the case of Kleene star, and $\mathcal{B}[[\text{---}]]$ is straightforward to compute in all other cases. Summarizing the full chain of equalities, we have:

$$[[p^*]](a)(\{b\}) = \mathcal{B}[[p^*]]_{a,b} = \lim_{n \rightarrow \infty} \mathcal{B}[[p^{(n)}]]_{a,b} = \lim_{n \rightarrow \infty} \sum_{a'} \mathcal{S}[[p]]_{(a,\emptyset),(a',b)}^n = (SU)_{(a,\emptyset),(\emptyset,b)}^\infty$$

following from Theorem 4.3.1, Definition of $\mathcal{B}[[\text{---}]]$, Proposition 4.4.2, and finally Theorem 4.4.7. □

B.3 Background on Datacenter Topologies

Data center topologies typically organize the network fabric into several levels of switches.

FatTree. A FatTree [3] is perhaps the most common example of a multi-level, multi-rooted tree topology. Figure 4.5 shows a 3-level FatTree topology with 20 switches. The bottom level, *edge*, consists of top-of-rack (ToR) switches; each ToR switch connects all the hosts within a rack (not shown in the figure). These switches act as ingress and egress for intra-data center traffic. The other two levels, *aggregation* and *core*, redundantly connect the switches from the edge layer.

The redundant structure of a FatTree makes it possible to implement fault-tolerant routing schemes that detect and automatically route around failed links. For instance, consider routing from a source to a destination along shortest paths—e.g., the green links in the figure depict one possible path from ($s7$) to ($s1$). On the way from the ToR to the core switch, there are multiple paths that could be used to carry the traffic. Hence, if

one of the links goes down, the switches can route around the failure by simply choosing a different path. Equal-cost multi-path (ECMP) routing is widely used—it automatically chooses among the available paths while avoiding longer paths that might increase latency.

However, after reaching a core switch, there is a *unique* shortest path down to the destination. Hence, ECMP no longer provides any resilience if a switch fails in the aggregation layer (*cf.* the red cross in Figure 4.5). A more sophisticated scheme could take a longer (5-hop) detour going all the way to another edge switch, as shown by the red lines in the figure. Unfortunately, such detours can lead to increased latency and congestion.

AB FatTree. The long detours on the downward paths in FatTrees are dictated by the symmetric wiring of aggregation and core switches. AB FatTrees [106] alleviate this by using two types of subtrees, differing in their wiring to higher levels. Figure 4.10(a) shows how to rewire a FatTree to make it an AB FatTree. The two types of subtrees are as follows:

- i) Type A: switches depicted in blue and wired to core using dashed lines.
- ii) Type B: switches depicted in red and wired to core using solid lines.

Type A subtrees are wired in a way similar to FatTree, but Type B subtrees differ in their connections to core switches. In our diagrams, each aggregation switch in a Type A subtree is wired to adjacent core switches, while each aggregation switch in a Type B subtree is wired to core switches in a staggered manner. (See the original paper by Liu et al. [106] for the general construction.)

This slight change in wiring enables much shorter detours around failures in the downward direction. Consider again routing from source (s_7) to destination (s_1). As before, we have multiple options going upwards when following shortest paths (*e.g.*, the

one depicted in green), as well as a unique downward path. But unlike FatTree, if the aggregation switch on the downward path fails, there is a short detour, as shown in blue. This path exists because the core switch, which needs to re-route traffic, is connected to aggregation switches of both types of subtrees. More generally, aggregation switches of the same type as the failed switch provide a 5-hop detour; but aggregation switches of the opposite type provide an efficient 3-hop detour.

Chapter C

Appendix to Chapter 5

C.1 Omitted Proofs

Theorem 5.2.4. *The language model is sound and complete for the relational model:*

$$\llbracket e \rrbracket = \llbracket f \rrbracket \iff \forall i. \mathcal{R}_i \llbracket e \rrbracket = \mathcal{R}_i \llbracket f \rrbracket$$

Proof. Recall from Remark 5.2.1 that there is a language-preserving map φ from GKAT to KAT expressions. As with GKAT's language model, GKAT's relational model is inherited from KAT; that is, $\mathcal{R}_i \llbracket - \rrbracket = \mathcal{R}_i^{\text{KAT}} \llbracket - \rrbracket \circ \varphi$. Thus, the claim follows by Kozen and Smith [96], who showed the equivalent of Theorem 5.2.4 for KAT:

$$\mathcal{K} \llbracket e \rrbracket = \mathcal{K} \llbracket f \rrbracket \iff \forall i. \mathcal{R}_i^{\text{KAT}} \llbracket e \rrbracket = \mathcal{R}_i^{\text{KAT}} \llbracket f \rrbracket. \quad \square$$

Lemma C.1.1. $\mathcal{P}_i \llbracket e \rrbracket$ is a well-defined subprobability kernel. In particular, $\mathcal{P}_i \llbracket (e +_b 1)^n \cdot \bar{b} \rrbracket(\sigma)(\sigma')$ increases monotonically in n and the limit for $n \rightarrow \infty$ exists.

Proof. We begin by showing the first claim by well-founded induction on \prec , the smallest partial order subsuming the subterm order and satisfying

$$(e +_b 1)^n \cdot \bar{b} \prec e^{(b)}$$

for all e, b, n . The claim is obvious except when $e = f^{(b)}$. In that case, we have by induction hypothesis that $F_n := \mathcal{P}_i \llbracket (f +_b 1)^n \cdot \bar{b} \rrbracket(\sigma)(\sigma')$ is well defined and bounded

above by 1 for all n . To establish that $\lim_{n \rightarrow \infty} F_n$ exist and is also bounded above by 1, it then suffices to show the claim that F_n increases monotonically in n .

If $F_n = 0$ then $F_{n+1} \geq F_n$ holds trivially, so assume $F_n > 0$. This implies that $\sigma' \in \text{sat}^\dagger(\bar{b})$. Thus

$$\begin{aligned}
F_{n+1} &= \mathcal{P}_i \llbracket (f +_b 1)^{n+1} \cdot \bar{b} \rrbracket (\sigma)(\sigma') && \text{(def.)} \\
&= \mathcal{P}_i \llbracket (f +_b 1)^{n+1} \rrbracket (\sigma)(\sigma') && (\sigma' \in \text{sat}^\dagger(\bar{b})) \\
&= \sum_{\sigma''} \mathcal{P}_i \llbracket (f +_b 1)^n \rrbracket (\sigma)(\sigma'') \cdot \mathcal{P}_i \llbracket f +_b 1 \rrbracket (\sigma'')(\sigma') && \text{(def.)} \\
&\geq \mathcal{P}_i \llbracket (f +_b 1)^n \rrbracket (\sigma)(\sigma') \cdot \mathcal{P}_i \llbracket f +_b 1 \rrbracket (\sigma')(\sigma') && \text{(nonnegativity)} \\
&= \mathcal{P}_i \llbracket (f +_b 1)^n \rrbracket (\sigma)(\sigma') && (\sigma' \in \text{sat}^\dagger(\bar{b})) \\
&= F_n && \square
\end{aligned}$$

Theorem 5.2.7. *The language model is sound and complete for the probabilistic model:*

$$\llbracket e \rrbracket = \llbracket f \rrbracket \iff \forall i. \mathcal{P}_i \llbracket e \rrbracket = \mathcal{P}_i \llbracket f \rrbracket$$

Proof. By mutual implication.

\Rightarrow : For soundness, we will define a map $\kappa_i: \text{GS} \rightarrow \text{State} \rightarrow \mathcal{D}(\text{State})$ that interprets guarded strings as sub-Markov kernels, and lift it to languages via pointwise summation:

$$\kappa_i(L) := \sum_{w \in L} \kappa_i(w)$$

To establish the claim, we will then show the following equality:

$$\mathcal{P}_i \llbracket - \rrbracket = \kappa_i \circ \llbracket - \rrbracket \tag{C.1}$$

We define $\kappa_i: \text{GS} \rightarrow \text{State} \rightarrow \mathcal{D}(\text{State})$ inductively as follows:

$$\begin{aligned}
\kappa_i(\alpha)(\sigma) &:= [\sigma \in \text{sat}^\dagger(\alpha)] \cdot \delta_\sigma \\
\kappa_i(\alpha p w)(\sigma)(\sigma') &:= [\sigma \in \text{sat}^\dagger(\alpha)] \cdot \sum_{\sigma''} \text{eval}(p)(\sigma)(\sigma'') \cdot \kappa_i(w)(\sigma'')(\sigma)
\end{aligned}$$

To prove Equation (C.1), it suffices to establish the following equations:

$$\kappa_i(\llbracket p \rrbracket) = \text{eval}(p) \quad (\text{C.2})$$

$$\kappa_i(\llbracket b \rrbracket)(\sigma) = [\sigma \in \text{sat}^\dagger(b)] \cdot \delta_\sigma \quad (\text{C.3})$$

$$\kappa_i(\llbracket e \cdot f \rrbracket)(\sigma)(\sigma') = \sum_{\sigma''} \kappa_i(\llbracket e \rrbracket)(\sigma)(\sigma'') \cdot \kappa_i(\llbracket f \rrbracket)(\sigma'')(\sigma') \quad (\text{C.4})$$

$$\kappa_i(\llbracket e +_b f \rrbracket)(\sigma) = [\sigma \in \text{sat}^\dagger(b)] \cdot \kappa_i(\llbracket e \rrbracket)(\sigma) + [\sigma \in \text{sat}^\dagger(\bar{b})] \cdot \kappa_i(\llbracket f \rrbracket)(\sigma) \quad (\text{C.5})$$

From there, Equation (C.1) follows by a straightforward well-founded induction on \prec , the partial from the proof of Lemma C.1.1.

For Equation (C.2), we have

$$\begin{aligned} \kappa_i(\llbracket p \rrbracket)(\sigma)(\sigma') &= \sum_{\alpha, \beta} \kappa_i(\alpha p \beta)(\sigma)(\sigma') \\ &= \sum_{\alpha, \beta, \sigma''} [\sigma \in \text{sat}^\dagger(\alpha)] \cdot \text{eval}(p)(\sigma)(\sigma'') \cdot [\sigma'' \in \text{sat}^\dagger(\beta)] \cdot \delta_{\sigma'}(\sigma'') \\ &= \sum_{\alpha, \beta} [\sigma \in \text{sat}^\dagger(\alpha)] \cdot \text{eval}(p)(\sigma)(\sigma') \cdot [\sigma' \in \text{sat}^\dagger(\beta)] \\ &= \text{eval}(p)(\sigma)(\sigma'), \end{aligned}$$

where the last line follows because $\{\text{sat}^\dagger(b) \mid b \in \text{BExp}\} \subseteq 2^{\text{State}}$ is a Boolean algebra of sets with atoms $\text{sat}^\dagger(\alpha)$, $\alpha \in \text{At}$, meaning that

$$\text{State} = \bigsqcup_{\alpha \in \text{At}} \text{sat}^\dagger(\alpha) \quad \text{and thus} \quad \sum_{\alpha} [\sigma \in \text{sat}^\dagger(\alpha)] = 1.$$

For Equation (C.3), we have

$$\begin{aligned} \kappa_i(\llbracket b \rrbracket)(\sigma) &= \sum_{\alpha \leq b} \kappa_i(\alpha)(\sigma) = \sum_{\alpha \leq b} [\sigma \in \text{sat}^\dagger(\alpha)] \cdot \delta_\sigma = [\sigma \in \bigsqcup_{\alpha \leq b} \text{sat}^\dagger(\alpha)] \cdot \delta_\sigma \\ &= [\sigma \in \text{sat}^\dagger(b)] \cdot \delta_\sigma. \end{aligned}$$

For Equation (C.4), we need the following auxiliary facts:

$$(A1) \quad \kappa_i(\alpha x)(\sigma)(\sigma') = [\sigma \in \text{sat}^\dagger(\alpha)] \cdot \kappa_i(\alpha x)(\sigma)(\sigma')$$

$$(A2) \quad \kappa_i(x\alpha)(\sigma)(\sigma') = [\sigma' \in \text{sat}^\dagger(\alpha)] \cdot \kappa_i(x\alpha)(\sigma)(\sigma')$$

$$(A3) \quad \kappa_i(x\alpha y)(\sigma)(\sigma') = \sum_{\sigma''} \kappa_i(x\alpha)(\sigma)(\sigma'') \cdot \kappa_i(\alpha y)(\sigma'')(\sigma')$$

$$(A4) \quad \llbracket e \rrbracket \diamond \llbracket f \rrbracket \cong \{(\alpha, x\alpha, \alpha y) \mid \alpha \in \text{At}, x\alpha \in \llbracket e \rrbracket, \alpha y \in \llbracket f \rrbracket\}$$

Fact (A1) is immediate by definition of κ_i , and facts (A2) and (A3) follow by straightforward inductions on $|x|$. We defer the proof of (A4) to Lemma C.1.2. We can then compute:

$$\begin{aligned} & \kappa_i(\llbracket e \cdot f \rrbracket)(\sigma)(\sigma') \\ &= \sum_{w \in \llbracket e \rrbracket \diamond \llbracket f \rrbracket} \kappa_i(w)(\sigma)(\sigma') \\ &= \sum_{\alpha \in \text{At}} \sum_{x\alpha \in \llbracket e \rrbracket} \sum_{\alpha y \in \llbracket f \rrbracket} \kappa_i(x\alpha y)(\sigma)(\sigma') && \text{(by A4)} \\ &= \sum_{\alpha \in \text{At}} \sum_{x\alpha \in \llbracket e \rrbracket} \sum_{\alpha y \in \llbracket f \rrbracket} \sum_{\sigma''} \kappa_i(x\alpha)(\sigma)(\sigma'') \cdot \kappa_i(\alpha y)(\sigma'')(\sigma') && \text{(by A3)} \\ &= \sum_{\sigma''} \sum_{\alpha, \beta \in \text{At}} \sum_{x\alpha \in \llbracket e \rrbracket} \sum_{\alpha y \in \llbracket f \rrbracket} [\alpha = \beta] \cdot \kappa_i(x\alpha)(\sigma)(\sigma'') \cdot \kappa_i(\beta y)(\sigma'')(\sigma') \end{aligned}$$

and, observing that

$$\begin{aligned} & \kappa_i(x\alpha)(\sigma)(\sigma'') \cdot \kappa_i(\beta y)(\sigma'')(\sigma') \neq 0 \\ & \implies \sigma'' \in \text{sat}^\dagger(\alpha) \wedge \sigma'' \in \text{sat}^\dagger(\beta) && \text{(by A1 and A2)} \\ & \implies \sigma'' \in \text{sat}^\dagger(\alpha \cdot \beta) && \text{(Boolean algebra)} \\ & \implies \alpha = \beta && (\alpha, \beta \in \text{At}) \end{aligned}$$

we obtain Equation (C.4):

$$\begin{aligned} \kappa_i(\llbracket e \cdot f \rrbracket)(\sigma)(\sigma') &= \sum_{\sigma''} \sum_{\substack{\alpha, \beta \in \text{At} \\ x\alpha \in \llbracket e \rrbracket \\ \beta y \in \llbracket f \rrbracket}} \kappa_i(x\alpha)(\sigma)(\sigma'') \cdot \kappa_i(\beta y)(\sigma'')(\sigma') \\ &= \sum_{\sigma''} \kappa_i(\llbracket e \rrbracket)(\sigma)(\sigma'') \cdot \kappa_i(\llbracket f \rrbracket)(\sigma'')(\sigma'). \end{aligned}$$

For Equation (C.5), we need the following identity (for all α, x, b, σ):

$$[\alpha \leq b] \cdot \kappa_i(\alpha x)(\sigma) = [\sigma \in \text{sat}^\dagger(b)] \cdot \kappa_i(\alpha x)(\sigma) \quad (\text{C.6})$$

Using A1, it suffices to show the equivalence

$$\alpha \leq b \wedge \sigma \in \text{sat}^\dagger(\alpha) \iff \sigma \in \text{sat}^\dagger(b) \wedge \sigma \in \text{sat}^\dagger(\alpha)$$

The implication from left to right follows directly by monotonicity of sat^\dagger . For the implication from right to left, we have that either $\alpha \leq b$ or $\alpha \leq \bar{b}$. Using again monotonicity of sat^\dagger , the possibility $\alpha \leq \bar{b}$ is seen to cause a contradiction.

With Identity (C.6) at our disposal, Equation (C.5) is easy to establish:

$$\begin{aligned} & \kappa_i(\llbracket e +_b f \rrbracket)(\sigma) \\ = & \sum_{w \in \llbracket e +_b f \rrbracket} \kappa_i(w)(\sigma) \\ = & \sum_{\alpha x \in \llbracket e \rrbracket} [\alpha \leq b] \cdot \kappa_i(\alpha x)(\sigma) + \sum_{\beta y \in \llbracket f \rrbracket} [\alpha \leq \bar{b}] \cdot \kappa_i(\beta y)(\sigma) \\ = & \sum_{\alpha x \in \llbracket e \rrbracket} [\sigma \in \text{sat}^\dagger(b)] \cdot \kappa_i(\alpha x)(\sigma) + \sum_{\beta y \in \llbracket f \rrbracket} [\sigma \in \text{sat}^\dagger(\bar{b})] \cdot \kappa_i(\beta y)(\sigma) \\ = & [\sigma \in \text{sat}^\dagger(b)] \cdot \kappa_i(\llbracket e \rrbracket)(\sigma) + [\sigma \in \text{sat}^\dagger(\bar{b})] \cdot \kappa_i(\llbracket f \rrbracket)(\sigma) \end{aligned}$$

This concludes the soundness proof.

\Leftarrow : For completeness, we will exhibit an interpretation i over the state space GS such that

$$\llbracket e \rrbracket = \{\alpha x \in \text{GS} \mid \mathcal{P}_i \llbracket e \rrbracket(\alpha)(\alpha x) \neq 0\}. \quad (\text{C.7})$$

Define $i := (\text{GS}, \text{eval}, \text{sat})$, where

$$\text{eval}(p)(w) := \text{Unif}(\{wp\alpha \mid \alpha \in \text{At}\}) \quad \text{sat}(t) := \{x\alpha \in \text{GS} \mid \alpha \leq t\}$$

We need two auxiliary observations:

(A1) $\alpha \in \text{sat}^\dagger(b) \iff \alpha \in \llbracket b \rrbracket$

(A2) Monotonicity: $\mathcal{P}_i \llbracket e \rrbracket(v)(w) \neq 0 \implies \exists x. w = vx$.

They follow by straightforward inductions on b and e , respectively. To establish Equation (C.7), it suffices to show the following equivalence for all $x, y \in (\text{At} \cup \Sigma)^*$:

$$\mathcal{P}_i \llbracket e \rrbracket(x\alpha)(x\alpha y) \neq 0 \iff \alpha y \in \llbracket e \rrbracket$$

We proceed by well-founded induction on the ordering \prec on expressions from the proof of Lemma C.1.1:

- For $e = b$, we use fact (A1) to derive that

$$\mathcal{P}_i \llbracket b \rrbracket(x\alpha) = [\alpha \in \text{sat}^\dagger(b)] \cdot \delta_{x\alpha} = [\alpha \in \llbracket b \rrbracket] \cdot \delta_{x\alpha}.$$

Thus we have

$$\mathcal{P}_i \llbracket b \rrbracket(x\alpha)(x\alpha y) \neq 0 \iff y = \varepsilon \wedge \alpha \in \llbracket b \rrbracket \iff \alpha y \in \llbracket b \rrbracket.$$

- For $e = p$, we have that

$$\mathcal{P}_i \llbracket p \rrbracket(x\alpha) = \text{Unif}(\{x\alpha p\beta \mid \beta \in \text{At}\}).$$

It follows that

$$\mathcal{P}_i \llbracket p \rrbracket(x\alpha)(x\alpha y) \neq 0 \iff \exists \beta. y = p\beta \iff \alpha y \in \llbracket p \rrbracket.$$

- For $e +_b f$, we have that

$$\mathcal{P}_i \llbracket e +_b f \rrbracket(x\alpha)(x\alpha y) = \begin{cases} \mathcal{P}_i \llbracket e \rrbracket(x\alpha)(x\alpha y) & \text{if } \alpha \in \text{sat}^\dagger(b) \\ \mathcal{P}_i \llbracket f \rrbracket(x\alpha)(x\alpha y) & \text{if } \alpha \in \text{sat}^\dagger(\bar{b}) \end{cases}$$

We will argue the case $\alpha \in \text{sat}^\dagger(b)$ explicitly; the argument for the case $\alpha \in \text{sat}^\dagger(\bar{b})$ is analogous. We compute:

$$\begin{aligned}
\mathcal{P}_i\llbracket e +_b f \rrbracket(x\alpha)(x\alpha y) \neq 0 &\iff \mathcal{P}_i\llbracket e \rrbracket(x\alpha)(x\alpha y) \neq 0 && \text{(premise)} \\
&\iff \alpha y \in \llbracket e \rrbracket && \text{(ind. hypothesis)} \\
&\iff \alpha y \in \llbracket b \rrbracket \diamond \llbracket e \rrbracket && \text{(A1 and premise)} \\
&\iff \alpha y \in \llbracket e +_b f \rrbracket && \text{(A1 and premise)}
\end{aligned}$$

- For $e \cdot f$, recall that

$$\mathcal{P}_i\llbracket e \cdot f \rrbracket(x\alpha)(x\alpha y) = \sum_w \mathcal{P}_i\llbracket e \rrbracket(x\alpha)(w) \cdot \mathcal{P}_i\llbracket f \rrbracket(w)(x\alpha y).$$

Thus,

$$\begin{aligned}
&\mathcal{P}_i\llbracket e \cdot f \rrbracket(x\alpha)(x\alpha y) \neq 0 \\
&\iff \exists w. \mathcal{P}_i\llbracket e \rrbracket(x\alpha)(w) \neq 0 \wedge \mathcal{P}_i\llbracket f \rrbracket(w)(x\alpha y) \neq 0 && \text{(arg. above)} \\
&\iff \exists z. \mathcal{P}_i\llbracket e \rrbracket(x\alpha)(x\alpha z) \neq 0 \wedge \mathcal{P}_i\llbracket f \rrbracket(x\alpha z)(x\alpha y) \neq 0 && \text{(A2)} \\
&\iff \exists z. \alpha z \in \llbracket e \rrbracket \wedge \mathcal{P}_i\llbracket f \rrbracket(x\alpha z)(x\alpha y) \neq 0 && \text{(ind. hypothesis)} \\
&\iff \exists z, \beta. z\beta \in \llbracket e \rrbracket \wedge \mathcal{P}_i\llbracket f \rrbracket(xz\beta)(x\alpha y) \neq 0 && \text{(A2)} \\
&\iff \exists z, z', \beta. z\beta \in \llbracket e \rrbracket \wedge \mathcal{P}_i\llbracket f \rrbracket(xz\beta)(xz\beta z') \neq 0 \wedge \alpha y = z\beta z' && \text{(A2)} \\
&\iff \exists z, z', \beta. z\beta \in \llbracket e \rrbracket \wedge \beta z' \in \llbracket f \rrbracket \wedge \alpha y = z\beta z' && \text{(ind. hypothesis)} \\
&\iff \alpha y \in \llbracket e \cdot f \rrbracket && \text{(def. } \llbracket - \rrbracket, \diamond)
\end{aligned}$$

- For e^* , recall that

$$\mathcal{P}_i\llbracket e^* \rrbracket(x\alpha)(x\alpha y) = \lim_{n \rightarrow \infty} \mathcal{P}_i\llbracket (e +_b 1)^n \cdot \bar{b} \rrbracket(x\alpha)(x\alpha y)$$

Since this is the limit of a monotone sequence by Lemma C.1.1, it follows that

$$\begin{aligned}
& \mathcal{P}_i[[e^*]](x\alpha)(x\alpha y) \neq 0 \\
& \iff \exists n. \mathcal{P}_i[[e +_b 1]^n \cdot \bar{b}]](x\alpha)(x\alpha y) \neq 0 && \text{(arg. above)} \\
& \iff \exists n. \alpha y \in [[(e +_b 1)^n \cdot \bar{b}]] && \text{(ind. hypothesis)} \\
& \iff \exists m. \alpha y \in [(be)^m \cdot \bar{b}] && \text{(to be argued)} \\
& \iff \alpha y \in [e^{(b)}] && \text{(def. } [[-]] \text{)}
\end{aligned}$$

The penultimate step is justified by the identity

$$[[e +_b 1]^n \cdot \bar{b}] = \bigcup_{m=0}^n [(be)^m \cdot \bar{b}], \quad (\text{C.8})$$

which we establish by induction on $n \geq 0$:

The case $n = 0$ is trivial. For $n > 0$, we have the following KAT equivalence:

$$\begin{aligned}
& (be + \bar{b})^n \cdot \bar{b} \equiv (be + \bar{b}) \cdot (be + \bar{b})^{n-1} \cdot \bar{b} \\
& \equiv (be + \bar{b}) \cdot \sum_{m=0}^{n-1} (be)^m \cdot \bar{b} && \text{(ind. hypothesis)} \\
& \equiv \sum_{m=0}^{n-1} (be) \cdot (be)^m \cdot \bar{b} + \sum_{m=0}^{n-1} \bar{b} \cdot (be)^m \cdot \bar{b} \\
& \equiv \sum_{m=1}^n (be)^m \cdot \bar{b} + \bar{b} \equiv \sum_{m=0}^n (be)^m \cdot \bar{b},
\end{aligned}$$

where the induction hypothesis yields the KAT equivalence

$$(be + \bar{b})^{n-1} \cdot \bar{b} \equiv \sum_{m=0}^{n-1} (be)^m \cdot \bar{b}$$

thanks to completeness of the KAT axioms for the language model. The claim follows by soundness of the KAT axioms.

This concludes the proof of Theorem 5.2.7. □

Lemma C.1.2. *For deterministic languages $L, K \in \mathcal{L}$, we have the following isomorphism:*

$$L \diamond K \cong \{(\alpha, x\alpha, \alpha y) \mid \alpha \in \text{At}, x\alpha \in L, \alpha y \in K\}$$

Proof. We clearly have a surjective map

$$(\alpha, x\alpha, y\alpha) \mapsto x\alpha y$$

from right to left. To see that this map is also injective, we show that for all $x_1\alpha, x_2\beta \in L$ and $\alpha y_1, \beta y_2 \in K$ satisfying $x_1\alpha y_1 = x_2\beta y_2$, we must have $(\alpha, x_1\alpha, \alpha y_2) = (\beta, x_2\beta, \beta y_2)$. This is obvious when $|x_1| = |x_2|$, so assume $|x_1| \neq |x_2|$. We will show that this is impossible.

W.l.o.g. we have $|x_1| < |x_2|$. By the assumed equality, it follows that x_2 must be of the form $x_2 = x_1\alpha z$ for some z , and further $zy_1 = \beta y_2$. Now consider the language

$$L_{x_1} := \{w \in \text{GS} \mid x_1 w \in L\}.$$

The language is deterministic, and it contains both α and $\alpha z\beta$; but the latter contradicts the former. \square

Theorem 5.3.3 (Soundness). *The GKAT axioms are sound for the language model:*

$$e \equiv f \quad \Longrightarrow \quad \llbracket e \rrbracket = \llbracket f \rrbracket.$$

Proof. Formally, the proof proceeds by induction on the construction of \equiv as a congruence. Practically, it suffices to verify soundness of each rule—the inductive cases of the congruence are straightforward because of how $\llbracket - \rrbracket$ is defined.

(U1) For $e +_b e \equiv e$, we derive

$$\begin{aligned} \llbracket e +_b e \rrbracket &= \llbracket e \rrbracket +_{\llbracket b \rrbracket} \llbracket e \rrbracket && \text{(Def. } \llbracket - \rrbracket \text{)} \\ &= \llbracket b \rrbracket \diamond \llbracket e \rrbracket \cup \overline{\llbracket b \rrbracket} \diamond \llbracket e \rrbracket && \text{(Def. } +_B \text{)} \\ &= (\llbracket b \rrbracket \cup \overline{\llbracket b \rrbracket}) \diamond \llbracket e \rrbracket && \text{(Def. } \diamond \text{)} \\ &= \text{At} \diamond \llbracket e \rrbracket && \text{(Def. } \overline{B} \text{)} \\ &= \llbracket e \rrbracket && \text{(Def. } \diamond \text{)} \end{aligned}$$

(U2) For $e +_b f \equiv f +_{\bar{b}} e$, we derive

$$\begin{aligned}
\llbracket e +_b f \rrbracket &= \llbracket e \rrbracket +_{\llbracket b \rrbracket} \llbracket f \rrbracket && \text{(Def. } \llbracket - \rrbracket \text{)} \\
&= \llbracket b \rrbracket \diamond \llbracket e \rrbracket \cup \overline{\llbracket b \rrbracket} \diamond \llbracket f \rrbracket && \text{(Def. } +_B \text{)} \\
&= \overline{\llbracket b \rrbracket} \diamond \llbracket f \rrbracket \cup \llbracket b \rrbracket \diamond \llbracket e \rrbracket && \text{(Def. } \cup \text{)} \\
&= \overline{\llbracket b \rrbracket} \diamond \llbracket f \rrbracket \cup \overline{\overline{\llbracket b \rrbracket}} \diamond \llbracket e \rrbracket && \text{(Def. } \llbracket - \rrbracket, \overline{B} \text{)} \\
&= \llbracket f \rrbracket +_{\overline{\llbracket b \rrbracket}} \llbracket e \rrbracket && \text{(Def. } +_B \text{)} \\
&= \llbracket f +_{\bar{b}} e \rrbracket && \text{(Def. } \llbracket - \rrbracket \text{)}
\end{aligned}$$

(U3) For $(e +_b f) +_c g \equiv e +_{bc} (f +_c g)$, we derive

$$\begin{aligned}
\llbracket (e +_b f) +_c g \rrbracket &= \llbracket c \rrbracket \diamond (\llbracket b \rrbracket \diamond \llbracket e \rrbracket \cup \overline{\llbracket b \rrbracket} \diamond \llbracket f \rrbracket) \cup \overline{\llbracket c \rrbracket} \diamond \llbracket g \rrbracket && \text{(Def. } \llbracket - \rrbracket \text{)} \\
&= \llbracket b \rrbracket \diamond \llbracket c \rrbracket \diamond \llbracket e \rrbracket \cup \overline{\llbracket b \rrbracket} \diamond \llbracket c \rrbracket \diamond \llbracket f \rrbracket \cup \overline{\llbracket c \rrbracket} \diamond \llbracket g \rrbracket && \text{(Def. } \diamond \text{)} \\
&= \llbracket bc \rrbracket \diamond \llbracket e \rrbracket \cup \overline{\llbracket bc \rrbracket} \diamond (\llbracket c \rrbracket \diamond \llbracket f \rrbracket \cup \overline{\llbracket c \rrbracket} \diamond \llbracket g \rrbracket) && \text{(Def. } \llbracket - \rrbracket, \diamond \text{)} \\
&= \llbracket e \rrbracket +_{\llbracket bc \rrbracket} (\llbracket f \rrbracket +_{\llbracket c \rrbracket} \llbracket g \rrbracket) && \text{(Def. } +_B \text{)} \\
&= \llbracket e +_{bc} (f +_c g) \rrbracket && \text{(Def. } \llbracket - \rrbracket \text{)}
\end{aligned}$$

(U4) For $e +_b f \equiv be +_b f$, we derive

$$\begin{aligned}
\llbracket e +_b f \rrbracket &= \llbracket e \rrbracket +_{\llbracket b \rrbracket} \llbracket f \rrbracket && \text{(Def. } \llbracket - \rrbracket \text{)} \\
&= \llbracket b \rrbracket \diamond \llbracket e \rrbracket \cup \overline{\llbracket b \rrbracket} \diamond \llbracket f \rrbracket && \text{(Def. } +_B \text{)} \\
&= \llbracket b \rrbracket \diamond (\llbracket b \rrbracket \diamond \llbracket e \rrbracket) \cup \overline{\llbracket b \rrbracket} \diamond \llbracket f \rrbracket && \text{(Def. } \diamond \text{)} \\
&= (\llbracket b \rrbracket \diamond \llbracket e \rrbracket) +_{\llbracket b \rrbracket} \llbracket f \rrbracket && \text{(Def. } +_B \text{)} \\
&= \llbracket be +_b f \rrbracket && \text{(Def. } \llbracket - \rrbracket \text{)}
\end{aligned}$$

(U5) For $(e +_b f) \cdot g \equiv eg +_b fg$, we derive

$$\begin{aligned}
\llbracket (e +_b f) \cdot g \rrbracket &= (\llbracket e \rrbracket +_{\llbracket b \rrbracket} \llbracket f \rrbracket) \diamond \llbracket g \rrbracket && \text{(Def. } \llbracket - \rrbracket \text{)} \\
&= (\llbracket b \rrbracket \diamond \llbracket e \rrbracket \cup \overline{\llbracket b \rrbracket} \diamond \llbracket f \rrbracket) \diamond \llbracket g \rrbracket && \text{(Def. } +_B \text{)} \\
&= (\llbracket b \rrbracket \diamond \llbracket e \rrbracket) \diamond \llbracket g \rrbracket \cup (\overline{\llbracket b \rrbracket} \diamond \llbracket f \rrbracket) \diamond \llbracket g \rrbracket && \text{(Def. } \diamond \text{)} \\
&= \llbracket b \rrbracket \diamond (\llbracket e \rrbracket \diamond \llbracket g \rrbracket) \cup \overline{\llbracket b \rrbracket} \diamond (\llbracket f \rrbracket \diamond \llbracket g \rrbracket) && \text{(Def. } \diamond \text{)} \\
&= (\llbracket e \rrbracket \diamond \llbracket g \rrbracket) +_{\llbracket b \rrbracket} (\llbracket f \rrbracket \diamond \llbracket g \rrbracket) && \text{(Def. } \diamond \text{)} \\
&= \llbracket eg +_b fg \rrbracket && \text{(Def. } \llbracket - \rrbracket \text{)}
\end{aligned}$$

(S1) For $(e \cdot f) \cdot g \equiv e \cdot (f \cdot g)$, we derive

$$\begin{aligned}
\llbracket (e \cdot f) \cdot g \rrbracket &= (\llbracket e \rrbracket \diamond \llbracket f \rrbracket) \diamond \llbracket g \rrbracket && \text{(Def. } \llbracket - \rrbracket \text{)} \\
&= \llbracket e \rrbracket \diamond (\llbracket f \rrbracket \diamond \llbracket g \rrbracket) && \text{(Def. } \diamond \text{)} \\
&= \llbracket e \cdot (f \cdot g) \rrbracket && \text{(Def. } \llbracket - \rrbracket \text{)}
\end{aligned}$$

(S2) For $0 \cdot e \equiv 0$, we derive

$$\begin{aligned}
\llbracket 0 \cdot e \rrbracket &= \llbracket 0 \rrbracket \diamond \llbracket e \rrbracket && \text{(Def. } \llbracket - \rrbracket \text{)} \\
&= \emptyset \diamond \llbracket e \rrbracket && \text{(Def. } \llbracket - \rrbracket \text{)} \\
&= \emptyset && \text{(Def. } \diamond \text{)} \\
&= \llbracket 0 \rrbracket && \text{(Def. } \llbracket - \rrbracket \text{)}
\end{aligned}$$

(S3) The proof for $e \cdot 0 \equiv 0$ is similar to the above.

(S4) For $1 \cdot e \equiv e$, we derive

$$\begin{aligned}
\llbracket 1 \cdot e \rrbracket &= \llbracket 1 \rrbracket \diamond \llbracket e \rrbracket && \text{(Def. } \llbracket - \rrbracket \text{)} \\
&= \text{At} \diamond \llbracket e \rrbracket && \text{(Def. } \llbracket - \rrbracket \text{)} \\
&= \llbracket e \rrbracket && \text{(Def. } \diamond \text{)}
\end{aligned}$$

(S5) The proof for $e \cdot 1 \equiv e$ is similar to the above.

(W1) For $e^{(b)} \equiv ee^{(b)} +_b 1$, we derive

$$\begin{aligned}
\llbracket e^{(b)} \rrbracket &= \llbracket e \rrbracket^{(b)} && \text{(Def. } \llbracket - \rrbracket \text{)} \\
&= \bigcup_{n \geq 0} (\llbracket b \rrbracket \diamond \llbracket e \rrbracket)^n \diamond \overline{\llbracket b \rrbracket} && \text{(Def. } L^{(B)} \text{)} \\
&= \overline{\llbracket b \rrbracket} \diamond \llbracket 1 \rrbracket \cup \llbracket b \rrbracket \diamond \llbracket e \rrbracket \diamond \bigcup_{n \geq 0} (\llbracket b \rrbracket \diamond \llbracket e \rrbracket)^n \diamond \overline{\llbracket b \rrbracket} && \text{(Def. } \diamond, L^n, \cup \text{)} \\
&= \overline{\llbracket b \rrbracket} \diamond \llbracket 1 \rrbracket \cup \llbracket b \rrbracket \diamond \llbracket e \rrbracket \diamond \llbracket e^{(b)} \rrbracket && \text{(Def. } L^{(B)} \text{)} \\
&= \llbracket e \cdot e^{(b)} +_b 1 \rrbracket && \text{(Def. } \llbracket - \rrbracket, +_B \text{)}
\end{aligned}$$

(W2) For $(ce)^{(b)} \equiv (e +_c 1)^{(b)}$, we first argue that if $w \in ([[b]] \diamond [c] \diamond [e] \cup \overline{[c]})^n$ for some n , then $w \in ([[b]] \diamond [c] \diamond [e])^m$ for some $m \leq n$, by induction on n . In the base, where $n = 0$, we have $w \in \text{At}$; hence, the claim holds immediately. For the inductive step, let $n > 0$ and write

$$w = w_0 \diamond w' \quad w_0 \in [[b]] \diamond [c] \diamond [e] \cup \overline{[c]} \quad w' \in ([[b]] \diamond [c] \diamond [e] \cup \overline{[c]})^{n-1}$$

By induction, we know that $w' \in ([[b]] \diamond [c] \diamond [e])^{m'}$ for $m' \leq n - 1$. If $w_0 \in \overline{[c]}$, then $w = w'$, and the claim goes through if we choose $m = m'$. Otherwise, if $w_0 \in [[b]] \diamond [c] \diamond [e]$, then

$$w = w_0 \diamond w' \in [[b]] \diamond [c] \diamond [e] \diamond ([[b]] \diamond [c] \diamond [e])^{m'} = ([[b]] \diamond [c] \diamond [e])^{m'+1}$$

and thus the claim holds if we choose $m = m' + 1$. Using this, we derive

$$\begin{aligned} [[(ce)^{(b)}]] &= [[ce]]^{(b)} && \text{(Def. } [-]) \\ &= \bigcup_{n \geq 0} ([[b]] \diamond [c] \diamond [e])^n \diamond \overline{[b]} && \text{(Def. } L^{(B)}) \\ &= \bigcup_{n \geq 0} ([[b]] \diamond [c] \diamond [e] \cup \overline{[c]})^n \diamond \overline{[b]} && \text{(above derivation)} \\ &= ([[c] \diamond [e] \cup \overline{[c]} \diamond [1]])^{(b)} && \text{(Def. } L^{(B)}, \diamond, [-]) \\ &= ([e] +_{[c]} [1])^{(b)} && \text{(Def. } +_B) \\ &= [(e +_c 1)^{(b)}] && \text{(Def. } [-]) \end{aligned}$$

This completes the proof. \square

Theorem 5.3.7 (Fundamental Theorem). *For all GKAT programs e , the following equality holds:*

$$e \equiv 1 +_{E(e)} D(e), \quad \text{where } D(e) := \bigoplus_{\alpha: D_\alpha(e)=(p_\alpha, e_\alpha)} p_\alpha \cdot e_\alpha. \quad (5.1)$$

Proof. By induction on e . For a primitive action p , $D_\alpha(p) = (p, 1)$, for all $\alpha \in \text{At}$, and

$E(p) = 0$. Then

$$p \stackrel{\text{U7}}{\equiv} 1 +_0 p \stackrel{\text{Lem.C.2.3}}{\equiv} 1 +_0 \bigoplus_{\alpha \leq 1} \alpha \cdot p \cdot 1 = 1 +_{E(p)} \bigoplus_{\alpha: D_\alpha(e)=(p_\alpha, e_\alpha)} p_\alpha \cdot e_\alpha.$$

For a primitive test c , $D_\alpha(c) = [\alpha \leq c]$ and $E(c) = c$. Then

$$c \stackrel{\text{U6}}{\equiv} 1 +_c 0 = 1 +_{E(c)} \bigoplus_{\alpha: D_\alpha(e)=(p_\alpha, e_\alpha)} p_\alpha \cdot e_\alpha.$$

For a conditional $e_1 +_c e_2$, we have inductively:

$$e_i \equiv 1 +_{E(e_i)} \bigoplus_{\alpha: D_\alpha(e_i)=(p_\alpha, e_\alpha)} p_\alpha \cdot e_\alpha, \quad i \in \{1, 2\}. \quad (\text{C.9})$$

Then

$$\begin{aligned} e_1 +_c e_2 &\equiv \left(1 +_{E(e_1)} \bigoplus_{\alpha: D_\alpha(e_1)=(p_\alpha, e_\alpha)} p_\alpha \cdot e_\alpha \right) +_c \left(1 +_{E(e_2)} \bigoplus_{\alpha: D_\alpha(e_2)=(p_\alpha, e_\alpha)} p_\alpha \cdot e_\alpha \right) \\ &\hspace{20em} (\text{Eq. (C.9)}) \\ &= 1 +_{E(e_1)+_c E(e_2)} \left(\bigoplus_{\alpha: D_\alpha(e_1)=(p_\alpha, e_\alpha)} p_\alpha \cdot e_\alpha +_c \bigoplus_{\alpha: D_\alpha(e_2)=(p_\alpha, e_\alpha)} p_\alpha \cdot e_\alpha \right) \\ &\hspace{20em} (\text{skew assoc.}) \\ &= 1 +_{E(e_1+_c e_2)} \bigoplus_{\alpha: D_\alpha(e_1+_c e_2)=(p_\alpha, e_\alpha)} p_\alpha \cdot e_\alpha. \hspace{5em} (\text{def } D_\alpha(e_1 +_c e_2)) \end{aligned}$$

For sequential composition $e_1 \cdot e_2$, suppose e_1 and e_2 are decomposed as in (C.9).

$$\begin{aligned}
& e_1 \cdot e_2 \\
& \equiv \left(1 +_{E(e_1)} \bigoplus_{\alpha: D_\alpha(e_1)=(p_\alpha, e_\alpha)} p_\alpha \cdot e_\alpha \right) \cdot e_2 && \text{(Eq. (C.9))} \\
& = e_2 +_{E(e_1)} \bigoplus_{\alpha: D_\alpha(e_1)=(p_\alpha, e_\alpha)} p_\alpha \cdot e_\alpha \cdot e_2 && \text{(right distri. U5)} \\
& = \left(1 +_{E(e_2)} \bigoplus_{\alpha: D_\alpha(e_2)=(p_\alpha, e_\alpha)} p_\alpha \cdot e_\alpha \right) +_{E(e_1)} \bigoplus_{\alpha: D_\alpha(e_1)=(p_\alpha, e_\alpha)} p_\alpha \cdot e_\alpha \cdot e_2 && \text{(Eq. (C.9))} \\
& = 1 +_{E(e_1)E(e_2)} \left(\left(\bigoplus_{\alpha: D_\alpha(e_2)=(p_\alpha, e_\alpha)} p_\alpha \cdot e_\alpha \right) +_{E(e_1)} \left(\bigoplus_{\alpha: D_\alpha(e_1)=(p_\alpha, e_\alpha)} p_\alpha \cdot e_\alpha \cdot e_2 \right) \right) && \text{(skew assoc. U3)} \\
& = 1 +_{E(e_1)E(e_2)} \bigoplus_{\substack{\alpha: D_\alpha(e_1)=(p_\alpha, e_\alpha) \\ \alpha: D_\alpha(e_2)=(p_\alpha, e_\alpha)}} (p_\alpha \cdot e_\alpha +_{E(e_1)} p_\alpha \cdot e_\alpha \cdot e_2) && \text{(skew assoc. } \oplus \text{)} \\
& = 1 +_{E(e_1 e_2)} \bigoplus_{\alpha: D_\alpha(e_1 e_2)=(p_\alpha, e_\alpha)} p_\alpha \cdot e_\alpha && \text{(def } E(e_1 \cdot e_2) \text{ and } D_\alpha(e_1 \cdot e_2))
\end{aligned}$$

Finally, for a while loop $e^{(c)}$ we will use Lemma 5.3.9 (Productive Loop):

$$\begin{aligned}
e^{(c)} & \equiv (D(e))^{(c)} && \text{(Lemma 5.3.9)} \\
& \equiv 1 +_{\bar{c}} D(e) \cdot (D(e))^{(c)} && \text{(W1 and U2)} \\
& \equiv 1 +_{\bar{c}} \left(\bigoplus_{\alpha: D_\alpha(e)=(p_\alpha, x_\alpha)} p_\alpha \cdot x_\alpha \right) e^{(c)} && \text{(Lemma 5.3.9 and def. of } D(e)) \\
& \equiv 1 +_{\bar{c}} \left(\bigoplus_{\alpha: D_\alpha(e)=(p_\alpha, x_\alpha)} p_\alpha \cdot x_\alpha \cdot e^{(c)} \right) && \text{(U5)} \\
& = 1 +_{E(e^{(c)})} \bigoplus_{\alpha: D_\alpha(e^{(c)})=(p_\alpha, e_\alpha)} p_\alpha \cdot e_\alpha. && \text{(Def. } D(e^{(c)}) \text{ and } E(e^{(c)}) = \bar{c})
\end{aligned}$$

□

Lemma 5.3.8. *Let e be an expression. Then its components $E(e)$ and $D(e)$ satisfy the following identities:*

$$E(D(e)) \equiv 0 \qquad \overline{E(e)} \cdot D(e) \equiv D(e) \qquad \overline{E(e)} \cdot e \equiv D(e)$$

Proof.

$$\begin{aligned} E(D(e)) &= E\left(\sum_{\alpha: D_\alpha(e)=(p_\alpha, e_\alpha)} p_\alpha \cdot e_\alpha\right) = \sum_{\alpha: D_\alpha(e)=(p_\alpha, e_\alpha)} E(p_\alpha \cdot e_\alpha) = 0 \\ \overline{E(e)} \cdot D(e) &= \overline{E(e)} \cdot \left(\sum_{\alpha: D_\alpha(e)=(p_\alpha, e_\alpha)} p_\alpha \cdot e_\alpha\right) \stackrel{\text{Lem C.2.2}}{\equiv} \sum_{\substack{\alpha: D_\alpha(e)=(p_\alpha, e_\alpha) \\ \alpha \leq \overline{E(e)}}} p_\alpha \cdot e_\alpha \stackrel{*}{=} D(e) \\ \overline{E(e)} \cdot e &\stackrel{\text{FT}}{\equiv} \overline{E(e)} \cdot (1 +_{E(e)} D(e)) \stackrel{\text{U8}}{\equiv} D(e) \end{aligned}$$

Note that for $*$ we use the observation that for all α such that $D_\alpha(e) = (p_\alpha, e_\alpha)$ it is immediate that $\alpha \not\leq E(e)$ and hence the condition $\alpha \leq \overline{E(e)}$ is redundant. \square

Lemma 5.3.10. *The facts in Figure 5.2 are derivable from the axioms.*

Proof. We start by deriving the remaining facts for guarded union.

(U3') For $e +_b (f +_c g) \equiv (e +_b f) +_{b+c} g$, we derive

$$e +_b (f +_c g) \equiv (g +_{\bar{c}} f) +_{\bar{b}} e \tag{U2}$$

$$\equiv g +_{\bar{b}\bar{c}} (f +_{\bar{b}} e) \tag{U3}$$

$$\equiv g +_{\bar{b+c}} (f +_{\bar{b}} e) \tag{Boolean algebra}$$

$$\equiv (e +_b f) +_{b+c} g \tag{U2}$$

(U4') For $e +_b f \equiv e +_b \bar{b}f$, we derive

$$e +_b f \equiv f +_{\bar{b}} e \tag{U2}$$

$$\equiv \bar{b}f +_{\bar{b}} e \tag{U4}$$

$$\equiv e +_b \bar{b}f \tag{U2, Boolean algebra}$$

(U5') For $b \cdot (e +_b cf) \equiv be +_c bf$, we derive

$$\begin{aligned}
 b(e +_c f) &\equiv b \cdot (f +_{\bar{c}} e) && \text{(U2)} \\
 &\equiv ((b + c)(b + \bar{c}))(f +_{\bar{c}} e) && \text{(Boolean algebra)} \\
 &\equiv (b + c)((b + \bar{c})(f +_{\bar{c}} e)) && \text{(S1)} \\
 &\equiv (b + c)((f +_{\bar{c}} e) +_{b+\bar{c}} 0) && \text{(U6)} \\
 &\equiv (b + c)(f +_{\bar{c}} (e +_b 0)) && \text{(U3')} \\
 &\equiv (b + c)((e +_b 0) +_c f) && \text{(U2)} \\
 &\equiv (b + c)(be +_c f) && \text{(U6)} \\
 &\equiv (be +_c f) +_{b+c} 0 && \text{(U6)} \\
 &\equiv be +_c (f +_b 0) && \text{(U3')} \\
 &\equiv be +_c bf && \text{(U6)}
 \end{aligned}$$

(U7) For $e +_0 f \equiv f$, we derive

$$\begin{aligned}
 e +_0 f &\equiv (0 \cdot e) +_0 f && \text{(U4)} \\
 &\equiv 0 +_0 f && \text{(S2)} \\
 &\equiv (0 \cdot f) +_0 f && \text{(S2)} \\
 &\equiv f +_0 f && \text{(U4)} \\
 &\equiv f && \text{(U1)}
 \end{aligned}$$

(U8) For $b \cdot (e +_b f) \equiv be$, we derive

$$\begin{aligned}
 b(e +_b f) &\equiv be +_b bf && \text{(U5')} \\
 &\equiv be +_b \bar{b}bf && \text{(U4')} \\
 &\equiv be +_b 0f && \text{(Boolean algebra)} \\
 &\equiv be +_b 0 && \text{(S2)} \\
 &\equiv be && \text{(U6)}
 \end{aligned}$$

Next, we derive the remaining loop facts.

(W4) For $e^{(b)} \equiv e^{(b)}\bar{b}$, we derive

$$\begin{aligned}
 e^{(b)} &\equiv (D(e))^{(b)} && \text{(Productive loop lemma)} \\
 &\equiv D(e)(D(e))^{(b)} +_b 1 && \text{(W1)} \\
 &\equiv D(e)(D(e))^{(b)} +_b \bar{b} && \text{(U4')} \\
 &\equiv (D(e))^{(b)}\bar{b} && \text{(W3)} \\
 &\equiv e^{(b)}\bar{b} && \text{(Productive loop lemma)}
 \end{aligned}$$

(W4') For $e^{(b)} \equiv (be)^{(b)}$, we derive

$$\begin{aligned}
 e^{(b)} &\equiv (D(e))^{(b)} && \text{(Productive loop lemma)} \\
 &\equiv D(e)(D(e))^{(b)} +_b 1 && \text{(W1)} \\
 &\equiv b \cdot D(e)(D(e))^{(b)} +_b 1 && \text{(U4)} \\
 &\equiv (b \cdot D(e))^{(b)} && \text{(W3)} \\
 &\equiv (D(be))^{(b)} && \text{(Def. } D) \\
 &\equiv (be)^{(b)} && \text{(Productive loop lemma)}
 \end{aligned}$$

(W5) For $e^{(0)} \equiv 1$, we derive

$$\begin{aligned}
 e^{(0)} &\equiv (0 \cdot e)^{(0)} && \text{(W4')} \\
 &\equiv 0^{(0)} && \text{(S2)} \\
 &\equiv 0 \cdot 0^{(0)} +_0 1 && \text{(W1)} \\
 &\equiv 0 +_0 1 && \text{(S2)} \\
 &\equiv 1 && \text{(U7)}
 \end{aligned}$$

(W6) For $e^{(1)} \equiv 0$, we derive

$$e^{(1)} \equiv e^{(1)} \cdot \bar{1} \quad (\text{W4})$$

$$\equiv e^{(1)} \cdot 0 \quad (\text{Boolean algebra})$$

$$\equiv 0 \quad (\text{S3})$$

(W6') For $b^{(c)} \equiv \bar{c}$, we derive

$$b^{(c)} \equiv (D(b))^{(c)} \quad (\text{Productive loop lemma})$$

$$\equiv 0^{(c)} \quad (\text{Def. } D)$$

$$\equiv 0 \cdot 0^{(c)} +_c 1 \quad (\text{W1})$$

$$\equiv 0 +_c 1 \quad (\text{S2})$$

$$\equiv 1 +_{\bar{c}} 0 \quad (\text{U2})$$

$$\equiv \bar{c} \cdot 1 \quad (\text{U6})$$

$$\equiv \bar{c} \quad (\text{Boolean algebra})$$

This completes the proof. \square

Proposition 5.4.5. *If s solves \mathcal{X} and x is a state, then $\llbracket s(x) \rrbracket = \ell^{\mathcal{X}}(x)$.*

Proof. We show that

$$w \in \llbracket s(x) \rrbracket \iff w \in L^{\mathcal{X}}(x)$$

for all states x by induction on the length of $w \in \text{GS}$. We will use that w is of the form

$w = \alpha u$ for some $\alpha \in \text{At}$, $u \in (\text{At} \cdot \Sigma)^*$ and thus

$$w \in \llbracket s(x) \rrbracket \iff w \in \llbracket \alpha \cdot s(x) \rrbracket \quad (\text{def. } \llbracket - \rrbracket)$$

$$\iff w \in \llbracket [\delta^{\mathcal{X}}(x)(\alpha)]_s \rrbracket \quad (\text{def. sol., soundness})$$

For $w = \alpha$, we have

$$\alpha \in \llbracket [\delta^{\mathcal{X}}(x)(\alpha)]_s \rrbracket \iff \delta^{\mathcal{X}}(x)(\alpha) = 1 \quad (\text{def. } \llbracket - \rrbracket \text{ \& } \llbracket - \rrbracket)$$

$$\iff \alpha \in L^{\mathcal{X}}(x) \quad (\text{def. } L^{\mathcal{X}})$$

For $w = \alpha p v$, we have

$$\begin{aligned}
\alpha p v \in \llbracket [\delta^{\mathcal{X}}(x)(\alpha)]_s \rrbracket &\iff \exists y. \delta^{\mathcal{X}}(x)(\alpha) = \langle p, y \rangle \wedge v \in \llbracket s(y) \rrbracket && \text{(def. } \llbracket - \rrbracket \text{ \& } \llbracket - \rrbracket \text{)} \\
&\iff \exists y. \delta^{\mathcal{X}}(x)(\alpha) = \langle p, y \rangle \wedge v \in L^{\mathcal{X}}(y) && \text{(induction)} \\
&\iff \alpha p v \in L^{\mathcal{X}}(x) && \text{(def. } L^{\mathcal{X}} \text{)}
\end{aligned}$$

This concludes the proof. \square

Lemma C.1.3. *Let $\mathcal{X} = \langle X, \delta^{\mathcal{X}} \rangle$ be a G -coalgebra. A function $s: X \rightarrow \text{Exp}$ is a solution to \mathcal{X} if and only if for all $\alpha \in \text{At}$ and $x \in X$ it holds that $\alpha \cdot s(x) \equiv \alpha \cdot [\delta^{\mathcal{X}}(x)(\alpha)]_s$.*

Proof. We shall use some of the observations about \vdash from Appendix C.2.

(\Rightarrow) Let s be a solution to \mathcal{X} ; we then derive for $\alpha \in \text{At}$ and $x \in X$ that

$$\begin{aligned}
\alpha \cdot s(x) &\equiv \alpha \cdot \bigoplus_{\alpha \leq 1} [\delta^{\mathcal{X}}(x)(\alpha)]_s && \text{(} s \text{ solves } \mathcal{X} \text{)} \\
&\equiv \alpha \cdot \bigoplus_{\alpha \leq \alpha} [\delta^{\mathcal{X}}(x)(\alpha)]_s && \text{(Lemma C.2.2)} \\
&\equiv \alpha \cdot [\delta^{\mathcal{X}}(x)(\alpha)]_s && \text{(Def. } \oplus, \text{U8)}
\end{aligned}$$

(\Leftarrow) Suppose that for all $\alpha \in \text{At}$ and $x \in X$ we have $\alpha \cdot s(x) \equiv \alpha \cdot [\delta^{\mathcal{X}}(x)(\alpha)]_s$. We can then derive

$$\begin{aligned}
s(x) &\equiv \bigoplus_{\alpha \leq 1} s(x) && \text{(Lemma C.2.3)} \\
&\equiv \bigoplus_{\alpha \leq 1} \alpha \cdot s(x) && \text{(Lemma C.2.4)} \\
&\equiv \bigoplus_{\alpha \leq 1} \alpha \cdot [\delta^{\mathcal{X}}(x)(\alpha)]_s && \text{(premise)} \\
&\equiv \bigoplus_{\alpha \leq 1} [\delta^{\mathcal{X}}(x)(\alpha)]_s && \text{(Lemma C.2.4)}
\end{aligned}$$

This completes the proof. \square

Theorem 5.4.7 (Existence of Solutions). *Any simple coalgebra admits a solution.*

Proof. Assume \mathcal{X} is simple. We proceed by rule induction on the simplicity derivation.

(S1) Suppose $\delta^{\mathcal{X}}: X \rightarrow 2^{\text{At}}$. Then

$$s^{\mathcal{X}}(x) := \sum \{\alpha \in \text{At} \mid \delta^{\mathcal{X}}(x)(\alpha) = 1\}$$

is a solution to \mathcal{X} .

(S2) Suppose $\mathcal{X} = (\mathcal{Y} + \mathcal{Z})[Y, h]$, where $h \in G(Y + Z)$ and \mathcal{Y} and \mathcal{Z} are simple with solutions $s^{\mathcal{Y}}$ and $s^{\mathcal{Z}}$. We need to exhibit a solution $s^{\mathcal{X}}$ to \mathcal{X} . For $y \in Y$ and $z \in Z$ we define

$$\begin{aligned} s^{\mathcal{X}}(y) &:= s^{\mathcal{Y}}(y) \cdot \ell & s^{\mathcal{X}}(z) &:= s^{\mathcal{Z}}(z) \\ \ell &:= \left(\bigoplus_{\alpha \leq b} [h(\alpha)]_{s^{\mathcal{Y}}} \right)^{(b)} \cdot \bigoplus_{\alpha \leq \bar{b}} [h(\alpha)]_{s^{\mathcal{Z}}} & b &:= \sum \{\alpha \in \text{At} \mid h(\alpha) \in \Sigma \times Y\} \end{aligned}$$

By Lemma C.1.3, it then suffices to prove that for $x \in Y + Z$ and $\alpha \in \text{At}$, we have

$$\alpha \cdot s^{\mathcal{X}}(x) \equiv \alpha \cdot [\delta^{\mathcal{X}}(x)(\alpha)]_{s^{\mathcal{X}}}$$

There are two cases to distinguish.

- If $x \in Z$, then

$$\begin{aligned} \alpha \cdot s^{\mathcal{X}}(x) &= \alpha \cdot s^{\mathcal{Z}}(x) && \text{(def. } s^{\mathcal{X}}) \\ &\equiv \alpha \cdot [\delta^{\mathcal{Z}}(x)(\alpha)]_{s^{\mathcal{Z}}} && (s^{\mathcal{Z}} \text{ solves } \mathcal{Z}) \\ &= \alpha \cdot [\delta^{\mathcal{Z}}(x)(\alpha)]_{s^{\mathcal{X}}} && \text{(def. } s^{\mathcal{X}}) \\ &= \alpha \cdot [\delta^{\mathcal{X}}(x)(\alpha)]_{s^{\mathcal{X}}} && \text{(def. } \mathcal{X}) \end{aligned}$$

- If $x \in Y$, then we find by choice of $s^{\mathcal{X}}$ and $s^{\mathcal{Y}}$ that

$$\alpha \cdot s^{\mathcal{X}}(x) = \alpha \cdot s^{\mathcal{Y}}(x) \cdot \ell = \alpha \cdot [\delta^{\mathcal{Y}}(x)(\alpha)]_{s^{\mathcal{Y}}} \cdot \ell$$

We distinguish three subcases:

– If $\delta^{\mathcal{Y}}(x)(\alpha) \in \{0\} \cup \Sigma \times Y$ then $\delta^{\mathcal{Y}}(x)(\alpha) = \delta^{\mathcal{X}}(x)(\alpha)$ and thus

$$\alpha \cdot [\delta^{\mathcal{Y}}(x)(\alpha)]_{s^{\mathcal{Y}}} \cdot \ell = \alpha \cdot [\delta^{\mathcal{X}}(x)(\alpha)]_{s^{\mathcal{Y}}} \cdot \ell \quad (\text{def. } \mathcal{X})$$

$$\equiv \alpha \cdot [\delta^{\mathcal{X}}(x)(\alpha)]_{s^{\mathcal{X}}} \quad (\text{def. } s^{\mathcal{X}})$$

– If $\delta^{\mathcal{Y}}(x)(\alpha) = 1$ and $h(\alpha) \in \Sigma \times Y$, then $\alpha \leq b$ and we can derive

$$\alpha \cdot [\delta^{\mathcal{Y}}(x)(\alpha)]_{s^{\mathcal{Y}}} \cdot \ell \equiv \alpha \cdot \ell \quad (\text{def. } [-])$$

$$\equiv \alpha \cdot [h(\alpha)]_{s^{\mathcal{Y}}} \cdot \ell \quad (\alpha \leq b)$$

$$= \alpha \cdot [h(\alpha)]_{s^{\mathcal{X}}} \quad (\text{def. } s^{\mathcal{X}})$$

$$= \alpha \cdot [\delta^{\mathcal{X}}(x)(\alpha)]_{s^{\mathcal{X}}} \quad (\text{def. } \mathcal{X})$$

– If $\delta^{\mathcal{Y}}(x)(\alpha) = 1$ and $h(\alpha) \notin \Sigma \times Y$, then $\alpha \leq \bar{b}$ and we can derive

$$\alpha \cdot [\delta^{\mathcal{Y}}(x)(\alpha)]_{s^{\mathcal{Y}}} \cdot \ell \equiv \alpha \cdot \ell \quad (\text{def. } [-])$$

$$\equiv \alpha \cdot [h(\alpha)]_{s^{\mathcal{Z}}} \quad (\alpha \leq \bar{b})$$

$$= \alpha \cdot [h(\alpha)]_{s^{\mathcal{X}}} \quad (\text{def. } s^{\mathcal{X}})$$

$$= \alpha \cdot [\delta^{\mathcal{X}}(x)(\alpha)]_{s^{\mathcal{X}}} \quad (\text{def. } \mathcal{X})$$

This completes the proof. □

Lemma C.1.4. *Let $e \in \text{Exp}$ and $\alpha \in \text{At}$. Now $\iota_e(\alpha) = 1$ if and only if $\alpha \leq E(e)$.*

Proof. We proceed by induction on e . In the base, there are two cases.

- If $e = b \in \text{BExp}$, then $\iota_e(\alpha) = 1$ if and only if $\alpha \leq b = E(b)$.
- If $e = p \in \Sigma$, then $\iota_e(\alpha) = 0$ and $E(e) = 0$.

For the inductive step, there are three cases.

- If $e = f +_b g$, then suppose $\alpha \leq b$. In that case, $\iota_e(\alpha) = 1$ holds if and only if $\iota_f(\alpha) = 1$, which by induction is true precisely when $\alpha \leq E(f)$, which is equivalent to $\alpha \leq E(f +_b g)$. The other case can be treated analogously.

- If $e = f \cdot g$, then $\iota_e(\alpha) = 1$ implies that $\iota_f(\alpha) = 1$ and $\iota_g(\alpha) = 1$, which means that $\alpha \leq E(f)$ and $\alpha \leq E(g)$ by induction, and hence $\alpha \leq E(e)$. The other implication can be derived in a similar fashion.
- If $e = f^{(b)}$, then $\iota_e(\alpha) = 1$ is equivalent to $\alpha \leq \bar{b} = E(e)$. □

Theorem 5.4.8 (Correctness II). *Let $e \in \text{Exp}$. Then \mathcal{X}_e^ι admits a solution s such that $e \equiv s(\iota)$.*

Proof. We proceed by induction on e , showing that we can construct a solution s_e to \mathcal{X}_e . For the main claim, if we then show that $e \equiv \bigoplus_{\alpha \leq 1} [\iota_e(\alpha)]_{s_e}$, it follows that we can extend s_e to a solution s of \mathcal{X}_e^ι , by setting $s(\iota) = e$ and $s(x) = s_e(x)$ for $x \in X_e$. In the base, there are two cases.

- If $e = b \in \text{BExp}$, then we choose for s_e the (empty) map from X_e to Exp ; this (vacuously) makes s_e a solution to \mathcal{X}_e . For the second part, we can derive using Lemmas C.2.3 and C.2.4:

$$b \equiv \bigoplus_{\alpha \leq 1} b \equiv \bigoplus_{\alpha \leq 1} \alpha b \equiv \bigoplus_{\alpha \leq 1} \alpha \cdot [\alpha \leq b] \equiv \bigoplus_{\alpha \leq 1} [\alpha \leq b] \equiv \bigoplus_{\alpha \leq 1} [\iota_b(\alpha)]_{s_e}$$

- If $e = p \in \Sigma$, then we choose $s_e(*) = 1$. To see that s_e is a solution to \mathcal{X}_e , note by Lemma C.2.3:

$$s_e(*) = 1 \equiv \bigoplus_{\alpha \leq 1} 1 \equiv \bigoplus_{\alpha \leq 1} [\delta_p(*)](\alpha)]_{s_e}$$

For the second part, derive as follows, using the same Lemma:

$$e = p \equiv \bigoplus_{\alpha \leq 1} p \equiv \bigoplus_{\alpha \leq 1} p \cdot s_e(*) \equiv \bigoplus_{\alpha \leq 1} [\iota_p(\alpha)]_{s_e}$$

For the inductive step, there are three cases.

- If $e = f +_b g$, then by induction we have solutions s_f and s_g to \mathcal{X}_f and \mathcal{X}_g respectively. We now choose s_e as follows:

$$s_e(x) = \begin{cases} s_f(x) & x \in X_f \\ s_g(x) & x \in X_g \end{cases}$$

To see that s_e is a solution, we use Lemma C.1.3. Suppose $x \in \mathcal{X}_f$; we derive for $\alpha \in \text{At}$ that

$$\begin{aligned}
\alpha \cdot \lfloor \delta_e(x)(\alpha) \rfloor_{s_e} &\equiv \alpha \cdot \lfloor \delta_f(x)(\alpha) \rfloor_{s_e} && \text{(def. } \delta_e) \\
&\equiv \alpha \cdot \lfloor \delta_f(x)(\alpha) \rfloor_{s_f} && \text{(def. } s_e) \\
&\equiv \alpha \cdot s_f(x) && \text{(induction)} \\
&\equiv \alpha \cdot s_e(x) && \text{(def. } s_e)
\end{aligned}$$

The case where $x \in \mathcal{X}_g$ is similar. For the second part of the claim, we derive

$$\begin{aligned}
e &= f +_b g \\
&\equiv \left(\bigoplus_{\alpha \leq 1} \lfloor \iota_f(\alpha) \rfloor_{s_f} \right) +_b \left(\bigoplus_{\alpha \leq 1} \lfloor \iota_g(\alpha) \rfloor_{s_g} \right) && \text{(induction)} \\
&\equiv \left(b \cdot \bigoplus_{\alpha \leq 1} \lfloor \iota_f(\alpha) \rfloor_{s_f} \right) +_b \left(\bar{b} \cdot \bigoplus_{\alpha \leq 1} \lfloor \iota_g(\alpha) \rfloor_{s_g} \right) && \text{(U4, U4')} \\
&\equiv \left(\bigoplus_{\alpha \leq b} \lfloor \iota_f(\alpha) \rfloor_{s_f} \right) +_b \left(\bigoplus_{\alpha \leq \bar{b}} \lfloor \iota_g(\alpha) \rfloor_{s_g} \right) && \text{(Lemma C.2.2)} \\
&\equiv \left(\bigoplus_{\alpha \leq b} \lfloor \iota_e(\alpha) \rfloor_{s_e} \right) +_b \left(\bigoplus_{\alpha \leq \bar{b}} \lfloor \iota_e(\alpha) \rfloor_{s_e} \right) && \text{(def. } \iota_e) \\
&\equiv \left(b \cdot \bigoplus_{\alpha \leq 1} \lfloor \iota_e(\alpha) \rfloor_{s_e} \right) +_b \left(\bar{b} \cdot \bigoplus_{\alpha \leq 1} \lfloor \iota_e(\alpha) \rfloor_{s_e} \right) && \text{(Lemma C.2.2)} \\
&\equiv \left(\bigoplus_{\alpha \leq 1} \lfloor \iota_e(\alpha) \rfloor_{s_e} \right) +_b \left(\bigoplus_{\alpha \leq 1} \lfloor \iota_e(\alpha) \rfloor_{s_e} \right) && \text{(U4, U4')} \\
&\equiv \left(\bigoplus_{\alpha \leq 1} \lfloor \iota_e(\alpha) \rfloor_{s_e} \right) && \text{(U1)}
\end{aligned}$$

The case where $\alpha \leq \bar{b}$ follows similarly.

- If $e = f \cdot g$, then by induction we have solutions s_f and s_g to \mathcal{X}_f and \mathcal{X}_g respectively.

We now choose s_e as follows:

$$s_e(x) = \begin{cases} s_f(x) \cdot g & x \in X_f \\ s_g(x) & x \in X_g \end{cases}$$

To see that s_e is a solution to \mathcal{X}_e , we use Lemma C.1.3; there are three cases to consider.

- If $x \in X_f$ and $\delta_f(x)(\alpha) = 1$, then we can derive

$$\begin{aligned}
\alpha \cdot [\delta_e(x)(\alpha)]_{s_e} &\equiv \alpha \cdot [\iota_g(\alpha)]_{s_e} && \text{(def. } \delta_e) \\
&\equiv \alpha \cdot [\iota_g(\alpha)]_{s_g} && \text{(def. } s_e) \\
&\equiv \alpha \cdot g && \text{(induction)} \\
&\equiv \alpha \cdot [\delta_f(x)(\alpha)]_{s_f} \cdot g && \text{(premise)} \\
&\equiv \alpha \cdot s_f(x) \cdot g && \text{(induction)} \\
&\equiv \alpha \cdot s_e(x) && \text{(def. } s_e)
\end{aligned}$$

- If $x \in X_f$ and $\delta_f(x)(\alpha) \neq 1$, then we can derive

$$\begin{aligned}
\alpha \cdot [\delta_e(x)(\alpha)]_{s_e} &\equiv \alpha \cdot [\delta_f(x)(\alpha)]_{s_e} && \text{(def. } \delta_e) \\
&\equiv \alpha \cdot [\delta_f(x)(\alpha)]_{s_f} \cdot g && \text{(premise)} \\
&\equiv \alpha \cdot s_f(x) \cdot g && \text{(induction)} \\
&\equiv \alpha \cdot s_e(x) && \text{(def. } s_e)
\end{aligned}$$

- If $x \in X_g$, then we can derive

$$\begin{aligned}
\alpha \cdot [\delta_e(x)(\alpha)]_{s_e} &\equiv \alpha \cdot [\delta_g(x)(\alpha)]_{s_e} && \text{(def. } \delta_e) \\
&\equiv \alpha \cdot [\delta_g(x)(\alpha)]_{s_g} && \text{(def. } s_e) \\
&\equiv \alpha \cdot s_g(x) && \text{(induction)} \\
&\equiv \alpha \cdot s_e(x) && \text{(def. } s_e)
\end{aligned}$$

For the second claim, suppose $\iota_f(\alpha) = 1$; we then derive

$$\begin{aligned}
\alpha \cdot f \cdot g &\equiv \alpha \cdot [\iota_f(\alpha)]_{s_f} \cdot g && \text{(induction)} \\
&\equiv \alpha \cdot g && \text{(premise)} \\
&\equiv \alpha \cdot [\iota_g(\alpha)]_{s_g} && \text{(induction)} \\
&\equiv \alpha \cdot [\iota_e(\alpha)]_{s_e} && \text{(def. } \iota_e)
\end{aligned}$$

Otherwise, if $\iota_f(\alpha) \neq 1$, then we derive

$$\begin{aligned}
\alpha \cdot f \cdot g &\equiv \alpha \cdot [\iota_f(\alpha)]_{s_f} \cdot g && \text{(induction)} \\
&\equiv \alpha \cdot [\iota_f(\alpha)]_{s_e} && \text{(def. } s_e) \\
&\equiv \alpha \cdot [\iota_e(\alpha)]_{s_e} && \text{(def. } \iota_e)
\end{aligned}$$

From the above and Lemma C.2.3 we can conclude that $e = f \cdot g \equiv \bigoplus_{\alpha \leq 1} [\iota_e(\alpha)]_{s_e}$.

- If $e = f^{(b)}$, then by induction we have a solution s_f to \mathcal{X}_f . We now choose s_e by setting $s_e(x) = s_f(x) \cdot e$. To see that s_e is a solution to \mathcal{X}_e , we use Lemma C.1.3; there are two cases:

- If $\delta_f(x)(\alpha) = 1$, then we can derive

$$\begin{aligned}
\alpha \cdot [\delta_e(x)(\alpha)]_{s_e} &\equiv \alpha \cdot [\iota_e(\alpha)]_{s_e} && \text{(def. } \delta_e) \\
&\equiv \alpha \cdot e && \text{(induction)} \\
&\equiv \alpha \cdot [\delta_f(x)(\alpha)]_{s_f} \cdot e && \text{(premise)} \\
&\equiv \alpha \cdot s_f(x) \cdot e && \text{(induction)} \\
&\equiv \alpha \cdot s_e(x) && \text{(def. } s_e)
\end{aligned}$$

- Otherwise, if $\delta_f(x)(\alpha) \neq 1$, then we can derive

$$\begin{aligned}
\alpha \cdot [\delta_e(x)(\alpha)]_{s_e} &\equiv \alpha \cdot [\delta_f(x)(\alpha)]_{s_e} && \text{(def. } \delta_e) \\
&\equiv \alpha \cdot [\delta_f(x)(\alpha)]_{s_f} \cdot e && \text{(premise)} \\
&\equiv \alpha \cdot s_f(x) \cdot e && \text{(induction)} \\
&\equiv \alpha \cdot s_e(x) && \text{(def. } s_e)
\end{aligned}$$

For the second part of the claim, we consider three cases:

– If $\alpha \leq b$ and $\iota_f(\alpha) = 1$, then derive

$$\begin{aligned}
\alpha \cdot e &\equiv \alpha \cdot (1 +_{E(f)} f)^{(b)} && \text{(Theorem 5.3.7)} \\
&\equiv \alpha \cdot (\overline{E(f)} \cdot f)^{(b)} && \text{(U2, W2)} \\
&\equiv \alpha \cdot (\overline{E(f)} \cdot f \cdot (\overline{E(f)} \cdot f)^{(b)} +_b 1) && \text{(W1)} \\
&\equiv \alpha \cdot \overline{E(f)} \cdot f \cdot e && (\alpha \leq b, \text{U8}) \\
&\equiv 0 && \text{(Lemma C.1.4)} \\
&\equiv \alpha \cdot [\iota_e(\alpha)]_{s_e} && \text{(def. } \iota_e)
\end{aligned}$$

– If $\alpha \leq b$ and $\iota_f(\alpha) \neq 1$, then we derive

$$\begin{aligned}
\alpha \cdot e &\equiv \alpha \cdot (f f^{(b)} +_b 1) && \text{(W1)} \\
&\equiv \alpha \cdot f f^{(b)} && (\alpha \leq b, \text{U8}) \\
&\equiv \alpha \cdot [\iota_f(\alpha)]_{s_f} \cdot e && \text{(induction)} \\
&\equiv \alpha \cdot [\iota_f(\alpha)]_{s_e} && \text{(premise)} \\
&\equiv \alpha \cdot [\iota_e(\alpha)]_{s_e} && \text{(def. } \iota_e)
\end{aligned}$$

– Otherwise, if $\alpha \leq \bar{b}$, then we derive

$$\begin{aligned}
\alpha \cdot e &\equiv \alpha \cdot (f f^{(b)} +_b 1) && \text{(W1)} \\
&\equiv \alpha && (\alpha \leq \bar{b}, \text{U8}) \\
&\equiv \alpha \cdot [\iota_e(\alpha)]_{s_e} && \text{(def. } \iota_e)
\end{aligned}$$

The claim then follows by Lemma C.2.3. □

Theorem 5.6.2. *The uniqueness axiom is sound in the model of guarded strings: given a system of left-affine equations as in (5.4) that is Salomaa, there exists at most one $R : \{x_1, \dots, x_n\} \rightarrow 2^{\text{GS}}$ s.t.*

$$R(x_i) = \left(\bigcup_{1 \leq j \leq n} \llbracket b_{ij} \rrbracket \diamond \llbracket e_{ij} \rrbracket \diamond R(x_j) \right) \cup \llbracket d_i \rrbracket$$

Proof. We recast this system as a matrix-vector equation of the form $x = Mx + D$ in the Kleene algebra with Tests of n -by- n matrices over 2^{GS} ; solutions to x in this equation are in one-to-one correspondence with functions R as above.

We now argue that the solution is unique when the system is Salomaa. We do this by showing that the map $\sigma(x) = Mx + D$ is contractive in a certain metric on $(2^{\text{GS}})^n$, therefore has a unique fixpoint by the Banach fixpoint theorem.

For a finite guarded string $x \in \text{GS}$, let $|x|$ denote the number of action symbols in x . For example, $|\alpha| = 0$ and $|\alpha p \beta| = 1$. For $A, B \subseteq \text{GS}$, define

$$|A| = \begin{cases} \min\{|x| \mid x \in A\} & A \neq \emptyset \\ \infty & A = \emptyset \end{cases} \quad d(A, B) = 2^{-|A \Delta B|}$$

where $2^{-\infty} = 0$ by convention. One can show that $d(-, -)$ is a metric; in fact, it is an ultrametric, as $d(A, C) \leq \max\{d(A, B), d(B, C)\}$, a consequence of the inclusion $A \Delta C \subseteq A \Delta B \cup B \Delta C$. Intuitively, two sets A and B are close if they agree on short guarded strings; in other words, the shortest guarded string in their symmetric difference is long. Moreover, the space is complete, as any Cauchy sequence A_n converges to the limit

$$\bigcup_m \bigcap_{n > m} A_n = \{x \in \text{GS} \mid x \in A_n \text{ for all but finitely many } n\}.$$

For n -tuples of sets A_1, \dots, A_n and B_1, \dots, B_n , define

$$d(A_1, \dots, A_n, B_1, \dots, B_n) = \max_{i=1}^n d(A_i, B_i).$$

This also gives a complete metric space $(2^{\text{GS}})^n$.

For $A, B, C \subseteq \text{GS}$, from Lemma C.1.5(i) and the fact $|A \diamond B| \geq |A| + |B|$, we have

$$|(A \diamond B) \Delta (A \diamond C)| \geq |A \diamond (B \Delta C)| \geq |A| + |B \Delta C|,$$

from which it follows that

$$d(A \diamond B, A \diamond C) \leq 2^{-|A|} d(B, C).$$

In particular, if $D_\alpha(e) \neq 1$ for all α , it is easily shown by induction on e that $|x| \geq 1$ for all $x \in \llbracket e \rrbracket$, thus $|\llbracket e \rrbracket| \geq 1$, and

$$d(\llbracket e \rrbracket \diamond B, \llbracket e \rrbracket \diamond C) \leq 2^{-|\llbracket e \rrbracket|} d(B, C) \leq \frac{1}{2} d(B, C). \quad (\text{C.10})$$

From Lemma C.1.5(ii) and the fact $|A \cup B| = \min |A|, |B|$, we have

$$\begin{aligned} |(bA_1 \cup \bar{b}A_2) \triangle (bB_1 \cup \bar{b}B_2)| &= |(bA_1 \triangle bB_1) \cup (\bar{b}A_2 \triangle \bar{b}B_2)| \\ &= \min |bA_1 \triangle bB_1|, |\bar{b}A_2 \triangle \bar{b}B_2|, \end{aligned}$$

from which it follows that

$$d(bA_1 \cup \bar{b}A_2, bB_1 \cup \bar{b}B_2) = \max d(bA_1, bB_1), d(\bar{b}A_2, \bar{b}B_2).$$

Extrapolating to any guarded sum by induction,

$$d\left(\bigcup_{\alpha} \alpha A_{\alpha}, \bigcup_{\alpha} \alpha B_{\alpha}\right) = \max_{\alpha} d(\alpha A_{\alpha}, \alpha B_{\alpha}). \quad (\text{C.11})$$

Putting everything together,

$$\begin{aligned} d(\sigma(A), \sigma(B)) &= \max_i d\left(\bigcup_j \llbracket e_{ij} \rrbracket \diamond A_j \cup \llbracket d_i \rrbracket, \bigcup_j \llbracket e_{ij} \rrbracket \diamond B_j \cup \llbracket d_i \rrbracket\right) \\ &= \max_i (\max(\max_j d(\llbracket e_{ij} \rrbracket \diamond A_j, \llbracket e_{ij} \rrbracket \diamond B_j), d(\llbracket d_i \rrbracket, \llbracket d_i \rrbracket))) \quad \text{by (C.11)} \\ &= \max_i \max_j d(\llbracket e_{ij} \rrbracket \diamond A_j, \llbracket e_{ij} \rrbracket \diamond B_j) \\ &\leq \frac{1}{2} \max_j d(A_j, B_j) \quad \text{by (C.10)} \\ &= \frac{1}{2} d(A, B). \end{aligned}$$

Thus the map σ is contractive in the metric d with constant of contraction $1/2$. By the Banach fixpoint theorem, σ has a unique solution. \square

Lemma C.1.5. *Let $A \triangle B$ denote the symmetric difference of A and B . We have:*

$$(i) \quad (A \diamond B) \triangle (A \diamond C) \subseteq A \diamond (B \triangle C).$$

$$(ii) (bA_1 \cup \bar{b}A_2) \Delta (bB_1 \cup \bar{b}B_2) = (bA_1 \Delta bB_1) \cup (\bar{b}A_2 \Delta \bar{b}B_2).$$

Proof. (i) Suppose $x \in (A \diamond B) \setminus (A \diamond C)$. Then $x = y \diamond z$ with $y \in A$ and $z \in B$. But $z \notin C$ since $x \notin A \diamond C$, so $z \in B \setminus C$, therefore $x \in A \diamond (B \setminus C)$. Since x was arbitrary, we have shown

$$(A \diamond B) \setminus (A \diamond C) \subseteq A \diamond (B \setminus C).$$

It follows that

$$\begin{aligned} (A \diamond B) \Delta (A \diamond C) &= (A \diamond B) \setminus (A \diamond C) \cup (A \diamond C) \setminus (A \diamond B) \\ &\subseteq A \diamond (B \setminus C) \cup A \diamond (C \setminus B) \\ &= A \diamond ((B \setminus C) \cup (C \setminus B)) \\ &= A \diamond (B \Delta C). \end{aligned}$$

(ii) Using the facts

$$A = bA \cup \bar{b}A \qquad b(A \Delta B) = bA \Delta bB,$$

we have

$$A \Delta B = b(A \Delta B) \cup \bar{b}(A \Delta B) = (bA \Delta bB) \cup (\bar{b}A \Delta \bar{b}B),$$

therefore

$$\begin{aligned} &(bA_1 \cup \bar{b}A_2) \Delta (bB_1 \cup \bar{b}B_2) \\ &= (b(bA_1 \cup \bar{b}A_2) \Delta b(bB_1 \cup \bar{b}B_2)) \cup (\bar{b}(bA_1 \cup \bar{b}A_2) \Delta \bar{b}(bB_1 \cup \bar{b}B_2)) \\ &= (bA_1 \Delta bB_1) \cup (\bar{b}A_2 \Delta \bar{b}B_2). \end{aligned}$$

□

C.2 Generalized Guarded Union

In Section 5.3.2 we needed a more general type of guarded union:

Definition 5.3.5. Let $\Phi \subseteq \text{At}$, and let $\{e_\alpha\}_{\alpha \in \Phi}$ be a set of expressions indexed by Φ . We write

$$\bigoplus_{\alpha \in \Phi} e_\alpha = \begin{cases} e_\beta +_\beta \left(\bigoplus_{\alpha \in \Phi \setminus \{\beta\}} e_\alpha \right) & \beta \in \Phi \\ 0 & \Phi = \emptyset \end{cases}$$

Like other operators on indexed sets, we may abuse notation and replace Φ by a predicate over some atom α , with e_α a function of α ; for instance, we could write $\bigoplus_{\alpha \leq 1} \alpha \equiv 1$. \square

The definition above is ambiguous in that the choice of β is not fixed. However, that does not change the meaning of the expression above, as far as \equiv is concerned.

Lemma C.2.1. *The operator \bigoplus above is well-defined up-to \equiv .*

Proof. We proceed by induction on the number of atoms in Φ . In the base cases, when $\Phi = \emptyset$ or $\Phi = \{\alpha\}$, the claim holds immediately as the whole expression is equal to, respectively, 0 and e_α . For the inductive step, we need to show that for any $\beta, \gamma \in \Phi$:

$$e_\beta +_\beta \left(\bigoplus_{\alpha \in \Phi \setminus \{\beta\}} e_\alpha \right) \equiv e_\gamma +_\gamma \left(\bigoplus_{\alpha \in \Phi \setminus \{\gamma\}} e_\alpha \right)$$

We can derive

$$\begin{aligned}
e_\beta + \beta \left(\bigoplus_{\alpha \in \Phi \setminus \{\beta\}} e_\alpha \right) &\equiv e_\beta + \beta \left(e_\gamma + \gamma \left(\bigoplus_{\alpha \in \Phi \setminus \{\beta, \gamma\}} e_\alpha \right) \right) && \text{(induction)} \\
&\equiv (e_\beta + \beta e_\gamma) + \beta + \gamma \left(\bigoplus_{\alpha \in \Phi \setminus \{\beta, \gamma\}} e_\alpha \right) && \text{(U3')} \\
&\equiv (e_\gamma + \bar{\beta} e_\beta) + \beta + \gamma \left(\bigoplus_{\alpha \in \Phi \setminus \{\beta, \gamma\}} e_\alpha \right) && \text{(U2)} \\
&\equiv e_\gamma + \bar{\beta}(\beta + \gamma) \left(e_\beta + \beta + \gamma \left(\bigoplus_{\alpha \in \Phi \setminus \{\beta, \gamma\}} e_\alpha \right) \right) && \text{(U3)} \\
&\equiv e_\gamma + \gamma \left(e_\beta + \beta \left(\bigoplus_{\alpha \in \Phi \setminus \{\beta, \gamma\}} e_\alpha \right) \right) && \text{(Boolean algebra)} \\
&\equiv e_\gamma + \gamma \left(\bigoplus_{\alpha \in \Phi \setminus \{\gamma\}} e_\alpha \right) && \text{(induction)}
\end{aligned}$$

This completes the proof. \square

The following properties are useful for calculations with \bigoplus .

Lemma C.2.2. *Let $b, c \in \text{BExp}$ and suppose that for every $\alpha \leq b$, we have an $e_\alpha \in \text{Exp}$. The following then holds:*

$$c \cdot \bigoplus_{\alpha \leq b} e_\alpha \equiv \bigoplus_{\alpha \leq bc} e_\alpha$$

Recall from above that the predicate $\alpha \leq b$ is replacing the set $\Phi = \{\alpha \mid \alpha \leq b\}$.

Proof. We proceed by induction on the number of atoms below b . In the base, where $b \equiv 0$, the claim holds vacuously. For the inductive step, assume the claim holds for all b' with strictly fewer atoms. Let $\beta \in \text{At}$ with $b = \beta + b'$ and $\beta \not\leq b'$. There are two cases.

- If $\beta \leq c$, then we derive

$$c \cdot \bigoplus_{\alpha \leq b} e_\alpha \equiv c \cdot \left(e_\beta +_\beta \left(\bigoplus_{\alpha \leq b'} e_\alpha \right) \right) \quad (\text{Def. } \oplus)$$

$$\equiv c \cdot e_\beta +_\beta c \cdot \left(\bigoplus_{\alpha \leq b'} e_\alpha \right) \quad (\text{U5'})$$

$$\equiv c \cdot e_\beta +_\beta \left(\bigoplus_{\alpha \leq b'c} e_\alpha \right) \quad (\text{induction})$$

$$\equiv e_\beta +_\beta \left(\bigoplus_{\alpha \leq b'c} e_\alpha \right) \quad (\text{U4, Boolean algebra})$$

$$\equiv \bigoplus_{\alpha \leq bc} e_\alpha \quad (\text{Def. } \oplus, \text{ Boolean algebra})$$

where in the last step we use $b + c \equiv \beta + b'c$ and $\beta \not\leq b'c$.

- If $\beta \not\leq c$, then we derive

$$c \cdot \bigoplus_{\alpha \leq b} e_\alpha \equiv c \cdot \left(e_\beta +_\beta \left(\bigoplus_{\alpha \leq b'} e_\alpha \right) \right) \quad (\text{Def. } \oplus)$$

$$\equiv c \cdot \left(\bigoplus_{\alpha \leq b'} e_\alpha \right) \quad (\text{U8, Boolean algebra})$$

$$\equiv \bigoplus_{\alpha \leq b'c} e_\alpha \quad (\text{induction})$$

$$\equiv \bigoplus_{\alpha \leq bc} e_\alpha \quad (\text{Boolean algebra})$$

where for the last step we use $bc \equiv (b' + \beta)c = b'c$. □

Lemma C.2.3. For all $e \in \text{Exp}$ and $b \in \text{BExp}$, we have $\bigoplus_{\alpha \leq b} e \equiv be$

Proof. The proof proceeds by induction on the number of atoms below b . In the base, where $b \equiv 0$, the claim holds immediately. Otherwise, assume the claim holds for all $b' \in \text{BExp}$ with strictly fewer atoms than b . Let $\beta \in \text{At}$ be such that $b = \beta \vee b'$ and $\beta \not\leq b'$.

We then calculate:

$$\begin{aligned}
\bigoplus_{\alpha \leq b} e &\equiv e +_{\beta} \left(\bigoplus_{\alpha \leq b'} e \right) && \text{(Def. } \oplus \text{)} \\
&\equiv e +_{\beta} b' e && \text{(induction)} \\
&\equiv \beta e +_{\beta} \bar{\beta} b' e && \text{(U4, U4')} \\
&\equiv \beta b e +_{\beta} \bar{\beta} b e && \text{(Boolean algebra)} \\
&\equiv b e +_{\beta} b e && \text{(U4, U4')} \\
&\equiv b e && \text{(U1)}
\end{aligned}$$

This completes the proof. \square

Lemma C.2.4. *Let $b \in \text{BExp}$ and suppose that for $\alpha \leq b$ we have an $e_{\alpha} \in \text{Exp}$. The following holds:*

$$\bigoplus_{\alpha \leq b} e_{\alpha} \equiv \bigoplus_{\alpha \leq b} \alpha e_{\alpha}$$

Proof. The proof proceeds by induction on the number of atoms below b . In the base, where $b \equiv 0$, the claim holds immediately. Otherwise, assume that the claim holds for all $b' \in \text{BExp}$ with strictly fewer atoms. Let $\beta \in \text{At}$ be such that $b = \beta \vee b'$ and $\beta \not\leq b'$. We then calculate:

$$\begin{aligned}
\bigoplus_{\alpha \leq b} e_{\alpha} &\equiv e +_{\beta} \left(\bigoplus_{\alpha \leq b'} e_{\alpha} \right) && \text{(Def. } \oplus \text{)} \\
&\equiv \beta e_{\beta} +_{\beta} \left(\bigoplus_{\alpha \leq b'} \alpha e_{\alpha} \right) && \text{(U4)} \\
&\equiv \bigoplus_{\alpha \leq b} \alpha e_{\alpha} && \text{(Def. } \oplus \text{)}
\end{aligned}$$

This completes the proof. \square