PRIMITIVES FOR MATCH-ACTION IN THEORY AND PRACTICE

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

Xiang Long

May 2021

PRIMITIVES FOR MATCH-ACTION IN THEORY AND PRACTICE

Xiang Long, Ph.D.

Cornell University 2021

The *match-action* paradigm has remained a popular low-level programming model for specifying packet forwarding behavior in network switches. There are efficient hardware implementations of match-action, and it is a simple model accepted by network programmers. Nevertheless, high-level domain specific languages (DSLs) such as NetKAT and P4 are available to provide abstractions for network policies that can then be compiled down to match-action tables in the target switch. Despite much theoretical work surrounding these DSLs, there has been comparatively little investigation towards putting match-action itself on a firm theoretical foundation.

The aim of this dissertation is to understand existing practice surrounding match-action using theory. In the first part of our work, considering how match tables are implemented currently, we attempt to formally model their computational power. We argue that even the apparently low-level *primitives* of match-action: building blocks such as tables, rules, match patterns, etc., are open to formalization. We take both complexity-theoretic and language-theoretic approaches. On the one hand, we give a circuit complexity model for match-action, and we relate the complexity class AC_0 to match tables of various sizes. Further, we give a *multimatch* extension to traditional match-action, and discover the complexity class that characterizes its expressivity. On the other hand, we describe an algebraic language for match-action called MatchKAT. Although closely related to NetKAT, it shows the same type of language-theoretic techniques can still be applied at match-action's lower abstraction level. In particular, we consider MatchKAT's equational theory, and hence *program equivalence* between match tables expressing the same

forwarding policy.

The second part considers mutable global state. Traditionally, NetKAT regards the packet forwarding plane, and hence match tables, to be stateless. We present Stateful NetKAT, a natural extension to NetKAT by adding state that can be modified during packet processing, and we show that subtle interplay between packet and state data now occurs.

The last part of this dissertation considers the VPP software switch. VPP exposes a set of low-level interfaces on the underlying hardware that compilers of networking DSLs can potentially target. We give performance benchmarks of a network program compiled to VPP from P4, in order to make the case that exposing more low-level interfaces is beneficial as more optimizations can be made. This work is presented in order to give some evidence to our claim that further study into low-level packet forwarding constructs, such as match-action, can lead to more informed compilation from above.

BIOGRAPHICAL SKETCH

Xiang Long was born in China but moved at a young age with his parents to the United Kingdom. He lived in various English cities for short periods of time before settling in Glasgow, Scotland. From 2009 to 2013 Xiang was an undergraduate at Emmanuel College, University of Cambridge. There he completed degrees in Computer Science while rediscovering his English accent. Since 2013, as a PhD student at Cornell University, Xiang has been interested in aspects of Software-Defined Networks, as well as category theory and the theory of programming languages in general. While at Cornell, Xiang also minored in Law, specializing in public law. This realized his dream of studying Law at university since his undergraduate days. During his studies Xiang has interned at and/or collaborated on research with Cisco Systems and Google. He will be working at Google Cloud as a Software Engineer after graduation.

ACKNOWLEDGEMENTS

For me, the PhD journey has been a discovery of new things. A new country, new cities, new university, new friends and colleagues, new areas of Computer Science that I would have never dreamt of researching, a new minor that I have always dreamt of studying, new frustrations, new results, and finally, a new degree and a new job. The first and foremost person I would like to thank is my advisor Dexter Kozen, who has been my guide on this long road. Through his endless patience and belief, Dexter has encouraged me to make progress in my own way, on challenging problems that I find interesting. "Follow your nose", he said, to look for new areas of theory that could be developed to explain old practice. No example of this is better than the multitude of research directions and collaborations Dexter plays a part, and I am privileged to have been a student of just one facet of his interests.

The direction of my research would not be as it is without Nate Foster, who introduced me to Software-Defined Networking. Along with Dexter, Nate showed me how to apply techniques in Programming Languages to Networks, the latter I did not entirely pay attention to as an undergrad. Early on, his guidance on industry helped me connect with Cisco, kicking off a collaboration that gave me invaluable experience. It is thanks to Nate that I am able to mix practice with theory.

I am also deeply thankful to the professors I had the chance to study with at Cornell Law School. Firstly, I am grateful to James Grimmelmann for representing the Law minor on my special committee. Prof Grimmelmann kindly agreed to step in to evaluate my work with the Law School, despite arriving near the end of my minor study. I thank Stephen Yale-Loehr for satisfying my morbid curiosity with Immigration Law, and for making my first foray into Law a pleasant one. I also had the joy of studying multiple topics with Mitchel Lasser. This was a pleasure made greater due to having been able to play, I hope, the British foil to his French

sensibilities in class. *Je vous remercie!* Lastly, the greatest credit must go to Cynthia R. Farina, who not only guided my entire Law curriculum but taught me Administrative Law as well. I was very lucky she was so understanding of the predicament of a Computer Science student who wished to study Law, and I am full of gratitude that she has allowed me to realize this long-held ambition. Prof Farina, may you have a long and happy retirement!

Special thanks go to my collaborators outside of Cornell: Sean Choi, Muhammad Shahbaz, Skip Booth, Andy Keep, John Marshall, and Changhoon Kim. Skip, John, and Andy in particular gave me the opportunity to see the networking industry up close, and without them I would not have had so many North Carolina summers that I remember fondly.

My work would not have been possible without the support from my friends, new or old, personal or otherwise. The PLab was my home in Computer Science. Particular thanks go to Andrew Hirsch, Fabian Mühlböck, Matthew Milano, Jonathan DiLorenzo, Michael Roberts, Pedro Amorim, Dietrich Geisler, Haobin Ni, and Siqiu Yao for making our space so intellectually stimulating during my time there. Long may PLab's heirs continue this spirit, though I fear it will soon lose its advantageous view overlooking Hoy Field. More widely, many thanks to the members of the Programming Languages Discussion Group at Cornell for the numerous illuminating talks over the years. Finally, a very special friend I made at Cornell was Shuhan Wang, who I could always count on to discuss anything in life while keeping my Chinese in check.

Among my personal friends, none are as important to me as Jordan, Amy, and Sammie, who have stayed with me through thick and thin since we were children. Thank you and the dearest love to you all. Thanks also go to Harry and Wez, who have long been my political frenemies, and may the satire keep on flowing. In Ithaca, first I must thank Kyrylo, for all the banter so far that I hope will be endless. Oh and the nagging, I guess. Huge thank yous go to everyone at the

Ithaca Bridge Club, who have been a source of support as well as fun and entertainment before we were cruelly torn apart by the pandemic. Of course, I could never forget Debbie, who has been my bridge partner-in-crime not only in New York but also Pennsylvania, Massachusetts, Washington DC, Georgia, Tennessee, and even Texas. I may have taught you bridge, Debbie, but you taught me America.

The support of the National Science Foundation under grants AitF-1637532, CCF-2008083, and SaTC-1717581, and gifts from Cisco Systems are gratefully acknowledged. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation or Cisco.

This dissertation was typeset in LyX using $X_{\exists}LaTeX$.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

The *match-action* paradigm has been a popular low-level programming model for specifying packet forwarding behavior in network switches. In this model, a switch is organized as one or more *match tables* in a *pipeline*, each table containing *rules* with *patterns* and *actions*. The pattern is some match specification on the binary data fields in a packet header, such as a ternary expression containing 0, 1 or don't-care. The action is some modification on the packet header. When a packet arrives at a match table, a rule is selected among those with matching patterns and the associated action is executed. The selection criterion could be some pre-configured *priority* ordering on the rules, or based on some property of the pattern such as selecting the one with the fewest don't-cares (longest prefix matching) [47]. A few examples of match-action rules are:

- Firewall: if the *destination port* is *22*, drop the packet.
- Access control list: if the *source ip address* is from the subnet *10.0.0.0/24*, forward the packet to subnet *10.0.1.0/24*.
- Network address translation: if the *source port* is in the range *3250-3256*, forward the packet to *192.168.1.3*.

Instead of high-level descriptions as presented here, in reality the rules are stored as low-level patterns that correspond closely with how packet data is stored in hardware. These rules can be layered and prioritized in a pipeline of one or more tables to specify a forwarding policy for a particular switch.

Despite ever more powerful hardware, the match-action paradigm with its limited range of operations nevertheless persists in being the implementation of choice for packet forwarding logic in network devices. This is due to two reasons. First, simplicity translates to efficient implementation in hardware. Parallel matching against multiple rules can be performed very efficiently in hardware using Ternary Content-Addressable Memory (TCAM) [44, 52]. This is essential for the nanosecond timescales required for state of the art network devices to reach the hundreds of Gigabit or even Terabit throughput "line-rates". Actions are typically fixed for each rule that may be matched, and implementing more arbitrary computation logic on the packet header would drive up cost and power consumption. In exchange for simplicity, low expense and speed, computation in the match-action setting is limited to matching against and potentially making fixed modifications on packet headers without complex calculation.

Second, from the point of view of the network programmer, match-action is simple to explain and reason about. As a straightforward programming model, it can be easily held mentally by programmers without needing complex language semantics. The low-level local behavior of a single switch can be explained directly from the match tables it contains, and this is still useful when programming or debugging a switch in isolation.

However, things become more difficult when we start to think about high-level *network policies* that specify how multiple switches must work in concert. The local match-action rules on the collection of switches in the network must be considered as a whole, and working with them quickly becomes cumbersome. Until recently, these rules were hand-configured using low-level rule description languages closely matching the structure of the underlying ruleset implementation. An additional factor was that, with the advent of Software-Defined Networking (SDN) [47],

networks became more complex, and switches in turn became more computationally powerful. Instead of implementing processing logic for common network protocols fixed in hardware, increasingly "blank-slates" are preferred where perhaps even the simplest network functions must be programmed by the user, in exchange for much more flexibility in the sort of protocols and functionality that can be specified [18]. For example, in the past a switch might have been hard-coded to process only IP traffic, whereas a set of SDN switches can be programmed to converse in a custom network protocol that the user has designed.

The implication is that automated configuration processes have become essential in order to maintain the accuracy of network policies, instead of configuring switches by hand. Nowadays there is the option for administrators to define their network policies in a high-level domain-specific language (DSL). Two such languages created for SDN are NetKAT [9] and P4 [16]. They provide abstractions that allow for writing human-readable network policies that can be compiled down ultimately to raw match-action rules and distributed to the switches [6, 22, 58, 60].

1.1 Structure of Our Work

This dissertation focuses chiefly on match-action and NetKAT, with a small amount of P4. Our aim is to understand existing practice surrounding match-action using theory. In the first part of our work, considering how match tables are implemented currently, we attempt to formally model their computational power. Despite much theoretic work surrounding NetKAT [9], P4 [16, 50], and other high-level SDN DSLs such as Frenetic [26], Merlin [63], FatTire [55], and Nettle [66], there has been comparatively little investigation towards putting match-action itself on a firm

theoretical foundation. We argue that even the apparently low-level *primitives* of match-action: building blocks such as tables, rules, match patterns, etc., are open to formalization. This effort gives rise to interesting theoretical results such as limits on what functions can be computed given the size of match tables and an algebraic equational theory for match-action. Our hope is that further research in this direction would benefit those working in the higher abstraction levels. For example, one possible outcome is more informed and optimized compilation of DSLs into the match-action programming model.

This first part is organized as follows:

- Chapter 2 gives a circuit complexity model for match-action, in order to explore the expressive power of match tables. We relate the complexity class AC_0 to match tables of various sizes. Further, we give a *multimatch* extension to traditional match-action, and discover the complexity class that characterizes its expressivity.
- Chapter 3 describes an algebraic language for match-action called MatchKAT. Although closely related to NetKAT [9], this shows the same type of language-theoretic techniques can still be applied at match-action's lower abstraction level. In particular, we consider MatchKAT's equational theory, and hence *program equivalence* between match tables expressing the same forwarding policy.

Together, these chapters demonstrate that we can formalize match-action from both combinatorial and algebraic viewpoints.

Now that we have explored what *is* possible with match-action, the second part of this dissertation highlights something that is not: mutable global state. Chapter 4 presents the Stateful

NetKAT language. Traditionally, NetKAT regards the packet forwarding plane, and hence match tables, to be stateless. Stateful NetKAT is a natural extension to NetKAT obtained by adding state that can be modified during packet processing. We show that there is a subtle interplay between packet data and state data. Additionally, we give a language model for Stateful NetKAT, which gives rise to a decision procedure for a significant fragment. We also start on the effort of giving a coalgebraic decision procedure and axiomatization for the language.

Chapter 5 forms the last part. Instead of match-action, here we consider Cisco's VPP software switch architecture [67, 12]. VPP exposes a set of low-level interfaces on the underlying hardware that compilers of SDN DSLs can potentially target. We give performance benchmarks of a network program compiled to VPP from P4 in order to make the case that exposing more low-level interfaces is beneficial as more optimizations can be made. This chapter is offered here to support our claim that further study into low-level packet forwarding constructs, such as match-action, can lead to more informed compilation from above. This work is based on publications [22, 21] with Sean Choi, Muhammad Shahbaz, Skip Booth, Andy Keep, John Marshall, and Changhoon Kim.

CHAPTER 2

CIRCUIT COMPLEXITY MODEL

The question of the expressiveness of fixed circuits implementing match-action arises naturally, as that corresponds to the reality in hardware. In particular, can we characterize the computational expressiveness of match-action in terms of familiar computation models involving circuits? That is the starting point of this investigation.

As far as we are aware, this is a question that has not been previously explored formally. As a first step in this direction, this chapter aims to capture and describe, in complexity-theoretic terms, the computational power of pipelines of match-action tables in switches. We are not aware of any attempt to do this before now and this work would be the first in rigorously understanding the power and limitation of the match-action paradigm.

Although there has been much practical and implementation work done in the area of packet classification and SDN, theoretical work in this field has been largely about the semantics of network specification languages such as [9, 37]. There is previous work in the algorithmic exploration of match-action, such as developing new algorithms to solve particular network problems [49], optimizing the use of match rules in expressing particular kinds of (range-based) data [48, 56], leveraging and improving the underlying TCAM architecture to find better matching techniques [44, 57], and verifying the correctness of tables [20, 38, 70, 45]. All such work has practical problems related to match-action or TCAMs in mind and introduces ideas towards solving those problems. On the other hand, we are interested in the bounds of expressiveness and computational power of match-action, which we propose to study through defining an abstract

model and relating it to known circuit complexity and Boolean function theory. We are hopeful this theoretical perspective can enhance understanding of the match-action paradigm, and eventually contribute to the improvement of implementations in practice.

The next section gives some further background on match-action and network hardware. We also review the standard circuit complexity class AC_0, which will be important in characterizing the match-action computational model. The contributions of the chapter are summarized as follows:

- We give an abstract model of match-action tables and pipelines (Section 2.2.1). The model is broad enough to capture the behavior of match-action tables currently found in standard industrial products, which we call *single-match tables*, and also various relaxations that are possible improvements on existing practice.
- Using our model, we show that a single-match pipeline with constant depth is able to compute any arbitrary Boolean function (Section 2.2.3) provided an exponential number of rules is allowed with respect to the size of the input. With polynomial rules, expressiveness falls within AC_0 (Section 2.2.4).
- As an example of optimization results that can be obtained, we demonstrate how table and pipeline size can be traded for hardware complexity to produce equivalent pipelines (Theorem 19).
- Finally, we propose an extension of existing practice called *multimatch tables*, by relaxing the restrictions in our model compared to single-match (Section 2.2.6). Although multimatch is not yet implemented in any available commodity hardware, we introduce a new circuit complexity class XC and characterize the computational power of multimatch

pipelines in terms of AC_0 (Theorem 23). This demonstrates our model's ability to reason about yet unimplemented enhancements of the existing match-action paradigm. We argue that multimatch is a natural extension if more unrestricted logic is desired within match-action tables.

2.1 Background

2.1.1 Match-action

We now give a more precise description of match-action than what has been introduced so far. We will use the term *switch* to refer to all packet-forwarding devices that use match-action. The exact type of device (switches, routers, etc.) is not relevant. Logically, we can divide the functionality of a collection of SDN switches into the *dataplane* and the *control-plane*. The former, which will be our main focus, contains the pure packet classification logic. It is implemented with simple, high-performance hardware so that forwarding decisions can be made at the rate necessary to maintain line-speed on the network.

Organization of the dataplane routing logic is laid out as a sequence of *match-action tables*. The tables contain rules that can be written in ternary form: 0, 1, or $*$ for don't-care. These rules are matched against bits in the packet header. For any given packet, there could be multiple rules that match, and different methods can be used to select a single rule from the matching set that will determine the fate of the packet. For example, *longest-prefix* matching selects the rule with the longest prefix not containing a don't-care. *Priority* matching assigns a priority to

each rule, and the selected rule from the matching set is the one with the highest priority. The *action* associated with the selected rule is then executed on the packet, which often includes rewriting fields in the header, before the packet moves to the next processing stage in the switch. In addition, the dataplane may keep some per-packet internal state called the *metadata* that is alive for the lifetime of the packet as it is processed within the switch. For reasons given in Section 2.3.1, in our theoretical development we concentrate mainly on the matching mechanics of tables as opposed to actions.

A sequence of match-action tables can be assembled as a *pipeline*. A switch with a fixed pipeline connecting a constant number of tables may have each table correspond to a hardware stage in a real hardware pipeline. Alternatively, such as in [18], pipelines may be described by some domain-specific language (e.g. NetKAT [9], P4 [16]) and then compiled to an allocation of hardware resources. Such pipelines lay out the general steps in the packet forwarding process, for example specifying that headers corresponding to a lower layer in the network stack must be examined first before examining a higher layer.

The actual packet forwarding decision-making, the description of what action to execute under what matching of bits, is described by the *match-action rules*. The rules can be populated efficiently in the tables at runtime by a network administrator, or generated automatically by a routing protocol. In any case, the memory for the state of the packet header and metadata is typically implemented as a bus of wires running through each match table so that state can be passed table-to-table without complex hardware. In our computation model, it is not necessary to consider the distinction between types of data, whether they are packet headers, payload or metadata. At all times, we will treat the data being processed as binary strings.

As mentioned previously, match-action favors a simple set of fixed operations with straightforward logic. The basic operation is matching on binary strings. Typically actions will be fixed at configuration time as simple modifications to packet data, such as assigning a field with a constant value. This is a much more limited setting than placing arbitrary and-, or-, and not-gates when constructing circuits. The packet processing logic is only fully specified when the switch is configured. Complex programmable logic would have higher cost and less efficiency compared to fixed match-action hardware.

It suffices to mention that the control-plane, in contrast, has an overarching view of the network topology and contains complex logic to configure the dataplane and handle tasks such as global routing, load balancing, and fault recovery. It is typically orders of magnitude slower than the dataplane. We will not be considering the control-plane in detail.

2.1.2 Circuit Complexity and AC_0

The use of circuits to classify complexity of Boolean functions has been a rich field of study for many years. In this chapter, we observe that match tables used in switches are essentially computing Boolean functions $2^n \to 2^n$, and relating a theoretical model of tables and pipelines to circuit complexity theory would make the computational power of those structures more well-defined. Classical circuit complexity theory has produced bounds on the Boolean functions computable through a rigorous reasoning of the properties of the complexity class, such as the non-computability of parity within AC_0 [29]. In the context of match-action, we should also keep optimization in mind, such as exploring whether functions can be computed with fewer rules

or tables. This is important in practice as using fewer resources will give rise to more efficient implementations.

The most relevant complexity class for our study is AC_0, the class of circuits using a combination of and-, or- and not-gates where:

- There are m bits of input.
- The number of layers of the circuit (maximum number of gates connected in series), i.e. the circuit depth, is bounded by a constant.
- The width of the circuit (maximum number of gates in parallel) is a polynomial in m.
- A gate may use as input any number of outputs from gates in the previous layer (unlimited fan-in).

We write AC_0^k for AC_0 circuits respectively with k layers.

Strictly speaking, the formal definition of circuit complexity classes also involves a uniformity condition, referring to the resource bounds on a Turing machine required to generate descriptions of circuits in the class [13]. For our purposes, we do not have to consider this too deeply, except to say that it is related to the maximum number of rules allowed in a match table and hence one source of variation in computational power.

2.2 Modeling Match Tables

In this section, we consider pipelines of match tables as a computational model for Boolean functions. We demonstrate results relating the computational power of match tables to the size of their outputs and number of rules they are allowed to contain. In practical network switches, each rule has an associated action that may perform some simple modification on the input, but here we start with match rules without associated actions in order to focus on the power of matching, since they already determine most of the structure in the input-output relation of the function that is computed.

2.2.1 The Model

To start, we formalize the notions of match expressions and match tables.

Definition 1. A length m *match expression* is a ternary string composed of 0s, 1s and $*$'s (don't-care). Such an expression e is said to *match* an m-bit binary string s by defining inductively on m:

- For $m = 1$, e matches s if $e = *$ or $e = s$.
- For $m > 1$, let $e = ae'$ and $s = xs'$ where a and x are length 1. Then e matches s if a matches s and e' matches s'.

When referred to as a set, we will write 2^m for the set of m-bit binary strings.

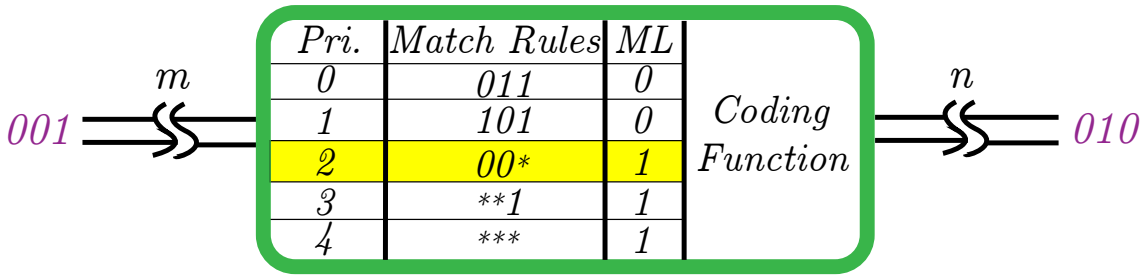

Figure 2.1: Match table from Example 4.
Priorities ($Pri.$) and the match line (ML) are shown. Since this is a single-match table, the output is the binary representation of the priority of the highest rule (highlighted) that matches the input.

Definition 2. A *match table* is a construct specified by the following:

- An *input* and an *output size* of m and n bits respectively, with $n \leq m$. These represent the number of wires flowing in and out of the table. The input and output of the table are therefore m- and n-bit binary strings.
- An ordered list of b distinct *match rules*. Each rule is a match expression of length m that attempts to match the input string.
- The position of a rule within the list in the range $[0, b-1]$ is its *priority*. By convention we say 0 has the highest priority and $b-1$ the lowest.
- For each input string, the table produces a *match line*, a b-bit vector whose i^{th} element is 1 if and only if the corresponding i^{th} rule matched.
- A binary *coding function* $c : 2^b \to 2^n$ that computes an n-bit result from the match line.

Figure 2.1 depicts the layout of a match table. The final coding function is necessary because when match tables are connected in practice, the available bandwidth between tables is typically

much smaller than the number of rules a table may hold within memory. The match line must be digested by the coding function c into a concise representation that can be reasonably consumed by subsequent logic. If c is allowed to be an arbitrary function, the matching process does not limit the computational power, since we can place one rule for every distinct input string. The priority ordering would not be considered in that case as all the rules are mutually exclusive. A more realistic specification is to utilize the priority order within c.

Definition 3. A match table is called a *single-match table* if it has these additional constraints:

- $b \leq 2^n$.
- The last match rule is always a length-m string of $*$'s, i.e. a match-all. This is called the *default rule*.
- c is the PRIORITY function. From the match line x, PRIORITY computes the n-bit binary representation of the position of the highest priority 1-bit in x.

This structure corresponds closely with modern switches that use the TCAM architecture, such as the Cisco Catalyst [1] and Juniper Networks EX [5] series, where a single matching rule with the highest priority will be the one whose action is executed. A single-match table is so named because matching only the rule with the highest priority is sufficient to determine the output of the table. This restriction is quite severe and makes it quite difficult to directly specify arbitrary computations. As we will see, even without actions, we can manipulate the ordering of rules in order to compute exactly the Boolean functions we desire. Since there are only n bits of output, the restriction $b \leq 2^n$ does not reduce the expressiveness of the table.

Example 4. Consider a single-match table with 3 input bits and 3 output bits. The rules, from highest to lowest priority, are 011, 101, $00*$, $**1$ and $***$. On input string 001, the match line is 00111, since the last three rules matched. The output is 010 since the rule $00*$ with priority 2 is highest-matched rule, and 010 is the binary representation of 2.

2.2.2 Pipelines

In real switches, multiple match tables can be laid out in a pipeline to enable step-wise processing of packet headers. For example, in practice we typically wish to match against header fields corresponding to one layer of the network stack, determine the protocol for the next layer, and then process accordingly.

Definition 5. An (n, t)-*pipeline* is a sequence of match tables, where:

- There are a total of t tables numbered $0, \ldots, t-1$.
- Each table has n bits of output.
- For $0 \leq i \leq t-1$, table i has $(i+1)\,n$ bits of input.
- The input bits to table i can be divided into $i+1$ times n-bit segments: bits $[0, n-1]$, $[n, 2n-1]$, etc. through $[in, (i+1)\,n-1]$. For $0 \leq j \leq i$ the segment $[jn, (j+1)\,n-1]$ is connected to the output of table $i-j-1$. Hence every table in the pipeline will receive as input all the inputs given to all previous tables in the sequence, and also the output of the immediately preceding table.

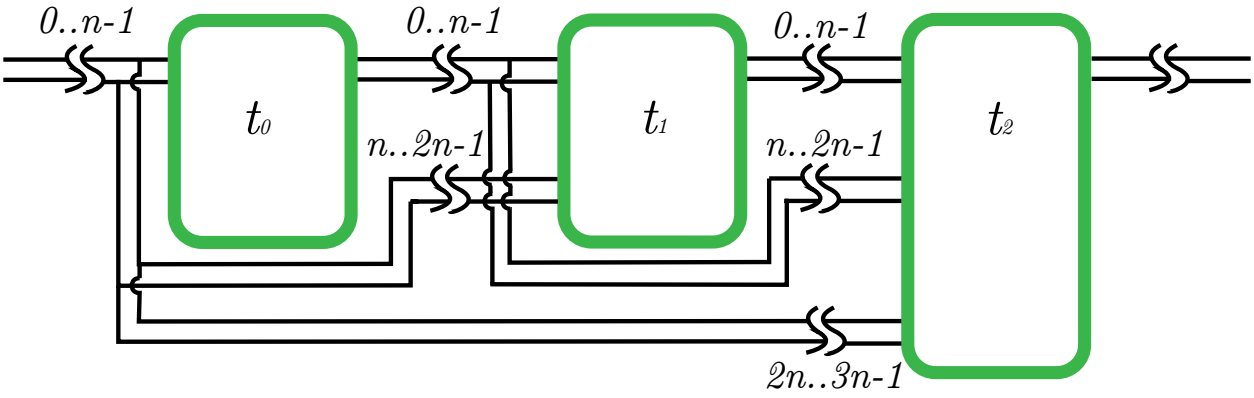

Figure 2.2: Example of an $(n, 3)$-pipeline.
At table i, for $0 \leq j \leq i$, input bits $[jn, (j+1)\, n-1]$ are connected to the output of table $i-j-1$.

- The input to the whole pipeline is deemed to be the input to table 0, and the output is deemed to be the output of table $t-1$. An (n, t)-*pipeline* therefore uses n bits for both the input and output.

Figure 2.2 gives an example of an $(n, 3)$-pipeline.

The input-output arrangement of tables in the pipeline quite accurately matches the structure of SDN-programmable pipelines that were conceptualized by [18] and implemented on switch processors such as the Intel Tofino chip [4]. A single wide bus of wires runs through every table in the pipeline. A table placed later can access any data placed on some wire at any earlier pipeline stage. Also, we will assume t is always a constant with respect to n. This again matches practice quite closely since the number of pipeline stages is inherently limited and fixed with respect to the size of packet headers that the switch is able to process.

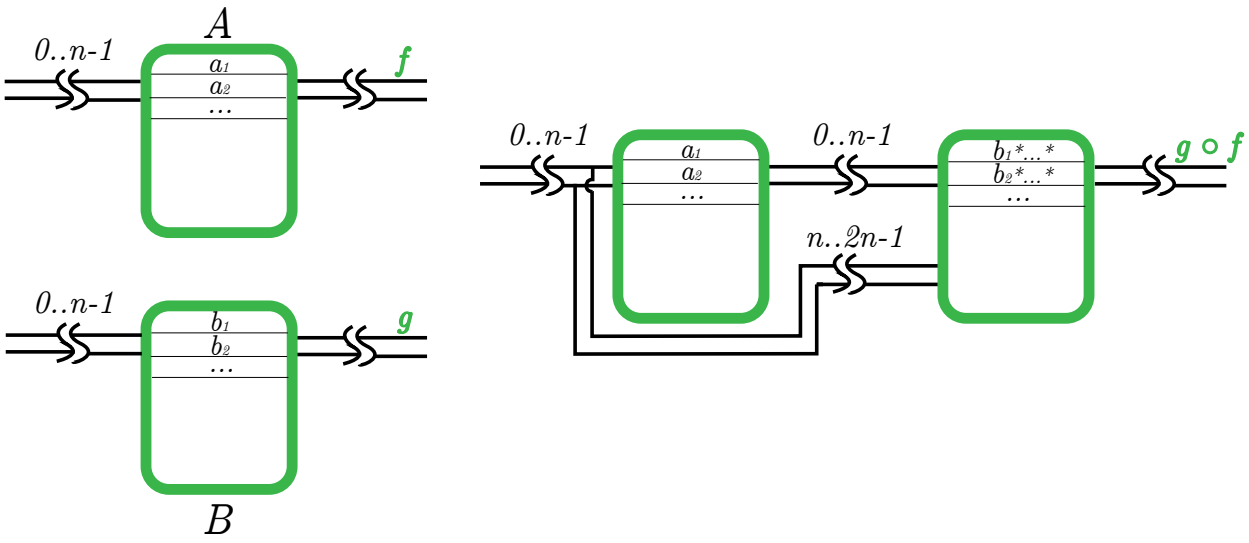

Figure 2.3: Composition of tables computing f and $g : 2^n \to 2^n$ as a pipeline computing $g \circ f$.

Definition 6. Similar to a single table, a pipeline *computes* a function $f : 2^n \to 2^n$ if the inputs and outputs of the pipeline match that of f.

Suppose we have tables A and B computing functions f and $g : 2^n \to 2^n$ respectively. We can construct a pipeline using A and B that computes $g \circ f$. First we extend B so that it has $2n$ input bits. We place B after A, and as per the rules for a table pipeline, the first n input bits are connected to the output of A, and the second n inputs are the inputs to A. We can ignore the extended bits since B only cares about the output of A. This is done by appending n don't-cares to each match rule in B. The output of B is unaffected. Furthermore, if B was a single-match table, B is still single-match after this modification. Figure 2.3 illustrates the composition process.

2.2.3 Single-match with Exponential Rules

In this and the following sections, we examine how the number of rules that tables are allowed to contain affects their computational power. We will first consider single-match tables, where the only restriction on the number of rules b is $b \leq 2^n$. We will show that with this assumption, single-match table pipelines with constant depth can compute every Boolean function $2^n \to 2^n$. Although this result is not surprising, the construction is not at all obvious as the output of a single-match table is limited to the binary representation of the priority of the highest rule that matched. We must also work around the consideration that negation, the specification that a string does *not* match an expression, cannot be implemented directly except through the default rule at the end of the table. All match rules must be specified positively.

Lemma 7. *Every bijection $f : 2^n \to 2^n$ can be computed by a single-match table with n bits of input and output.*

Proof. Consider a general match table with n bits of input and output. For each $x \in 2^n$, create a rule in the table whose string is exactly x and its priority has binary representation $f(x)$. Since f is a bijection, there is exactly one such rule for every priority. There will be 2^n rules, with priority ranging from 0 to $2^n - 1$. To make this table a single-match table, replace the last rule, which has priority $2^n - 1$, with a length-n don't-care $* \cdots *$ rule. This will have no effect on the output. The coding function is PRIORITY. For any x that is an input to the table, the output is $f(x)$. □

Corollary 8. *Since distinct output values require distinct rules in a single-match table, computing a bijection $f : 2^n \to 2^n$ requires 2^n rules.*

For more general $2^n \to 2^n$ functions, we need to first lay some groundwork. In fact, it is straightforward to prove that a lone single-match table is insufficient and a pipeline is required.

Example 9. Consider the function $f : 2^2 \to 2$ defined by $f(00) = f(11) = 0$ and $f(01) = f(10) = 1$. If a single-match table is able to compute f, its first rule (with priority 0) must be $**$. However $**$ matches 01, thus making the table give the incorrect output for 01.

To understand what is happening, we can precisely characterize the structure that a single match rule is able to capture.

Definition 10. *Hypercube in* 2^n. A subset C of 2^n forms a *hypercube* if for all $1 \leq i \leq n$:

- The i^{th} bits of all $x \in C$ are the same value, either 0 or 1; or
- For all $x \in C$, if its i^{th} bit is 0 (resp. 1), then there exists $y \in C$ that is the same as x but with 1 (resp. 0) as its i^{th} bit.

It is straightforward to see this definition captures exactly a set of strings that match a rule specified by 0, 1 and $*$. The key to analyzing the substructure of single-match tables as functions is the observation that a match rule corresponds to a hypercube in 2^n, and every output value of a table is mapped by a single rule. We must use pipelines if there are strings that do not form a hypercube but must be mapped to the same result. The process of constructing a single-match pipeline that computes an arbitrary binary function involves analyzing the structure of its mappings, as follows.

Definition 11. For any $f : 2^n \to 2^n$, the function f' is *a (domain) permutation of* f if $f' = f \circ h$ for some bijection $h : 2^n \to 2^n$.

Since a bijection can be implemented as one single-match table, we can characterize Boolean functions up to permutation by a structure that is invariant under permutation:

Definition 12. For any $f : 2^n \to 2^n$, its *pre-image function* $f_{\leftarrow} : 2^n \to \wp(2^n)$ is defined by $f_{\leftarrow}(y) = \{x \in 2^n \mid f(x) = y\}$, i.e. the pre-image of y under f. We write $F_{\leftarrow}$ for $\{f_{\leftarrow}(y) \mid y \in 2^n\}$, the set of all of f's pre-image sets. The *multiset of cardinalities of* $F_{\leftarrow}$ is the multiset $\{|S| \mid S \in F_{\leftarrow}\}$.

Let f^+ be a permutation of f. Observe that $F_{\leftarrow}$ and $F^+_{\leftarrow}$ have the same multiset of cardinalities. The structure of Boolean functions up to permutation is therefore determined uniquely by the multiset of cardinalities of their pre-image sets. Towards capturing this structure in a match table pipeline, we first make this observation:

Definition 13. A set $X \subseteq \wp(2^n)$ is *match-organized* if:

- For all $S \in X$, $|S|$ is a power of 2, possibly empty or a singleton.
- $|X|$ is a power of 2.

Lemma 14. *For every set $S \subseteq 2^n$, it is possible to partition S into a match-organized set.*

Proof. Consider the binary representation of $|S|$. Its 1-bits give a guide for a potential partitioning, except there may not be a power of 2 number of 1-bits. However, note that the binary representation of $|S|$ has length $\lfloor \log_2 |S| \rfloor + 1$ and there must exist a power of 2 between that value and $|S|$. Starting from a partitioning guided by the binary representation of $|S|$, we can successively break in half large partitions until we attain a power of 2 number of partitions. □

We make the key observation that a match-organized partitioning gives a way to potentially map all the elements of a set X to the same value by using single-match tables in sequence. Ideally, we want to use one match rule to capture each partition, since their sizes are powers of 2, and the collection of outputs from all these match rules are captured by one match rule in a subsequent table, as there are a power of 2 number of partitions. However, we are still one step short, since the exact members within a partition of a match-organized set may cause it to be unsuitable for capture. For example, a partition may not form a hypercube within 2^n. Thus we also require the following:

Definition 15. A set $X \subseteq \wp\left(2^n\right)$ is *initially-ordered* if:

- There is an ordering $\{S_1, S_2, \ldots, S_i\} = X$ in descending order of cardinality.
- Each $S \in X$ is a segment of 2^n, not necessarily initial. Concatenated together they form an initial segment. Viewing binary strings as numbers, this means $S_1 = [0, |S_1 - 1|]$, and $S_2 = [|S_1|, |S_1| + |S_2| - 1]$ and so on.

Now consider $f_{\leftarrow}$. For every y, let $F_M\left(y\right)$ be a match-organized partitioning of $f_{\leftarrow}\left(y\right)$. Note that the partitions $P \in \bigcup_y F_M\left(y\right)$ must be mutually disjoint since $f_{\leftarrow}$ gives sets of pre-images. The set $\bigcup_y F_M\left(y\right)$ may not be initially-ordered, however it can be made so by precomposing with a permutation. There is an F_M^+ where:

- For every y, $F_M^+\left(y\right)$ is a match-organized partition of $f_{\leftarrow}^+\left(y\right)$ for f^+ a permutation of f.
- In particular, f^+ is the permutation of f such that $\bigcup_y F_M^+\left(y\right)$ is initially-ordered.

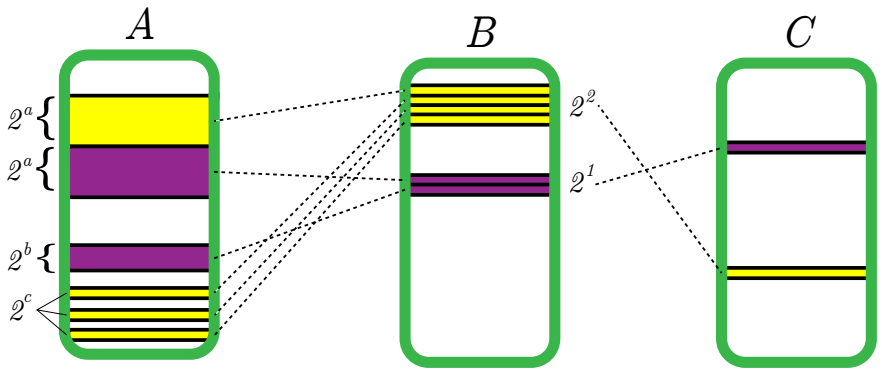

Figure 2.4: Three tables computing a conjugate of some arbitrary $f : 2^n \to 2^n$. In A, the domain of f is divided into sets of partitions that are match-organized and initially-ordered. The cardinalities of the partitions are descending ($a > b > c$). Partitions of the domain (having the same color) that ultimately must be mapped to the same output are captured by adjacent rules in B, again sorted by descending size. Since there is also a power of 2 number of partitions for each output value, sets of rules in B can be captured by single rules in C with the appropriate priority.

F_M^+ always exists for the properties described. Armed with this definition of $F_M^+(y)$ for each y, we are ready to prove our main theorem.

Theorem 16. *Every Boolean function $2^n \to 2^n$ can be computed by an $(n, 5)$-pipeline of single-match tables.*

Proof. Consider an arbitrary $f : 2^n \to 2^n$. We show that there exists a bijection h such that the function $h \circ f^+$ is computable with an $(n, 3)$-pipeline, where f^+ is the permutation associated with F_M^+. The theorem then follows because f^+ is a permutation of f and so is $h \circ f^+$, and by Lemma 7 at most one additional table at the start and one at the end of the pipeline is required to reverse the bijections. In the sense of group theory, we are producing a conjugate of f computable by an $(n, 3)$-pipeline. Although we are working with the definition of pipeline as given in Definition 5, the particular construction we will describe will not require a table to use outputs of any previous

tables besides the immediately preceding one. Hence all match rules can simply don't-care all the bits besides the first n input bits of their table.

Recall that the domain of f^+ is divided into partitions in $F_M^+(y)$ for every y. We now construct our pipeline computing $h \circ f^+$ for some h. For the first table, each rule will capture one partition within $\bigcup_y F_M^+(y)$ and assign a unique priority to the partition. By way of example, suppose the first partition is of size 2^k, then it must contain strings in the range $\left[\underbrace{0 \ldots 0}_{n}, \underbrace{0 \ldots 0}_{n-k}\underbrace{1 \ldots 1}_{k}\right]$ since we required $\bigcup_y F_M^+(y)$ to be initially-ordered. The rule to capture the partition is therefore $\underbrace{0 \ldots 0}_{n-k}\underbrace{* \cdots *}_{k}$. Suppose the second partition has size 2^k again, then the second rule would be $\underbrace{0 \ldots 01}_{n-k}\underbrace{* \cdots *}_{k}$. Suppose the third partition has size 2^{k-3}, then the third rule would be $\underbrace{0 \ldots 010000}_{n-k+3}\underbrace{* \cdots *}_{k-3}$. In general the rules are constructed by advancing the most significant prefix to the left of the sequence of don't-cares to accommodate the next partition taking up a segment in 2^n. To make this process successful, we have carefully constructed the partitions to always have size a power of 2, there is a power of 2 number of partitions within each $F_M^+(y)$ (match-organized), the partitions are sorted by descending cardinality, and each partition is an initial segment of 2^n (initially-ordered).

The second table will be a bijection. Each input value to this table represents a particular partition in $\bigcup_y F_M^+(y)$. We apply a bijection such that each rule matches exactly one value denoting a partition in $\bigcup_y F_M^+(y)$, and the rules that correspond to partitions belonging to the same y are placed consecutively. Essentially this is a regrouping of the partitions so that instead of being grouped by cardinality from the initially-sorted condition, they are grouped by y value. Note that there will be a power of 2 number of rules for each y value, since each $\left|F_M^+(y)\right|$ is a power of 2 from being match-organized. Within this table, we also order the group of rules with

the same y value based on their group size, where larger groups are placed with higher priority in the table.

Finally in the third table, we gather together each group of rules from the second table using a single rule. The method to do this is similar to that of the first table, since each group size is a power of 2 and they are sorted by descending size.

The three tables together calculate f^+ but with a permutation in the image, i.e. the pipeline computes $h \circ f^+$ for some bijection h. An example is given in Figure 2.4. □

2.2.4 Polynomial Rules

Now let us consider a more realistic limitation on the number of rules b in a single-match table, namely that $b = p(n)$ for some polynomial p. We can no longer compute every possible Boolean function (Corollary 8), but it is possible to relate the class of functions that are computable to a circuit complexity class that captures the expressive power of the tables and pipelines. In this case, we can use the circuit complexity class AC_0.

Theorem 17. PRIORITY *can be implemented using an* AC_0^3 *circuit, where the match line is the input. Consequently, an* (n, t)*-pipeline of single-match tables can be implemented as an* AC_0^{5t} *circuit of at most* n *inputs.*

Proof. To implement PRIORITY, recall that the match line is the bit vector representing matching rules in the table. Without loss of generality, we can assume the match line elements are sorted top to bottom in descending priority of match rule. For each bit i of the match line, put down a

not-gate connected to every bit above i. Then put down an and-gate with all these not-gates as inputs along with bit i. The output to all these and-gates signals the highest matching priority in unary representation. We can then build a binary encoder composed of or-gates connected to appropriate bits in the unary representation to convert the position of the first 1-bit to a binary number. In total the circuit consists of three layers.

Using the above result, we show that an m-input n-output single-match table can be implemented with an AC_0 circuit of five layers. We can encode each rule in the match table as an and-gate whose inputs are either connected directly to an input bit (for a 1-bit in the match rule), connected via a not-gate (for a 0-bit), or not connected (for don't-care). This takes two layers as these gates can be in parallel. The output of the gates form the match line, and then we can use the previous implementation of PRIORITY to form the output. Since $b = p(n)$ and $n \leq m$, we fall within the width requirement for an AC_0 circuit. This construction then extends straightforwardly to every table of a single-match table pipeline, with the inputs to the part of the circuit representing a table connected appropriately to outputs of previous parts. Since the number of tables in a pipeline is constant with respect to the size of the input and output of the pipeline, we can implement an (n, t)-pipeline using an AC_0 circuit of n inputs and $5t$ layers. □

Figure 2.5 illustrates examples of the above constructions. Since it is well known that AC_0 is not able to compute functions such as parity, it must be similarly impossible for single-match (n, t)-pipelines to compute them.

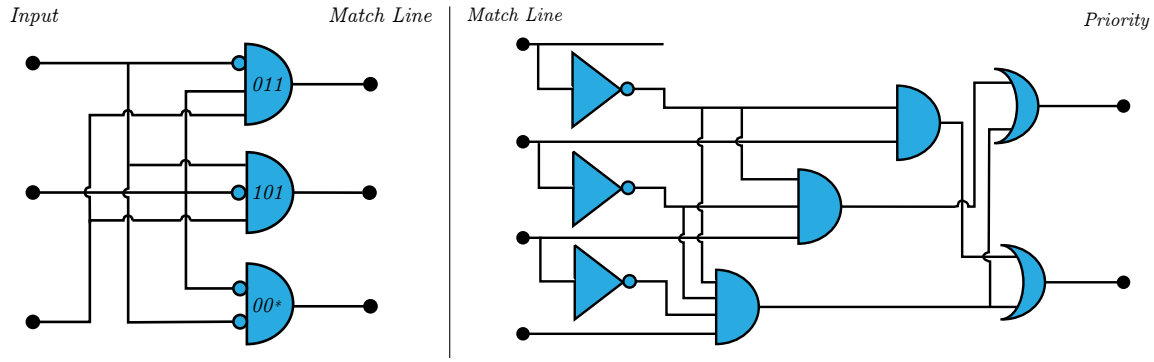

Figure 2.5: Left: an implementation of the first three rules from Example 4. Right: implementation of 4-bit PRIORITY in AC^3_0.
Highest match line element and most significant bit of priority are at the top.

2.2.5 Serial and Non-Serial Pipelines

Recall that Definition 5 allows tables in a pipeline to access the outputs of any preceding table, not just the one immediately before. It is possible to consider the relationship between this definition and a more straightforward one where only the output of the previous table can be accessed.

Definition 18. An (n, t)-pipeline is *serial* if the input of every table is connected only to the output of the immediately previous table, and ignores all other input bits.

Although Definition 5 reserves the possibility for pipelines to be non-serial, the examples we have constructed explicitly so far have all been serial, such as in Theorem 16. With a polynomial number of rules allowed within each table, we can in fact demonstrate that serial and non-serial single-match pipelines are equivalent up to a blowup in the number of tables and rules required in the serial case.

Theorem 19. *Any non-serial single-match $(n, 2)$-pipeline can be expressed as a serial single-match $(n, 6)$-pipeline. For both pipelines, the number of rules in each table is at most a polynomial in n.*

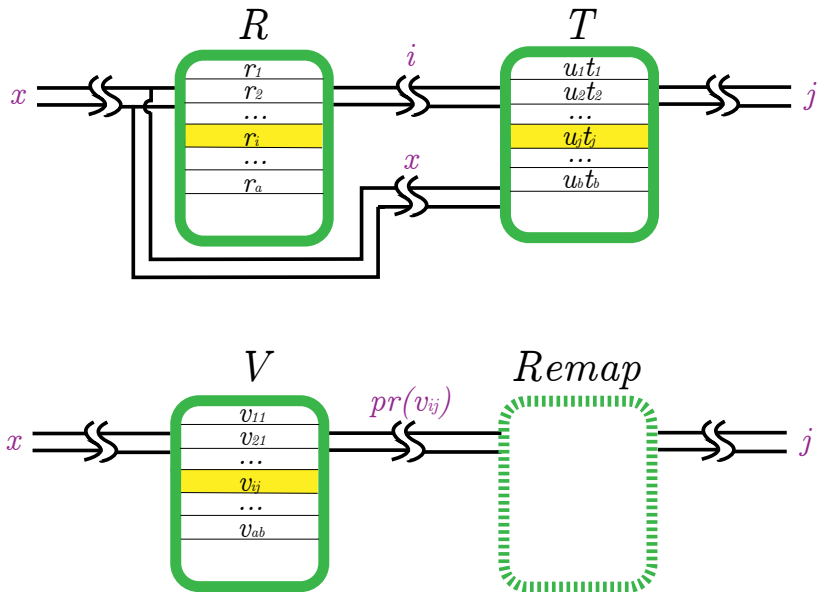

Figure 2.6: A non-serial $(n, 2)$-pipeline (above) equivalent to a serial $(n, 6)$-pipeline (below), with input/output values at each table.
Matching rules are highlighted. The priority of rule v_{ij} is denoted by $pr\left(v_{ij}\right)$. *Remap* is a serial pipeline of at most 5 tables (Theorem 16) that maps $pr\left(v_{ij}\right)$ to j.

Proof. Let R and T be the first and second tables respectively in the $(n, 2)$-pipeline as laid out in Figure 2.6. Suppose R contain rules $r_1, \ldots, r_a$. Since T's input is composed of the input and output of R, we write the rules of T as $u_1t_1, \ldots, u_bt_b$, where the u parts match on the output of R and the t parts match on R's input. The idea is to create a new table V whose rules combine those of R and T through a special kind of multiplication. Consider the case where an input $x \in 2^n$ to R produces the output i. It must be the case that r_i was the highest priority matching rule. The input to T is then ix, and suppose the output j is produced. T's highest matching rule must be u_jt_j and it must have the property that u_j matches i as a binary number and t_j matches the original input x.

Generalizing this idea, the rules of V are v_{ij} defined as follows for every $1 \leq i \leq a$ and $1 \leq j \leq b$:

$$v_{ij} = \begin{cases} r_i \sqcap t_j & u_j \text{ matches binary number } i \\ \bot & \text{otherwise,} \end{cases}$$

where $\sqcap$ is the bitwise conjunction

$\sqcap$	0	1	$*$
0	0	$\bot$	0
1	$\bot$	1	1
$*$	0	1	$*$

$\bot$ is the impossible case where the input and output of R are inconsistent, and it means the rules should be skipped from inclusion in V. The term $r_i \sqcap t_j$ also results in $\bot$ if the conjunction at any bit is $\bot$. The rules within V are ordered ascending by j first then i, i.e. $v_{11}, v_{21}, \ldots, v_{a1}, v_{12}, v_{22}, \ldots, v_{ab}$. We can show that V preserves the priority output ordering of RT. In other words, on inputs x and x', $RT(x) \leq RT(x')$ if and only if $V(x) \leq V(x')$. In the "if" direction, let v_{ij} and $v_{i'j'}$ be

the highest matching rules in V on x and x' respectively. Since $V(x) \leq V(x')$, it must be that $j \leq j'$. Since $RT(x)$ and $RT(x')$ are the priorities of the rules $u_j t_j$ and $u_{j'} t_{j'}$ in T, we have $RT(x) = j \leq j' = RT(x')$. "Only if" follows similarly.

Hence we are able to translate the RT pipeline into a single table V that preserves the priority order structure, except we now have "repeats" of up to ab possible output values instead of T's b possible output values. To completely realize the original function, we need to remap all output values that arose from rules $v_{1j}, \ldots, v_{aj}$ of the same j to priority j. This can be done with at most 5 subsequent single-match tables in a serial pipeline as given in the proof of Theorem 16: we permute the rules so that sets having the same j are match-organized, capture the sets into single values, then permute again to give the correct j values as output. Each new table does not require more than the number of rules ab in V, since at each stage we are either permuting or mapping sets of output values to a single value. Since a and b are polynomials of n, so is ab. □

This theorem extends more generally to the case of translating non-serial single-match (n, t)-pipelines. The resulting serial pipeline contains a constant number of tables with respect to n, with the per-table rules count being at most the product of the number of rules in each original table. Since t is also a constant compared to n, this is still polynomial.

2.2.6 Multimatch Tables

So far we have considered match tables with what we call the single-match restrictions, which corresponds to practice. However our definitions of tables and pipelines are more general. Recall

that one hallmark of the single-match table is that its coding function is PRIORITY. By relaxing this restriction, we can consider a more general class of tables.

Definition 20. A match table is *multimatch* if:

- With b rules in the table, the coding function c can be implemented by a circuit in AC_0 with b inputs.
- The number of outputs to the table is at most $\lceil \log b \rceil$, enforced by requiring the width of the last layer of the circuit implementing c to $\lceil \log b \rceil$.

Single-match tables are in fact a subset of multimatch tables, since PRIORITY can be implemented with an AC_0 circuit and there are at most 2^n rules in the table for output size n. Multimatch is so named because multiple rules being matched could affect the table output, depending on the coding function, besides the one with the highest priority. As mentioned in Section 2.2.1, there is a practical limitation on how much data can be transported between tables, so a realistic limitation is required on the size of the table output.

Example 21. Consider two fields a and b each occupying a disjoint segment of bits in n-bit strings. We would like to consider b only if a is within range $[1, r]$. Using a single-match table, every relevant combination of values of a and b requires one rule, potentially a total of $r\,|b|$ rules. Alternatively, we could use a pipeline to map a down to $\{0, 1\}$ and then match on this result and b. With multimatch, we can have a single table with separate sets of rules matching a and b, therefore needing at most $r + |b|$ rules. It is then up to the logic in the coding function to consider rules that match from both sets.

Although multimatch has not been implemented in practice, since it requires complex logic, it is instructive in demonstrating how our model accommodates possible developments in match table design. For convenience, we will write MM for the class of functions computable by pipelines composed of multimatch tables.

Definition 22. To characterize multimatch tables and pipelines using circuit complexity, we introduce a circuit complexity class XC. It is similar to AC_0 in having a constant number of layers, except:

- There are $\lceil \log b \rceil$ bits of input and output, for some b.
- The width of the circuit is $O(b)$, except at the last layer, which has width at most $\lceil \log b \rceil$.

We write XC^k for the class of XC circuits with k layers.

Theorem 23. *As complexity classes of functions,* MM *and* XC *are the same.*

Proof. The proof is by inclusion in both directions.

We first show $\mathsf{MM} \leq \mathsf{XC}$ by giving a translation of a single multimatch table to a constant-depth circuit. The translation of pipelines follow trivially through composition. Let a single MM table have $\lceil \log b \rceil$ bits of inputs, thus a maximum of $O(b)$ distinct rules. These inputs are exactly the same as those of the XC circuit we will construct.

For every match rule, put down an and-gate with $\lceil \log b \rceil$ inputs such that for each i^{th} input:

- It is connected to bit i of the circuit input if the rule asks for 1 at bit i.

- It is connected to a not-gate whose input is bit i if the rule asks for 0 at bit i.
- It is unconnected if the rule is $*$ at bit i.

These and-gates together produce the match line. Since the coding function c of the MM table can be implemented using an AC_0 circuit with the match line as the input, we just put down this circuit as the second half of the XC circuit. Recall that c takes the $O(b)$ bits of the match line as input but is limited to having $\lceil \log b \rceil$ bits of output. Its output matches the output size restriction of an XC circuit with $\lceil \log b \rceil$ bits of input, and its $O(b)$ bits of input fits in the width restriction of XC.

On the other hand, we show $\mathsf{XC} \leq \mathsf{MM}$ by encoding an entire circuit as one multimatch table. Suppose the XC circuit has $\lceil \log b \rceil$ bits of input for some b. For every bit $1 \leq i \leq b$, create a rule in the table such that the rule matches if and only if the i^{th} bit is 1, and don't-care the other bits. The match line will therefore mirror the input. There are $\lceil \log b \rceil$ input bits, which is also the size of the match line, and this is $O(b)$. Encode the entire XC circuit in the coding function c of the table. The circuit cannot be wider than $O(b)$, while c is allowed to have input size $O(b)$, so the maximum width enforced by AC_0 is sufficient. On the other hand, the circuit has at most $\lceil \log b \rceil$ outputs, which matches the output size restriction for a multimatch table. □

We have therefore shown $\mathsf{MM} = \mathsf{XC}$. In particular, an MM pipeline with k multimatch tables can be implemented in $\mathsf{XC}^{O(k)}$, and an XC^k circuit can be implemented as a single MM table whose coding function c is in AC_0^k. It is worthwhile to point out that prior to the last layer, the width $O(b)$ allowed by XC circuits is very permissive compared to the size of the input $\lceil \log b \rceil$. For example, any AC_0 circuit would be allowed since any polynomial in $\lceil \log b \rceil$ is $O(b)$. However,

with the last layer restricted to width $\lceil \log b \rceil$, only AC_0 circuits with the same restriction are in XC. XC captures the idea of multimatch tables in allowing unrestricted logic instead of fixed PRIORITY, but the amount of data transported between tables must still be concise compared to the size of the input. In practice, if multimatch were to be implemented, the complexity of the circuit would be limited by the timing that the coding function must achieve and the number of rules that can be stored within each table.

2.3 Discussion and Future Direction

In this chapter, we believe we have for the first time classified the computational power of widely-adopted mechanisms of match tables and pipelines using well-developed circuit complexity theory. Although there is a rich body of applied work in the field, including those such as [37] that analyze packet headers as a structured bag of bits, none have approached the problem of expressivity of match-action tables from the direction of complexity theory. We have given a theoretical model, the single-match table and pipeline, which corresponds to how packet header matching is done in practice (without actions), and demonstrated how results on computational power could be inferred. This model is a special case of the general definition of match tables we have developed, which we believe can accommodate future developments to how match tables are used in practice. Besides deriving complexity bounds, we have also demonstrated how our model accommodates reasoning about pipelines formally to yield optimization results.

It remains to be seen how match-action evolves in practice. The current behavior, modeled by our single-match tables, is inherent in the hardware TCAMs used to perform match efficiently in

parallel. The convention is to always return the highest priority rule that matches, and they are not able to match against negated rules. Proposals such as [44, 52] have been made to improve the functionality of TCAMs. As demonstrated by our multimatch example, our model is flexible enough to consider future match-action behavior through restriction to circuit complexity classes. It will be worthwhile to return here once we have a clearer picture of what the future capabilities of match-action hardware will be.

2.3.1 Actions

In reality, match tables used in network switches have *actions* associated with each rule that are executed when the rule is matched. Actions are commands that could modify bits in the packet header or other metadata in the dataplane in a simple way. We decided to focus solely on the logic of matching. Essentially, this means our tables have no actions and classification of packets rely on a read-only model of the header, where we do not modify or write additional state. We took this decision in order to study the power provided by the restricted but concise mechanism of matching that is fundamental to all implementations of match-action pipelines. In fact, our model of match tables can already accommodate action by simply stating that they are incorporated within the coding function c of each table. Future study of how actions fit in our theoretical model would involve considering different complexity constraints so that we can make interesting observations based on an orderly classification of action functions.

2.3.2 Future Work

We have made the argument for a rigorous and formal consideration of the computational power of match tables within network switches. There are three chief directions along which we envisage our work could be developed. The first is aiding in the creation of new dataplane paradigms beyond match-action. Since there is a trade-off between the logical complexity of the dataplane and the speed of processing that can be achieved by the hardware, a faster switch may not be able to implement more advanced network algorithms that require more complicated logic. Reasoning about the packet classification mechanism formally at least provides guidance on what computations are and are not possible.

The second direction is in the development of minimization algorithms for match table configurations. Through complexity-theoretic arguments it may be possible to determine precise bounds for the resources that are required to compute functions used by network features. This would inform device designers on whether it would be futile to continue optimizing features to use fewer resources as the designs may already be asymptotically optimal. On the other hand, proving better asymptotic bounds could lead to more efficient algorithms and designs than current implementations.

Finally, we envision bridging the gap between the current theory done in SDNs, focused mainly on semantics and programming language theory, and the actual computational power of underlying hardware. While current theory is capable of answering questions such as how specification languages should be designed for SDNs to be extended with desirable network features, it does not offer a way to formally reason about the performance of the implementations taking

into account the structure of the system. By modeling packet classification logic formally and approaching the field from a complexity-theoretic angle, we may be able to expand the set of questions we can answer from what functions are possible to what sort of functions will have efficient implementations.

CHAPTER 3

MATCHKAT

We now turn instead to the language-theoretic perspective of formalizing match-action. In this chapter, we present MatchKAT, a Kleene algebra with tests (KAT) [41] that employs match expressions on binary strings as tests. It is able to encode match and action while having a metatheory closely related to NetKAT. While NetKAT is appropriate for specifying and reasoning about global network policies, MatchKAT is more concerned with match-action, and hence the local behavior of switches. Leveraging results from NetKAT, we are able to show MatchKAT is sound and complete with respect to its own packet filtering semantics. Through a translation to NetKAT, decision procedures such as those in [28] can also be adapted to MatchKAT. Although this chapter will mainly introduce the basics of MatchKAT and its metatheory, the application-level motivation is that in the future we may be able to give a formal semantics for match-action as used in network switches. It is hoped that MatchKAT will eventually allow for algebraic reasoning on local switch configurations similar to NetKAT for global network policies, which could allow applications such as proving the equivalence of match-action switch configurations and decompiling match-action rules to higher-level policies. Previous attempts at reasoning with match expressions on binary strings in the context of packet classification, such as in [36, 37], have been more ad hoc and without a formal metatheory.

Our contributions can be summarized as follows:

- We give an algebraic formalization of ternary (0, 1, don't-care) match expressions on binary strings (Section 3.1.2). Although others have studied aspects of the theory of match

expressions, for example [37], we present it here in a formal algebraic language, as match expressions will be integral to the formalization of MatchKAT.

- We give the syntax and a packet filtering semantics for MatchKAT (Section 3.2) and show how it is able to reason about match-action (Section 3.2.4). Despite being related to NetKAT, MatchKAT is able to encode operations that would require much longer expressions in NetKAT.
- We show that MatchKAT has a sound and complete equational theory with respect to its semantics by leveraging a correspondence with the dup-free fragment of NetKAT (Sections 3.3 and 3.4). The problem of deciding equivalence between MatchKAT terms is shown to be PSPACE-complete (Section 3.4.1).

3.1 Preliminaries

In this section we give some background on KAT, as well as a formal presentation of match expressions on binary strings. We will defer discussion on NetKAT to Section 3.3 when we clarify its connection with MatchKAT.

3.1.1 Kleene Algebras with Tests

A Kleene algebra (KA) [40] has a signature $(P, +, \cdot, ^\star, 0, 1)$ with the following axioms for $p, q, r, x \in P$:

$$p + (q + r) = (p + q) + r \qquad p(qr) = (pq)r$$

$$p + q = q + p \qquad \mathbf{1} \cdot p = p \cdot \mathbf{1} = p$$

$$p + \mathbf{0} = p + p = p \qquad p \cdot \mathbf{0} = \mathbf{0} \cdot p = \mathbf{0}$$

$$p(q + r) = pq + pr \qquad (p + q)r = pr + qr$$

$$\mathbf{1} + pp^\star \leq p^\star \qquad q + px \leq x \;\Rightarrow\; p^\star q \leq x$$

$$\mathbf{1} + p^\star p \leq p^\star \qquad q + xp \leq x \;\Rightarrow\; qp^\star \leq x$$

where $p \leq q \Leftrightarrow p + q = q$. A Kleene algebra is therefore an *idempotent semiring* over P with an extra * operator.

A Kleene algebra with tests (KAT) [41] has a signature $(P, B, +, \cdot, ^\star, 0, 1, ^-)$ such that

- $(P, +, \cdot, ^\star, 0, 1)$ is a Kleene algebra.
- $(B, +, \cdot, ^-, 0, 1)$ is a Boolean algebra.
- $(B, +, \cdot, 0, 1)$ is a subalgebra of $(P, +, \cdot, 0, 1)$.

P is usually called the set of *actions* while members of B are *tests.* The axioms of Boolean algebra

are as follows for tests a and b:

$$a + (bc) = (a + b)(a + c) \qquad ab = ba$$

$$a + \mathbf{1} = \mathbf{1} \qquad a + \overline{a} = \mathbf{1}$$

$$a \cdot \overline{a} = \mathbf{0} \qquad aa = a$$

Note that 0 and 1 are the identities of $+$ and $\cdot$ respectively and 0 is an annihilator for $\cdot$. Terms of the KAT are then freely generated by P and B with the operators. KAT is well-covered in literature [23, 41, 43], but we highlight that a KAT can possess interesting equational theories, as we will be studying later. It is possible to axiomatically derive equivalences between KAT terms, as well as assign some denotational semantics to them. We say that an equational theory is *sound* with respect to those semantics if all provably equal terms have equal semantics, and *complete* if proofs of equivalence exist for any two terms that are semantically equal. The decision problem of whether two terms are equal can also be studied and its complexity classified. Since a KAT can often be used to encode programs, the equational theory is important for studying program equivalence.

3.1.2 Match Expressions

We give a formalization of match expressions on binary strings as found in match-action tables implemented in network switches. These expressions will form the tests within MatchKAT.

The set $\mathbb{E}$ will be the set of all match expressions that we will define. The syntax of expressions is found in Figure 3.1. Set $\mathbb{E}$ is equipped with a concatenation operation $@$ and is stratified into

$$E_0 ::= \bullet \mid \bot_0 \mid E_0 + E_0 \mid E_0 \sqcap E_0 \mid \overline{E}_0$$
$$E_{n+1} ::= E_n \mathbin{@} 1 \mid E_n \mathbin{@} 0 \mid E_n \mathbin{@} \mathrm{x} \mid \bot_{n+1} \mid E_{n+1} + E_{n+1} \mid E_{n+1} \sqcap E_{n+1} \mid \overline{E}_{n+1}$$
$$\top_0 \triangleq \bullet \qquad \top_n \triangleq \underbrace{\mathrm{x} \ldots \mathrm{x}}_{n} \text{ for all } n > 0$$

Figure 3.1: Syntax of match expressions.

subsets E_n for all $n \geq 0$, such that $\mathbb{E} \triangleq \bigcup_n E_n$. Each E_n is said to be *the set of match expressions with width* n, and has an algebraic signature $(E_n, +, \sqcap, \overline{\ }, \bot_n, \top_n)$. The operator $+$ is *union*, $\sqcap$ is *intersection*, $\overline{\ }$ is *complementation*, and $\bot_n$ and $\top_n$ are identities of $+$ and $\sqcap$ respectively. Terminology-wise, we will refer to the size of binary strings to be matched on as the *width*, reserving the word *length* for later quantifying the size of match expressions themselves.

The actual members of the set E_n are defined inductively on the width n. In the base case, there are the empty $\bullet$ and bottom $\bot$ expressions. Note that we distinguish between the empty expression and the empty binary string ϵ. E_{n+1} is then built from members of E_n with concatenation $@$. Notationally we will usually elide this operator.

Intuitively, 1, 0 and x will correspond to matching 1, 0 or anything (don't-care) at a given position in the binary string, $\bullet$ is for matching ϵ, and all instances of $\bot$ matches nothing. We use x to avoid confusion with the * operator of KATs. This intuition of an expression matching bits will be made formal shortly. Since E_n at each width has the signature $(E_n, +, \sqcap, \overline{\ }, \bot_n, \top_n)$, it is also extended freely with expressions built from $+$, $\sqcap$ and $\overline{\ }$.

Axiomatically, for every n we require $(E_n, +, \sqcap, \overline{\ }, \bot_n, \top_n)$ to be a Boolean algebra. The Boolean algebra axioms determine the behavior of $\bot_n$ and $\top_n$ when combined with the Boolean operators. However, additionally we also need axioms that relate 1, 0, x and concatenation. They

$$\overline{0} = 1$$

$$\begin{array}{rl}
1 + 0 = 0 + 1 = \mathrm{x} & 1 \sqcap 0 = 0 \sqcap 1 = \bot_1 \\
\bullet e = e \bullet = e & \bot_{n_2} e = e \bot_{n_2} = \bot_{n_1+n_2} \\
\overline{ee'} = \overline{e} \top_{n_2} + \top_{n_1} \overline{e'} & (e_1 e_2)\, e_3 = e_1\, (e_2 e_3) \\
e_1\, (e_2 + e_3) = e_1 e_2 + e_1 e_3 & e_1\, (e_2 \sqcap e_3) = e_1 e_2 \sqcap e_1 e_3 \\
(e_1 + e_2)\, e_3 = e_1 e_3 + e_2 e_3 & (e_1 \sqcap e_2)\, e_3 = e_1 e_3 \sqcap e_2 e_3
\end{array}$$

Figure 3.2: Axioms for match expressions.
e_1, e_2, and e_3 are expressions in $\mathbb{E}$, e is in E_{n_1}, and e' is in E_{n_2}. These are in addition to axioms that enforce $(E_n, +, \sqcap, \bar{}, \bot_n, \top_n)$ for each n as Boolean algebras.

are found in Figure 3.2. In particular, $\top_n$ is syntactic sugar for the wildcard expression matching any string of width n.

To formalize the semantics of match expressions, we come back to the notion of width and length as mentioned at the start of this section. Let 2^n be the set of binary strings of width n, and in particular let $2^0 = \{\epsilon\}$ be the set only containing the empty string ϵ. An expression $e \in E_n$ is said to have width n and matches strings in 2^n.

We can model what it means for a match expression to match a binary string by interpreting an expression as the set of all strings that match it. For some expression $e \in E_n$, its interpretation $(\!|e|\!)_n$ is the set of all strings in 2^n that matches e. The definition of of the interpretation is as follows for $e, e' \in E_n$, $e_1 \in E_{n_1}$ and $e_2 \in E_{n_2}$. It is given inductively and parameterized by n:

$$\begin{array}{llllll}
(\!|\bullet|\!)_0 \triangleq \{\epsilon\} & (\!|\mathrm{x}|\!)_1 \triangleq \{0,1\} & & (\!|e + e'|\!)_n \triangleq (\!|e|\!)_n \cup (\!|e'|\!)_n \\
(\!|0|\!)_1 \triangleq \{0\} & (\!|\bot_n|\!)_n \triangleq \emptyset & & (\!|e \sqcap e'|\!)_n \triangleq (\!|e|\!)_n \cap (\!|e'|\!)_n \\
(\!|1|\!)_1 \triangleq \{1\} & (\!|\overline{e}|\!)_n \triangleq 2^n - (\!|e|\!)_n &
\end{array}$$

$$(\!|e_1 e_2|\!)_{n_1+n_2} \triangleq \left\{ x_1 x_2 \mid x_1 \in (\!|e_1|\!)_{n_1}, x_2 \in (\!|e_2|\!)_{n_2} \right\}$$

Example 24. Since $\top_n \triangleq \underbrace{\mathrm{x} \ldots \mathrm{x}}_{n}$ for $n > 0$, $(\!|\top_n|\!)_n = 2^n$ by derivation from the definitions $(\!|\mathrm{x}|\!)_1$ and $(\!|e_1 e_2|\!)_{n_1+n_2}$.

For any $S \subseteq 2^n$, we say that e *captures* S if and only if $(\!|e|\!)_n = S$. When reasoning with binary strings, we often wish to refer to individual bits within the string. For $b \in 2^n$, we write $b\,[i]$ for the i-th bit of b, and $b\,[i \leftarrow x]$ for the string that is b but with x as the i-th bit. Conventionally, we will start bit indices at 1, strings are read left-to-right, and the most significant bit is to the left whenever a string is interpreted as a binary number.

It can be seen that a match expression has the same width as the strings it matches, and match expressions of different widths cannot mix as they belong to different languages. On the other hand, a match expression's length could be arbitrary in size, and it is a measure of the complexity of the expression.

Example 25. Suppose we are interested only in counting occurrences of $\sqcap$ and $+$. For some even width $2n$, consider the expression

$$\prod_{i=1}^{n} \left(\top_{i-1} 0 \top_{n-1} 0 \top_{n-i} + \top_{i-1} 1 \top_{n-1} 1 \top_{n-i}\right).$$

It captures exactly the set $\left\{bb \mid b \in 2^{2n}\right\}$ and its length is $O(n)$ since that many $\sqcap$ and + operators were used. An equivalent expression that captures the same set is

$$\overline{\sum_{i=1}^{n} \left(\top_{i-1} 1 \top_{n-1} 0 \top_{n-i} + \top_{i-1} 0 \top_{n-1} 1 \top_{n-i}\right)},$$

which is also length $O(n)$. However, if we are only allowed to use $+$, but not $\sqcap$ and complementation, an expression capturing this set must have length at least exponential in n. This is because each string of the form bb must occur in the match expression explicitly.

The same match expression could have different lengths depending on which operators we are interested in counting. This is useful for the application of relating match expression length to the complexity of a match program in a network switch. Some operations may be expensive, such as $\sqcap$, while concatenation can be "free" and does not need to be counted as it is simply multiple hardware units placed in parallel.

We end the discussion on match expressions by speaking briefly on the soundness and completeness of the equational theory of match expressions with respect to the binary strings model. Proving soundness is a straightforward albeit tedious task. We simply go through each axiom and show that the expressions on both sides of the equality capture the same set. Completeness is also fairly easy. We can decide whether two match expressions e and e' are equivalent by expanding both to their disjunctive normal forms and then eliminate all occurrences of $\sqcap$. Equality can then be checked if the expressions are identical up to commutativity of $+$. Unfortunately, this axiomatic proof of equivalent introduces an exponential blowup. A more tractable, co-NP decision procedure is to non-deterministically guess a string in the symmetric difference of $(\!|e|\!)_n$ and $(\!|e'|\!)_n$, which succeeds if and only if e and e' are not equivalent.

3.2 MatchKAT

Our discussion of MatchKAT starts with the intuition that each width-n space of match expressions E_n can be seen as a Boolean algebra over n variables. A binary string corresponds to an assignment of truth values and a match expression is a propositional formula that is satisfied by exactly the assignments of matching binary strings. This is an alternative way to think of the

underlying model that we are working with as our definition of MatchKAT evolves.

For the rest of this section we will let n be a constant positive integer, which as before was used to denote the widths of binary strings, but now we will refer to it as the *packet size*.

3.2.1 Definitions

Intuitively, MatchKAT is a KAT whose terms operates on the finite state space created by n bits of random access memory occupied by a packet header. It is defined by:

- Primitive tests are match expressions in E_n, matching the whole memory at once. For $1 \leq i \leq n$ and $k \in \{0, 1\}$, we will adopt the shorthand $i \simeq k$ for the match expression $\top_{i-1} k \top_{n-i}$, which solely tests whether the i-th bit is k.
- Primitive actions are in the form $i \leftarrow k$, for $1 \leq i \leq n$ and $k \in \{0, 1\}$, intended to mean assigning 0 or 1 to bit i.
- The operations are plus $+$, composition $\cdot$, complementation $\overline{p}$, and Kleene star $p^\star$. For tests, $+$ and $\cdot$ correspond respectively to $+$ and $\sqcap$ within match expressions E_n (not concatenation within $\mathbb{E}$). Sometimes we may write composition as $\sqcap$ between terms that are known to be tests.
- The identity of $+$ is $\bot_n$, and for $\cdot$ it is $\top_n$. We can elide the subscripts when they are n as an instance of MatchKAT is only concerned with a fixed packet size.

We admit all the axioms required of a KAT, and those of match expressions presented previously. This is already a sufficient definition for a valid KAT. However, we require additional *packet*

algebra axioms in order to allow commutation of actions and tests on unrelated memory locations, and absorption of related ones. For $i \neq j$:

$$i \leftarrow k \cdot j \leftarrow k' \equiv j \leftarrow k' \cdot i \leftarrow k \qquad i \leftarrow k \cdot j \simeq k' \equiv j \simeq k' \cdot i \leftarrow k$$
$$i \leftarrow k \cdot i \simeq k \equiv i \leftarrow k \qquad i \simeq k \cdot i \leftarrow k \equiv i \simeq k$$

We use $\equiv$ to denote the equivalence of terms in order to avoid ambiguity with $\simeq$ and $=$. Readers familiar with NetKAT may wonder why we do not require axioms of the forms $i \simeq k \cdot i \simeq k \equiv i \simeq k, k \neq k' \implies i \simeq k \cdot i \simeq k' \equiv \bot$, and $\sum_k i \simeq k \equiv \top$. These are derivable theorems within the algebra of match expressions.

Example 26. If $k \neq k'$, then

$$i \simeq k \cdot i \simeq k' \equiv (\top_{i-1} k \top_{n-i}) \sqcap (\top_{i-1} k' \top_{n-i}) \equiv \top_{i-1} (k \sqcap k') \top_{n-i} \equiv \bot$$

as $k \sqcap k' \equiv \bot$.

3.2.2 Packet Filtering Semantics

We now discuss the semantics of MatchKAT as applied to packet forwarding. Naturally, the n bits of state we have in mind will be modeled by packet headers, which we will just refer to as packets. The following semantics operate on sets of packets at both input and output, intending to model the packets that arrive at a switch and what packets will be forwarded after filtering by the MatchKAT term. We denote the set of packets as Pk, which we will represent as strings in 2^n

(so really $Pk = 2^n$). The semantics of a MatchKAT term e is a function $[\![e]\!] : \mathcal{P}(Pk) \to \mathcal{P}(Pk)$:

$$
\begin{array}{rclcrcl}
[\![\bot]\!](P) & \triangleq & \emptyset & & [\![p+q]\!](P) & \triangleq & [\![p]\!](P) \cup [\![q]\!](P) \\
[\![\top]\!](P) & \triangleq & P & & [\![p \cdot q]\!](P) & \triangleq & ([\![q]\!] \circ [\![p]\!])(P) \\
[\![a]\!](P) & \triangleq & P \cap (\!|a|\!),\ a \in E_n & & [\![\overline{p}]\!](P) & \triangleq & Pk - [\![p]\!](P) \\
[\![i \leftarrow k]\!](P) & \triangleq & \{\pi\,[i \leftarrow k] \mid \pi \in P\} & & [\![p^\star]\!](P) & \triangleq & \bigcup_{k \geq 0} [\![p]\!]^k(P)
\end{array}
$$

We call this the *packet filtering semantics* as the semantic functions are transformers on sets of packets. Suppose a network switch is modeled by a MatchKAT term. The output denotes the set of packets that is produced, given some set of input packets. The next sections will give examples of how MatchKAT terms can be used in practice, while later in Section 3.4 we will show the equational theory of MatchKAT is sound and complete with respect to this semantics, and deciding equivalence is PSPACE-complete. An improvement on this time bound for many cases in practice was shown in the recent work on Guarded Kleene algebra with tests (GKAT) [61], where it is possible to decide the equational theory of a fragment of a KA expressible in that language in nearly linear time.

3.2.3 Encoding Actions on Packets

In match-action, "match" refers to matching of binary data in packet headers, which we have covered so far. On the other hand, "actions" in this context refer to simple modifications of the packet header. Once a rule is matched, its action is performed and the switch then forwards (or keeps on processing) the packet based on the updated header fields. For example, there may be a *port* field specifying the egress port to which the packet should be moved. It is possible to encode modifications on fields in MatchKAT.

3.2.3.1 Direct and indirect assignment/test.

Assignment/test of a constant value over a range of bits can be performed by assigning/testing the value's binary representation.

Example 27. Assigning the value 6 (binary 110) to bits 2 through 4 can be written as $2 \leftarrow 1 \cdot 3 \leftarrow 1 \cdot 4 \leftarrow 0$.

Test and assignment of a range of bits against another range can be done in a single match expression bitwise.

Example 28. To assign the values contained in bits 1 through 3 to bits 4 through 6, we can write

$$\begin{aligned} & (1 \simeq 0 \cdot 4 \leftarrow 0 + 1 \simeq 1 \cdot 4 \leftarrow 1) \\ \cdot \quad & (2 \simeq 0 \cdot 5 \leftarrow 0 + 2 \simeq 1 \cdot 5 \leftarrow 1) \\ \cdot \quad & (3 \simeq 0 \cdot 6 \leftarrow 0 + 3 \simeq 1 \cdot 6 \leftarrow 1)\,. \end{aligned}$$

We can simply replace $\leftarrow$ with $\simeq$ above instead to test for equality.

3.2.3.2 Arithmetic.

Since we know in advance the packet size n and the range of bits to operate on, we can encode arithmetic on sets of bits in MatchKAT through simple fixed-width algorithms. We give the increment operation just as an example.

Example 29. Suppose a range of bits contains a binary value we wish to increment. We write $[i \ldots j]^{++}$ for the term that increments the value contained in bits i through j. It can be defined

inductively as:

$$[i \dots j]^{++} \triangleq \begin{cases} \top & j < i \\ j \simeq 0 \cdot j \leftarrow 1 + j \simeq 1 \cdot j \leftarrow 0 \cdot [i \dots j-1]^{++} & \text{otherwise} \end{cases}$$

3.2.4 Encoding Match-Action Tables

In real match-action tables, match patterns and actions are paired in rules. A single rule can be easily encoded in MatchKAT as the composition of a test with actions. Less straightforward is capturing the rule selection mechanism of the table. For example, let $b_1 \dots b_k$ be match expressions and $p_1 \dots p_k$ actions. In a table with rules $(b_1 p_1) \dots (b_k p_k)$, we may have multiple b expressions matching an incoming packet. In a *priority*-ordered table, the rule that is actually selected and has its action executed is based on some pre-assigned priority ordering on the rules. Here suppose 1 is the highest priority and k the lowest. A naive MatchKAT encoding of the table as $b_1 p_1 + \dots + b_k p_k$ does not work, since in a KAT $+$ is commutative. To impose an order, the simplest way is to negate all higher-priority tests:

$$b_1 p_1 + \overline{b_1} b_2 p_2 + \dots + \overline{b_1} \dots \overline{b_{k-1}} b_k p_k.$$

This term contains $O(k)$ sums and $O(k^2)$ compositions. Albeit inefficient, in this case indeed a rule's action will only be executed if no higher-priority rule matched.

An alternative encoding is to set aside some *metadata* bits as a counter to record the current rule being matched. Suppose this counter resides in bits i through j, then using incrementation from the previous section, we can write:

$$([i \dots j] \leftarrow 1) \left[\sum_{r=1}^{k} \left([i \dots j] = r \cdot \left(b_r p_r \cdot [i \dots j] \leftarrow (k+1) + \overline{b_r} \, [i \dots j]^{++} \right) \right) \right]^{\star}.$$

Here we write $[i \ldots j] = r$ as shorthand for testing the range bitwise for the binary number r. The encoded term works by only testing rule r if $[i \ldots j]$ has value r. If b_r succeeds then action p_r is executed, and the rule counter is set to the end value $k + 1$. If b_r fails then the rule counter is incremented. Kleene star is used to iterate through all the rules.

The above examples are not the only possible ways to encode match-action tables in MatchKAT. However, since we will prove that the equational theory of MatchKAT is sound and complete with respect to its packet filtering semantics, in principle we should be able to prove equivalence between all possible valid encodings. Even though different encodings have equivalent semantics, they may have different implementation qualities such as the length of match expressions, the depth of nesting, and the use of additional bits to store metadata such as $[i \ldots j]$ in the example above. Nevertheless we can establish a notion of program equivalence between these two ways of representing a table of match-action rules.

In some network switches there exist more than one match-action table organized in a *pipeline* [16, 47]. The tables can be sequentially composed, or possibly be in parallel with branching and loops. These can all be handled by MatchKAT's $\cdot$ for sequencing, $+$ for parallelism or branching, and * for loops.

3.3 Connection with NetKAT

NetKAT is an algebraic language based on Kleene algebra with tests that is able to specify packet forwarding policies in a network [9]. Before we study the equational theory of MatchKAT, we

will precisely define a connection between MatchKAT and NetKAT in both a syntactic and also semantic sense. This will allow us to leverage known results about NetKAT in the MatchKAT setting. Syntactically, there is a correspondence between MatchKAT and the dup-free fragment of NetKAT, and we will elaborate on this shortly. Semantically, NetKAT is mainly concerned with the possible progressions of a packet through the network, whereas we are more interested in the behavior of a single, local switch on packets. The syntactic and semantic relationships are entirely consistent. The dup operation can be used in NetKAT to record the states of a packet at different hops, so it is natural that without dup, we instead reason about what happens on the local hop. This is referred to in [60] as the "local program", where the switch configuration is still in NetKAT but agnostic about the network topology. However, we emphasize that MatchKAT is not intended to serve the same purpose as NetKAT. The language instead focuses on lower-level match expressions and manipulation of bits as this is closer to what is implemented in hardware.

We give a short description of NetKAT's syntax and its axioms, but since NetKAT is well-presented elsewhere, we will not discuss too many details here. However, what we will see by the end of this section are mutual translations between MatchKAT and NetKAT that will come in useful when we study MatchKAT's equational theory.

3.3.1 Syntax and Axioms of NetKAT

Let $F = \{f_1, f_2, \dots, f_n\}$ be some fixed, finite set of *fields*. NetKAT is a KAT again with signature $(P, B, +, \cdot, ^\star, 0, 1, ^-)$ whose primitive tests and actions are defined with respect to F:

- In addition to 0 and 1, primitive tests are of the form $f_i = k$, for some natural number k

and $f_i \in F$.

- There is a special primitive action named dup. Other primitive actions are in assignment form $f_i \leftarrow k$.

We assume for each field f_i there exists a finite set of natural numbers that could be associated with the field. Hence a NetKAT term is not well-formed if it contains $f_i = k$ or $f_i \leftarrow k$ for k not in that set. Just like MatchKAT, in addition to the standard KAT axioms, NetKAT requires packet algebra axioms governing mainly when tests and actions can commute [9]:

$$\begin{aligned}
f_1 \leftarrow i \,;\, f_2 \leftarrow j &\equiv f_2 \leftarrow j \,;\, f_1 \leftarrow i \quad (\text{if } f_1 \neq f_2) \\
f_1 \leftarrow i \,;\, f_2 = j &\equiv f_2 = j \,;\, f_1 \leftarrow i \quad (\text{if } f_1 \neq f_2) \\
f = i \,;\, \texttt{dup} &\equiv \texttt{dup} \,;\, f = i \\
f \leftarrow i \,;\, f = i &\equiv f \leftarrow i \\
f = i \,;\, f \leftarrow i &\equiv f = i \\
f \leftarrow i \,;\, f \leftarrow j &\equiv f \leftarrow j \\
f = i \,;\, f = j &\equiv \mathbf{0} \quad (\text{if } i \neq j) \\
\left(\textstyle\sum_i f = i\right) &\equiv \mathbf{1}.
\end{aligned}$$

The terms **0** and **1** are also commonly referred to as `drop` and `skip` respectively in presentations of NetKAT. We highlight the fact that tests and actions in NetKAT involve constant values. At first glance this may appear more limited than MatchKAT's ability to perform indirect assignment and computation on fields as demonstrated previously. We point out that this is only possible in MatchKAT's case since the size n of the state space is known and we are performing fixed-width arithmetic. Although we will see later that there is a close connection between

the two, this difference in focus between NetKAT and MatchKAT means they are still separate languages dealing with different levels of abstraction of network programs.

3.3.2 Semantics

We will talk briefly about the semantics of NetKAT, while readers interested in a formal detailed treatment are invited to read [9]. In NetKAT, a *packet* is a record of field-value pairs $\{f_1 = k_1, \ldots, f_n = k_n\}$ where each field has a valid assignment of values. This represents the header of a real-life packet that is of interest when we are deciding on its forwarding behavior. A *packet history* is simply a list of packets with the head being the most recent.

Definition 30. Let H be the set of packet histories. For $\pi \in Pk$, we write $\pi :: \langle\rangle$ for the packet history with π at its head and nothing else, and hd to be the function that takes packet history to their head packets. When we conflate notation and write $hd\, H$ for $H \subseteq \mathsf{H}$, we mean the set $\{hd\, h \mid h \in H\}$.

In NetKAT's packet filtering semantics, the interpretation of a term e is a function $[\![e]\!] : \mathsf{H} \to \mathcal{P}(\mathsf{H})$. Composition of these functions is done through Kleisli composition in the powerset monad. The semantics can be thought of as the behavior of a switch when it is presented with the head packet in a packet history. Each packet in the history represents a previous state of the head packet, possibly at a previous switch in the network. Using packet histories, as opposed to simply packets, allows us to distinguish packets that have taken different paths in the network. However, the input history beyond the head packet cannot be accessed directly by NetKAT terms, consistent with a switch not being able to see the operations that previous switches have done

to the packet.

The semantics of NetKAT can be explained intuitively. The test 1 lets a packet through unchanged, while 0 drops the packet. Programs $f = k$ and $f \leftarrow k$ tests and assigns the field f with the value k respectively, in the head packet of the input packet history. The program dup duplicates the current head packet and places a copy of it at the head of the history, i.e.

$$[\![\mathsf{dup}]\!]\,(\pi :: h) \triangleq \{\pi :: \pi :: h\}\,.$$

Note also that the codomain of the semantic function is sets of packet histories. This accommodates the fact that it is possible for a switch to egress multiple packets in response to a packet at ingress, possibly different in content and to different destinations. Composition $\cdot$ of interpretations having type $\mathsf{H} \to \mathcal{P}\,(\mathsf{H})$ is done through Kleisli composition in the powerset monad, in contrast to function composition in MatchKAT. The operators $+$ and $^-$ become union and complementation in the result sets respectively, and * takes the usual meaning of iterated composition.

Example 31. Suppose the set of fields is $\{\mathsf{pt}, \mathsf{proto}, \mathsf{ttl}\}$ and pt is understood to be the switch port where the packet is located. The NetKAT term $\mathsf{pt} = 1 \cdot \mathsf{proto} = 6 \cdot \mathsf{dup} \cdot \mathsf{ttl} \leftarrow 40 \cdot \mathsf{pt} \leftarrow 3$ is the policy "If the packet is at port 1 and has proto value 6, take a snapshot of its current state, change the ttl value to 40 and move the packet to port 3. Otherwise drop the packet."

3.3.3 MatchKAT to NetKAT

We will now formally define a translation from MatchKAT to NetKAT. For a MatchKAT with packet size n, the corresponding NetKAT will be over n fields, f_1 through f_n, each taking 0 or 1

in value. We define a homomorphism $\lceil \cdot \rceil$ that takes terms in this MatchKAT to the corresponding NetKAT terms as follows:

$$\lceil \bot \rceil \triangleq 0 \qquad \lceil \top \rceil \triangleq 1 \qquad \lceil i \leftarrow k \rceil \triangleq f_i \leftarrow k$$

The definitions for $+$, $\cdot$, * and $^-$ terms extend homomorphically, i.e.

$$\lceil e + e' \rceil \triangleq \lceil e \rceil + \lceil e' \rceil \qquad \lceil e \cdot e' \rceil \triangleq \lceil e \rceil \cdot \lceil e' \rceil \qquad \lceil e^\star \rceil \triangleq \lceil e \rceil^\star \qquad \lceil \overline{e} \rceil \triangleq \overline{\lceil e \rceil} \ .$$

We complete the definition for primitive tests by giving the translation in terms of match expressions on single bits and then concatenation. Translations of more complex match expressions extend naturally from the definitions for $+$ and $\cdot$.

$$\lceil 0 \rceil \triangleq f_i = 0 \qquad \lceil 1 \rceil \triangleq f_i = 1 \qquad \lceil \mathrm{x} \rceil \triangleq 1 \qquad \lceil e @ e' \rceil \triangleq \lceil e \rceil \cdot \lceil e' \rceil$$

Here i refers to the bit position that the single-bit expression 0 or 1 is matching. We can precompute these position values for every 0 or 1 that appears in the expression before carrying out the translation. Notice that the translation does not introduce any dups, and is a straightforward syntactic embedding into NetKAT. More importantly, this translation is semantic preserving in the following way.

Theorem 32. *For any MatchKAT term e, $[\![e]\!](P) = \bigcup_{\pi \in P} hd\,[[\![\lceil e \rceil]\!](\pi :: \langle\rangle)]$.*

The proof is a standard induction on e. We will simply observe that since the translation introduces no dups, $hd\,(\pi :: \langle\rangle) = \pi$, and it is clear that $\lceil e \rceil$ performs the same operations in NetKAT as e does in MatchKAT.

3.3.4 NetKAT to MatchKAT

Similarly, there is a translation from NetKAT to MatchKAT. Since the latter is dup-free, such a translation is forgetful in the sense that we lose the packet history structure entirely and only track the state of the head packet.

Suppose the particular NetKAT language we wish to translate from has fields f_1 through f_m. We assume it is possible to represent the values in each field in binary, and let $|f_i|$ denote the number of bits required to store f_i. We set the target MatchKAT packet size to be $n = \sum_i |f_i|$. The translation from NetKAT terms to MatchKAT terms is again a homomorphic function $\lfloor \cdot \rfloor$, and it is only necessary for us to specify its action on the on primitives:

$$\lfloor 0 \rfloor \triangleq \bot \qquad \lfloor 1 \rfloor \triangleq \top \qquad \lfloor \mathsf{dup} \rfloor \triangleq \top$$

$$\lfloor f_i = k \rfloor \triangleq \bigsqcap_{j=1}^{|f_i|} pos_j(f_i) \simeq bin_j(k) \qquad \lfloor f_i \leftarrow k \rfloor \triangleq \prod_{j=1}^{|f_i|} pos_j(f_i) \leftarrow bin_j(k)$$

The function $pos_j(f_i)$ gives bit position for the j-th bit in the header space allocated for f_i, i.e. it is $pos_j(f_i) = j + \sum_{i'<i} |f_{i'}|$.

On the other hand, $bin_j(k)$ is the j-th bit of the binary representation of k. The translation for assignment $f_i \leftarrow k$ just sets each bit in the space allocated for f_i in the target MatchKAT bitwise. This is the same method for test $f_i = k$, with the resulting bitwise tests composed by $\sqcap$, and we can always equivalently combine the tests into a match expression without $\sqcap$ by using match expression axioms like in Example 26. We forget the existence of dup by translating it as $\top$. Just like the translation to NetKAT, $\lfloor \cdot \rfloor$ implies a semantic correspondence.

Theorem 33. *For any NetKAT term e, $hd\,[[\![e]\!](h)] = [\![\lfloor e \rfloor]\!](\{hd\,h\})$.*

Again the proof proceeds by induction on e, but we will elaborate slightly this time. The base cases are all straightforward by the following reasoning. Both 0 and $\bot$ filter out all packet (histories), while 1 and $\top$ let through everything. Assignments and tests in both worlds perform the same operations on the (head) packet. The program dup does not change the head packet, and on both sides we only consider the head packets. The inductive cases then rely on hd commuting with the semantics of the NetKAT operators $+$, $\cdot$ and *, which it does since NetKAT terms do not examine or modify packets in the packet history beyond the head.

3.4 Equational Theories of MatchKAT and dup-Free NetKAT

As promised, we show that the equational theory of MatchKAT is sound and complete with respect to the packet filtering semantics, through borrowing soundness and completeness results of NetKAT's equational theory from [9].

Consider two NetKAT terms e and e'. Suppose

$$\forall h \in \mathsf{H}.\, hd\left[\llbracket e \rrbracket\,(h)\right] = hd\left[\llbracket e' \rrbracket\,(h)\right].$$

It is not necessarily the case that $\llbracket e \rrbracket = \llbracket e' \rrbracket$. Although NetKAT terms cannot access packets beyond the head in the input packet history, $\llbracket e \rrbracket$ may still produce different output packet histories compared to $\llbracket e' \rrbracket$ by using dup. If e and e' are dup-free however, we are then able to deduce $\llbracket e \rrbracket = \llbracket e' \rrbracket$. The equational theory of dup-free NetKAT is therefore determined entirely by the operations on the head packet. This idea can be developed into a proof for the soundness and completeness for the equational theory of MatchKAT. First we require two lemmas.

Lemma 34. *For any MatchKAT expression* e, $[\![e]\!] = [\![\lfloor \lceil e \rceil \rfloor]\!]$.

Proof. For all h, we have

$$hd\,[[\![\lceil e \rceil]\!]\,(h)] = [\![\lfloor \lceil e \rceil \rfloor]\!]\,(\{hd\,h\})$$

by Theorem 33. Hence for all $H \subseteq \mathsf{H}$,

$$\bigcup_{h \in H} hd\,[[\![\lceil e \rceil]\!]\,(h)] = [\![\lfloor \lceil e \rceil \rfloor]\!]\,(hd\,H)\,.$$

On the other hand, Theorem 32 gives us

$$[\![e]\!]\,(hd\,H) = \bigcup_{\pi \in hd\,H} hd\,[[\![\lceil e \rceil]\!]\,(\pi :: \langle\rangle)]$$

for all $H \subseteq \mathsf{H}$. Since $h \approx (hd\,h) :: \langle\rangle$ for all $h \in H$, it is safe to rewrite the latter equation to

$$[\![e]\!]\,(hd\,H) = \bigcup_{h \in H} hd\,[[\![\lceil e \rceil]\!]\,(h)]\,.$$

Combining this with the second equation gives the required result. □

Lemma 35. *For all MatchKAT expressions* e *and* e', $e \equiv e' \iff \lceil e \rceil \equiv \lceil e' \rceil$.

Proof. On the left of the bi-implication we have MatchKAT terms operating on n bits. On the right are NetKAT terms operating on n fields, each containing 1 bit. Through exhaustion we can prove that every axiom in the MatchKAT world gives rise to a corresponding axiom (or derivable theorem) in the NetKAT world, and vice versa, and hence a proof of equality in one produces automatically a proof of equality in the other. Instead of going through the full proof for every axiom, we give some reasons for why it works.

- KAT axioms are clearly present in both worlds, and $\lceil \cdot \rceil$ is a homomorphism.

- The packet algebra axioms are present in both as mentioned in Section 3.2.1.
- The axioms for manipulating match expressions are present in MatchKAT, but they are not in NetKAT. However NetKAT has extra axioms of the forms $i = k \cdot i = k \equiv i = k$, $k \neq k' \implies i = k \cdot i = k' \equiv 0$, and $\sum_k (i = k) \equiv 1$. These, along with the axioms of the Boolean algebra, are sufficient to derive equivalents of match expression axioms as theorems.

□

Theorem 36. *(Soundness and completeness.) For all MatchKAT expressions* e *and* e'*,* $e \equiv e' \iff [\![e]\!] = [\![e']\!]$.

This follows from the implications

$$\begin{aligned}
e \equiv e' &\iff \lceil e \rceil \equiv \lceil e' \rceil && \text{(Lemma 35)} \\
&\iff [\![\lceil e \rceil]\!] = [\![\lceil e' \rceil]\!] && \text{(NetKAT sound \& completeness)} \\
&\iff [\![\lfloor \lceil e \rceil \rfloor]\!] = [\![\lfloor \lceil e' \rceil \rfloor]\!] && \text{(Theorem 33)} \\
&\iff [\![e]\!] = [\![e']\!] && \text{(Lemma 34).}
\end{aligned}$$

This third step follows from Theorem 33 since $\lceil e \rceil$ and $\lceil e' \rceil$, being translations from MatchKAT and therefore dup-free, have interpretations determined entirely by modifications on the head packet.

3.4.1 Complexity of Deciding Equivalence

In this section, we discuss the complexity of deciding equivalence in MatchKAT, and how the result relates to NetKAT.

Theorem 37. *Deciding equivalence in MatchKAT is* PSPACE*-complete*

Membership of PSPACE is argued by translating the MatchKAT terms to the dup-free fragment of NetKAT as shown previously. The equational theory of this fragment is in PSPACE since that of NetKAT is in PSPACE [9]. There is no superpolynomial blowup in the translation as can be seen from the homomorphic definition of $\lceil - \rceil$.

For hardness, we can encode a word problem for a linear-bounded automaton as a MatchKAT term e, such that the automaton accepts the given word if and only if $e \not\equiv \bot$.

Proof. A linear-bounded automaton $M = (Q, \Sigma, \vdash, \dashv, \delta, s, t, r)$ is composed of:

- Finite set of states Q.
- Tape alphabet Σ.
- Left $\vdash$ and right $\dashv$ tape-end markers.
- Transition relation $\delta \subseteq [Q \times (\Sigma \cup \{\vdash, \dashv\})] \times [Q \times (\Sigma \cup \{\vdash, \dashv\}) \times \{L, R\}]$.
- Start s, accept t, and reject r states.

M can be seen as a non-deterministic Turing machine where the tape is finite and marked on both ends by $\vdash$ and $\dashv$. At the start, an input word is present on the tape, while the tape is bounded

to a linear size n with respect to the length of the input. The relation δ is restricted such that the end markers are unmodified and the tape head does not move off the ends of the tape. The automaton never transitions out of the accept or reject states once it enters them. We will further restrict Σ to two symbols $\{0, 1\}$. This is without loss of generality with a linear increase in the amount of tape required.

A packet in the MatchKAT encoding of M contains the following bits:

- n bits that we will refer to for convenience as $\mathsf{tape}_1, \ldots, \mathsf{tape}_n$, representing the tape.
- Bits $\mathsf{state}_1, \ldots, \mathsf{state}_{\log|Q|}$ to record the binary encoding of the current state.
- Bits $\mathsf{head}_1, \ldots, \mathsf{head}_{\log n}$ to record the binary encoding of the head position.

Instead of referring to each state and head bit individually, we will assign and test for them collectively for a particular state of Q or tape head position. We then construct expressions in the MatchKAT as follows:

- The *setup expression* α, which is an assignment of state with s, head with 1, and the tape fields as appropriate for the initial tape contents for a given input word.
- The *transition expression* β, consisting of sums guarded by state $\times$ head $\times$ $\mathsf{tape}_{\mathsf{head}}$ conditions. For a packet in a given configuration, it rewrites it according to the action of the transition relation.
- The *decision expression* γ, which is just a test for state $= t$.

Consider the expression $\alpha\,(\beta^\star)\,\gamma$, which is not equivalent to $\bot$ if and only M accepts the given word. For any non-empty set of input packets, $\alpha\,(\beta^\star)$ constructs the set of all reachable configurations of M, while γ filters this set to include only the packets that contain the accept state. On the other hand, $\bot$ drops all packets. The size of the expressions are polynomial in the size of the automaton specification. □

The word problem for a linear-bounded automaton is known to be PSPACE-hard [30]. The hardness result in [9] of deciding equivalence in NetKAT relies on a simple translation of regular expressions to NetKAT expressions containing many dups. Our result improves this slightly:

Corollary 38. *Deciding equivalence of* dup*-free NetKAT terms is* PSPACE*-complete.*

This can be seen through a similar encoding of the linear-bounded automaton.

3.5 Discussion and Conclusion

We will end by discussing the potential applications and decision procedures of MatchKAT, as the latter will be crucial in any real-world application in reasoning with match-action tables. Efficient procedures for NetKAT have already been discovered, such as in [28, 60], that work well on many real-life cases. Much of the difficult work is in reasoning with dup, which MatchKAT does without. We conjecture that it should be possible to adapt these previous decision procedures to MatchKAT with much simplification. A coalgebraic treatment of MatchKAT directly is also conjectured to be possible.

Application-wise, it is envisaged that MatchKAT could be used to reason about local switch behavior, in contrast to NetKAT on global network policies, when the switch has already been configured by match-action rules. This could be useful for various reasons:

- MatchKAT has a sound and complete equational theory. Equivalence of terms can be decided and is guaranteed to be sound. This helps in the verification of correctness as well as potential configuration optimizations in reducing the number of rules. We have previously talked about the notion of length for MatchKAT terms, and so equivalence of terms of different lengths is potentially proof of equivalence between optimized and unoptimized configurations.
- MatchKAT is equivalent to dup-free NetKAT, and there is a well-defined translation between the two. This could help in decompiling match-action tables to NetKAT in order to make sense of the global policies they are implementing.
- MatchKAT's match expressions are closer to how bits in packet headers are matched on switches at low level. MatchKAT could potentially help with efficient implementations of hardware that performs matching.

These all distinguish our work from previous attempts such as [36, 37] that also reason with binary data in packet headers theoretically. We also note with interest that other authors have also created new algebraic systems with a strong relationship to NetKAT, such as [31]. Further, [42] gives a uniform approach for showing the completeness of deductive systems for Kleene algebra with additional equations, although our work specifically applies to the case of match-action. In the future, we intend to further develop concrete applications of MatchKAT in the setting of match-action tables, and demonstrate the usefulness of its algebraic theory.

CHAPTER 4

STATEFUL NETKAT

NetKAT, as presented by [9], is stateless. One can express and reason about network topology and switch forwarding policies, but the switches themselves are assumed to be stateless and must be reconfigured by the controller to change their behavior. However, some packet forwarding devices support local state that can be configured to change at line speed in response to packet-processing activity without intervention from the controller. A simple example of how such an extension might be useful is in *load balancing*. A switch might be configured to spread traffic equally among three paths in a round-robin fashion. This could be accomplished with a three-state automaton in a switch A that changes state with each packet and sends the packet out on a different port depending on the state as follows:

if $switch = A$ **then**
  if $A.state = 0$ **then** $port \leftarrow 100$
  else if $A.state = 1$ **then** $port \leftarrow 101$
  else if $A.state = 2$ **then** $port \leftarrow 102$
  else drop
  $A.state \leftarrow (A.state + 1)$ **mod** 3

Other applications are Network Address Translation, firewalls, and rate limiters that reconfigure based on state transitions triggered by arriving packets.

In this chapter, we describe Stateful NetKAT, an extension of NetKAT to handle mutable state. This extension requires us to revisit the basic semantics of NetKAT. In the stateless case, the

standard NetKAT packet filter model interprets each expression as a map $H \to 2^H$, where H is the set of packet histories. In this model, the order and multiplicity of packets are less relevant, as these aspects are inconsequential for many common applications such as reachability, loop-freedom, and waypointing. However, in the presence of local state, order and multiplicity matter, because tests and actions performed in a switch depend on both packet information and local state. Processing packets in a different order can result in different state transition behavior. To account for this, our language must be able to express any arbitrary ordering of packet processing the programmer specifies.

We thus consider an alternative model in which the order and multiplicity of packets are retained. Whereas in NetKAT, the powerset operator is used to accumulate all packets produced in the network, here we use $H^\star$ for that purpose.

We begin by reinterpreting purely stateless programs as maps of type $H \to H^\star + \bot$, where $H^\star$ is the set of finite sequences of packet histories. The element $\bot$ is a new element representing a nonterminating computation. Algebraically, the structure $H^\star + \bot$ can be described as the *free monoid with zero on generators* H, where $\bot$ acts as a zero element satisfying $\bot x = x \bot = \bot$.

A purely state-manipulating program, one that tests and manipulates the local state without reference to a packet, can be interpreted as a partial map $S \rightharpoonup S$, where S is the set of (global) states, consisting of the aggregated values of all local state variables $A.x$.

In general, Stateful NetKAT programs can refer to both packets and state. These are interpreted as maps of type $S \times H^? \to S \times H^\star + \bot$, where S is the set of states and $H^? = H + \varepsilon$ is an optional input packet. Although Stateful NetKAT, inheriting from NetKAT, appears superficially

to have an imperative style, the language is presented using mainly functional ideas. Besides employing a monoidal structure to capture order and multiplicity, state is specified monadically, and sequencing of program terms is defined with Kleisli composition.

Beside works on NetKAT [9, 28, 60] and the previous chapter, the reader is invited to consult the background on Kleene algebras (KA) [39] and Kleene algebras with tests (KAT) [41].

4.1 Contributions

Our aim is to introduce general purpose mutable state to NetKAT that can be thought to persist and evolve as sequences of packets are processed. In particular, we wish to achieve this goal while extending NetKAT as conservatively as possible, while remaining general with regards to the types of packets and state that can be combined. We make the following contributions in our presentation of Stateful NetKAT:

- The syntax and semantics of Stateful NetKAT are given in Section 4.2, at first by presenting the stateless and pure state parts of the language separately. Drawing inspiration from [32], the combined language is the commutative coproduct of the two halves. In [32], any Kleene algebra with tests and any finite relation algebra can be combined in a free way to give a language that can simultaneously reason about both algebras. Stateful NetKAT's claim is similarly any instance of NetKAT can be combined with an algebra of mutable states with the fewest restrictions on both.
- Besides giving packet filtering semantics, as is traditional for languages deriving from NetKAT, we give a language model for Stateful NetKAT in Section 4.4. This captures the

input-output relation of programs as sets of regular strings.

- The language model then allows us to give a decision procedure for the equational theory of a large fragment of Stateful NetKAT in Section 4.5. We further consider the start of a possible coalgebraic approach and axiomatization.

4.2 Syntax and Semantics

4.2.1 Syntax

Unlike ordinary NetKAT, the ordering and multiplicity of ingressed packets can affect the execution and possible outputs of a packet forwarding program that manipulates the local state. We introduce a new Stateful NetKAT syntax that takes this into account. Like ordinary NetKAT, Stateful NetKAT is compositional. Terms in Stateful NetKAT are composed from *atomic programs* joined with *program operators*. For the syntax, firstly we define

$$\text{Variables } u, v ::= x \mid A.x.$$

The variables are a finite set and they are partitioned into *packet fields* x and *state variables* $A.x$. The packet fields are the same as in ordinary NetKAT, so operations on these correspond to operations a switch may take on the packet header. On the other hand, each state variable is associated with some particular switch A, and is so differentiated from a packet field by writing $A.x$.

There are two types of atomic programs, *actions* and *tests*:

$$\begin{aligned} \text{Atomic actions } a &::= \texttt{dup} \mid u \leftarrow n \mid u \leftarrow v \\ \text{Atomic tests } t &::= u = n \mid u = v. \end{aligned}$$

Here n is an element of a fixed finite set of values. Thus tests are between a variable and a constant n or between two variables of the same sort, and assignments change the value of a variable to a constant or to the value of another variable. The language allows assignments and tests on both packet fields and state variables. As in NetKAT, the action dup extends the packet history with a snapshot of the current packet.

Compound programs are formed as follows.

$$\begin{aligned} \text{Tests } b, c &::= t \mid \overline{b} \mid b \vee c \mid b\,;c \\ \text{Programs } p, q &::= a \mid b \mid p\,;q \mid p \,\&\, q \mid \texttt{if } b \texttt{ then } p \texttt{ else } q \mid \texttt{while } b \texttt{ do } p. \end{aligned}$$

The expressions $p\,;q$ and $p \;\&\; q$ denote sequential and parallel composition, respectively. The intuitive difference between the two is that the sequential composition $p;q$ applies p to the input and then q on its result, whereas the parallel composition $p \;\&\; q$ duplicates the input and applies p to one copy, then q to the other. Following [61], we abbreviate the conditional program $\texttt{if } b \texttt{ then } p \texttt{ else } q$ by $p +_b q$ and the while loop $\texttt{while } b \texttt{ do } p$ by $p^{(b)}$. We also write $x \neq a$ for the negation of the test $x = a$. Finally, we define the following four terms as syntactic sugar, for some arbitrary packet field x and state variable $A.x$:

$$\texttt{skip} \triangleq x = x \qquad \texttt{drop} \triangleq x \neq x \qquad \texttt{true} \triangleq A.x = A.x \qquad \texttt{false} \triangleq A.x \neq A.x$$

The test skip tests whether a packet is indeed present. On the other hand, the test true always succeeds and false always fails.

4.2.2 Semantics

We now consider the semantics of Stateful NetKAT. Let H denote the set of packet histories with elements h, and $H^? \triangleq H + \varepsilon$ the set whose elements are either one of H or none ε. Let $H^\star$ be the set of finite strings of packet histories, including ε representing the empty string. Although we always work with packet histories, for conciseness we will refer to them as "packets" when we are only interested in the head packet. We write $\bot$ for an element representing a nonterminating computation.

The semantics of Stateful NetKAT is presented in three stages:

1. We first give a revised semantics of stateless NetKAT in which programs are interpreted as maps of type $H \to H^\star + \bot$. This lifts naturally to type $H^\star + \bot \to H^\star + \bot$.
2. An algebra is then developed for terms that purely manipulates state. The elements of this algebra are partial functions $S \multimap S$ where S is the set of global states.
3. Finally, we show how the above can be combined into a single language, leading to a semantics in which programs are interpreted as maps $S \times H^? \to S \times H^\star + \bot$. This is done by taking the commutative coproduct similar to the method developed in [32].

The multiplication operation in the monoid with zero $H^\star + \bot$ is denoted $x \cdot y$ or xy. The result is $\bot$ if either x or y is $\bot$, otherwise string concatenation is performed.

4.2.2.1 Stateless Semantics

A stateless NetKAT program p is interpreted as a function $[\![p]\!] : H \to H^\star + \bot$. All programs have a unique homomorphic extension $[\![p]\!]^\dagger : H^\star + \bot \to H^\star + \bot$, as $H^\star + \bot$ is the free monoid with zero on generators H. Formally, for $x \in H^\star$, $h \in H$, and ε the empty sequence, define inductively

$$[\![p]\!]^\dagger(\varepsilon) \triangleq \varepsilon \qquad [\![p]\!]^\dagger(hx) \triangleq [\![p]\!](h) \cdot [\![p]\!]^\dagger(x) \qquad [\![p]\!]^\dagger(\bot) \triangleq \bot.$$

Intuitively, $[\![p]\!]^\dagger(x)$ processes the packets in the input string x sequentially and concatenates the results in the same order. It follows inductively that for all $x, y \in H^\star + \bot$,

$$[\![p]\!]^\dagger(xy) = [\![p]\!]^\dagger(x) \cdot [\![p]\!]^\dagger(y). \tag{4.1}$$

For the stateless semantics, the variables are restricted to packet fields. For atomic programs,

$$[\![x \leftarrow n]\!](h) \triangleq h[n/x] \qquad [\![x \leftarrow y]\!](h) \triangleq h[h(y)/x] \qquad [\![\mathtt{dup}]\!](\pi : h) \triangleq \pi : \pi : h$$

$$[\![x = n]\!](h) \triangleq \begin{cases} h, & h(x) = n \\ \varepsilon, & h(x) \neq n \end{cases} \qquad [\![x = y]\!](h) \triangleq \begin{cases} h, & h(x) = h(y) \\ \varepsilon, & h(x) \neq h(y). \end{cases}$$

Here $h[n/x]$ denotes the history h with the field x rebound to the value n in the head packet of h, and $h(x)$ denotes the value of the field x in the head packet of h. The dup operator prepends a copy of the head packet to the history. Note the following key difference with ordinary NetKAT: a dropped packet is represented by the empty sequence ε, not by $\emptyset$.

Compound programs except for $p^{(b)}$ are interpreted as follows.

$$\begin{aligned}
[\![p\,;q]\!]\,(h) &\triangleq [\![q]\!]^{\dagger}\,([\![p]\!]\,(h)) \qquad & [\![p +_b q]\!]\,(h) &\triangleq \begin{cases} [\![p]\!]\,(h)\,, & h \vDash b \\ [\![q]\!]\,(h)\,, & h \vDash \overline{b} \end{cases} \\
[\![\overline{b}]\!] &\triangleq [\![\mathtt{drop} +_b \mathtt{skip}]\!] & & \\
[\![b \vee c]\!] &\triangleq [\![\mathtt{skip} +_b c]\!] \qquad & [\![p \,\&\, q]\!]\,(h) &\triangleq [\![p]\!]\,(h) \cdot [\![q]\!]\,(h)\,.
\end{aligned}$$

Sequential composition $[\![p\,;q]\!]$ is just the Kleisli composition of $[\![p]\!]$ and $[\![q]\!]$ in the monad $-^{\star}+\bot$, i.e. the pointed list monad with point $\bot$. It expresses the operation of performing p on the input packet, then q on each packet in the resulting string from left to right.

Intuitively, for a (compound) test b, $[\![b]\!]\,(h)$ passes the packet history through unchanged if the test succeeds and drops it if not. The notation $h \vDash b$ indicates that b succeeds on the head packet of h. From this we can define the conditional $+_b$, and the semantics of a negated or disjunctive test used as a primitive program follow naturally.

For parallel composition $p \,\&\, q$, p and q are performed on the input packet h in parallel and the results are concatenated. Note that this operation is not commutative: all packets produced by p precede all packets produced by q. This allows us to maintain a global chronological order on packets. The sequential composition operator $p\,;q$ first performs p to get a sequence of packets, then performs q on that sequence. The definition of $[\![\overline{b}]\!]$ is implied by that of $[\![b]\!]$. Sequential composition of tests b and c, which forms a test, is consistent with the definition of $[\![b\,;c]\!]\,(h)$ such that h is let through if and only if the conjunction of b and c succeeds on h. Other logical operations such as disjunction can be achieved through negation.

The semantics $[\![p^{(b)}]\!]$ will be a certain solution of

$$[\![p^{(b)}]\!] \quad = \quad [\![p\,;p^{(b)} +_b \texttt{skip}]\!] \tag{4.2}$$

or in more conventional notation,

$$\begin{aligned}
[\![\texttt{while}\ b\ \texttt{do}\ p]\!] &= [\![\texttt{if}\ b\ \texttt{then}\ (p\,;\texttt{while}\ b\ \texttt{do}\ p)\ \texttt{else}\ \texttt{skip}]\!] \\
&= \lambda h.\begin{cases} [\![\texttt{while}\ b\ \texttt{do}\ p]\!]^{\dagger}([\![p]\!](h)), & h \vDash b \\ h, & h \vDash \overline{b}. \end{cases}
\end{aligned}$$

Let $\sqsubseteq$ be a partial order on $H^\star + \bot$ defined by: $x \sqsubseteq y$ if either $x = \bot$ or $x = y$. The structure $H^\star + \bot$ ordered by $\sqsubseteq$ is a chain- or directed-complete partial order (DCPO) with least element $\bot$. Multiplication in $H^\star + \bot$ is monotone with respect to this order. Suppose $x_1 \sqsubseteq y_1$ and $x_2 \sqsubseteq y_2$. If either x_1 or x_2 is $\bot$ then so is $x_1 x_2$, and so $x_1 x_2 \sqsubseteq y_1 y_2$. Otherwise, $x_1 x_2 = y_1 y_2$.

Definition 39. We can extend the ordering $\sqsubseteq$ to functions $H \to H^\star + \bot$ pointwise:

$$f \sqsubseteq g \overset{\Delta}{\Leftrightarrow} \forall h\ f(h) \sqsubseteq g(h).$$

The constant function giving $\bot$ is the $\sqsubseteq$-least element, which we will denote with ω.

We take the value of $[\![p^{(b)}]\!]$ to be the least solution of Equation 4.2 with respect to $\sqsubseteq$. Consider the map τ defined by

$$\tau(W) = \lambda h.\begin{cases} W^{\dagger}([\![p]\!](h)), & h \vDash b \\ h, & h \vDash \overline{b}. \end{cases}$$

Theorem 40. *$[\![p^{(b)}]\!]$ is the supremum of the chain $\omega \sqsubseteq \tau(\omega) \sqsubseteq \tau^2(\omega) \sqsubseteq \tau^3(\omega) \sqsubseteq \cdots$ in the function space $H \to H^\star + \bot$. It always exists and is well-defined.*

Proof. The existence of the least solution follows from the Knaster-Tarski theorem [64]. We must show that the map τ is monotone in the order $\sqsubseteq$. Suppose $W \sqsubseteq W'$. Then for all $h \in H$, $W(h) \sqsubseteq W'(h)$. It follows from monotonicity of multiplication in $H^\star + \bot$ that for all $x \in H^\star + \bot$, $W^\dagger(x) \sqsubseteq W'^\dagger(x)$. Then for all $h \in H$,

$$\begin{aligned}
\tau(W)(h) &= \begin{cases} W^\dagger(\llbracket p \rrbracket(h)), & h \vDash b \\ h, & h \vDash \overline{b} \end{cases} \\
&\sqsubseteq \begin{cases} W'^\dagger(\llbracket p \rrbracket(h)), & h \vDash b \\ h, & h \vDash \overline{b} \end{cases} \\
&= \tau(W')(h).
\end{aligned}$$

□

Theorem 41. $\llbracket p +_b q \rrbracket = \llbracket (b\,;p) \;\&\; (\overline{b}\,;q) \rrbracket$.

Proof. If $h \vDash b$ then $\llbracket b\,;p \rrbracket(h) = \llbracket p \rrbracket(h)$ and $\llbracket \overline{b}\,;q \rrbracket(h) = \llbracket \texttt{drop} \rrbracket(h)$. Otherwise $\llbracket b\,;p \rrbracket(h) = \llbracket \texttt{drop} \rrbracket(h)$ and $\llbracket \overline{b}\,;q \rrbracket(h) = \llbracket q \rrbracket(h)$. In either case the result follows. □

Using the above rule, expanding the definition of $\llbracket p^{(b)} \rrbracket$ shows it is the least solution of the equation

$$\llbracket p^{(b)} \rrbracket = \llbracket (b\,;p\,;p^{(b)}) \;\&\; \overline{b} \rrbracket$$

in the space of functions $H \to H^\star + \bot$, as lifted to $H^\star \to H^\star + \bot$ through 4.1 and Kleisli composition. Exactly one of $\llbracket b \rrbracket(h)$ and $\llbracket \overline{b} \rrbracket(h)$ is h and the other is ε, so

$$\llbracket \texttt{while}\ b\ \texttt{do}\ p \rrbracket(h) = \begin{cases} \llbracket p\,;\texttt{while}\ b\ \texttt{do}\ p \rrbracket(h), & h \vDash b \\ h, & h \vDash \overline{b}. \end{cases}$$

This shows $[\![\texttt{while}\ b\ \texttt{do}\ p]\!]$ matches what we expect operationally.

Example 42. Let $W \triangleq \texttt{while true do}\ x \leftarrow 1$. Its interpretation $[\![W]\!]$ is ω. This is because it is the least function satisfying $[\![W]\!] = [\![(x \leftarrow 1\,; W) +_{\texttt{true}} \texttt{skip}]\!]$. The result is the same if we replace $x \leftarrow 1$ in W with dup. Thus it is not possible for infinite packet histories to appear.

Example 43. Let $W \triangleq \texttt{while}\ x = 1\ \texttt{do}\ (\texttt{skip}\ \&\ x \leftarrow 2)$ and $\Omega \triangleq x \leftarrow 1\,; W$. For any packet h with $x = 1$, it must be the case that $[\![W]\!](h) = [\![W]\!](h) \cdot h\,[2/x]$. No packet sequence solution exists as they must be finite, but if we let $[\![W]\!](h) = \bot$ for every h with $x = 1$ then this is a solution. Hence $[\![\Omega]\!]$ must also be ω. In contrast to the previous example, where an *unproductive* computation does not terminate because it never emerges from the loop, here an infinite number of packets are produced. This is a case of *productive* nontermination.

4.2.2.2 Pure State

Let $A.x, A.y, A.z \ldots$ be a finite set of state variables, each with a finite domain of possible values. A (*global*) *state* is a valuation of state variables, that is, a map assigning a value to each state variable $A.x$. Let S be the set of all such valuations. We denote states by $s, s', \ldots \in S$. Although conventionally a state variable $A.x$ refers to the local variable x stored at switch A, the switch name A plays no part in the semantics besides being a distinguishing identifier. There is no restriction on which state variables can be accessed in a program, as they are all equally part of the global state.

A *pure state program* is one that does not refer to packets; that is, one whose atomic actions and tests are only of the form $A.x \leftarrow v$ and $A.x = v$, respectively, where v is a constant or

another state variable. In the absence of packets, the pure state language does not employ the $\&$ operator. Composition is solely expressed by $p\,;q$.

Pure state programs are interpreted as partial functions $f : S \rightharpoonup S$. The domain of f is denoted $\mathrm{dom}\, f$. In particular, pure state tests can be interpreted as subidentities, which are f such that $f(s) = s$ for all $s \in \mathrm{dom}\, f$. Thus,

$$\begin{aligned}
[\![A.x = n]\!](s) &\triangleq \begin{cases} s, & s(A.x) = n, \\ \text{undefined}, & s(A.x) \neq n, \end{cases} \\
[\![A.x = B.y]\!](s) &\triangleq \begin{cases} s, & s(A.x) = s(B.y), \\ \text{undefined}, & s(A.x) \neq s(B.y), \end{cases} \\
[\![A.x \leftarrow n]\!](s) &\triangleq s[n/A.x] \\
[\![A.x \leftarrow B.y]\!](s) &\triangleq s[s(B.y)/A.x].
\end{aligned}$$

The compositional operators are then interpreted homomorphically as compositions on partial

functions as follows:

$$\llbracket p\,;q \rrbracket(s) \triangleq \begin{cases} \llbracket q \rrbracket(\llbracket p \rrbracket(s)), & s \in \mathrm{dom}\llbracket p \rrbracket \wedge \llbracket p \rrbracket(s) \in \mathrm{dom}\llbracket q \rrbracket, \\ \text{undefined}, & \text{otherwise} \end{cases}$$

$$\llbracket p +_b q \rrbracket(s) \triangleq \begin{cases} \llbracket p \rrbracket(s), & s \vDash b, \\ \llbracket q \rrbracket(s), & s \vDash \overline{b}, \end{cases}$$

$$\llbracket \overline{b} \rrbracket(s) \triangleq \llbracket \mathtt{false} +_b \mathtt{true} \rrbracket$$

$$\llbracket b \vee c \rrbracket(s) \triangleq \llbracket \mathtt{true} +_b c \rrbracket$$

$$\llbracket p^{(b)} \rrbracket(s) \triangleq \begin{cases} \llbracket p \rrbracket^n(s), & n \in \mathbb{N} \text{ is the least number such that} \llbracket p \rrbracket^n(s) \vDash \overline{b}, \\ \text{undefined}, & \text{no such } n \text{ exists.} \end{cases}$$

There is a natural order on these interpretations given by

$$f \leq g \iff \forall s.\, s \notin \mathrm{dom}\, f \vee f\,(s) = g\,(s)\,. \tag{4.3}$$

A *complete assignment* is sequential list of assignments of constant values to every state variable. A *complete test* is a sequential test of every state variable against a constant. Complete assignments correspond to states in a one-to-one fashion, as do complete tests. We denote by $s!$ and $s?$ the complete assignment and complete test corresponding to state s, respectively. It is possible to completely axiomatize the pure state language along the lines of [9, 32], by treating each state as an atom of a finite Boolean algebra. This implies

$$s!t! = t! \qquad s! = s!s? \qquad s? = s?s! \qquad s?t? = \mathtt{false},\ s \neq t \qquad \sum_s s? = \mathtt{true}.$$

Alternatively, consider *functional matrices* that are square matrices over 0, 1 such that each row has at most one 1. Partial functions $f : S \rightharpoonup S$ and $S \times S$ functional matrices are in one-to-

one correspondence: f is represented by the matrix with 1 in location s, $f(s)$ for $s \in \mathrm{dom}\, f$ and 0 elsewhere. Subidentities are diagonal functional matrices. The algebra of pure state programs is characterized completely by $\mathrm{Mat}\,(S, 2)$, the KAT of $S \times S$ matrices over the binary KAT, in the following sense.

Theorem 44. *Every pure state program is equivalent to a sum of the form* $\sum_{s \in \mathrm{dom}\, f} s?f(s)!$ *for some partial function* $f : S \rightharpoonup S$*. For tests,* f *is a subidentity. Further, there exists a homomorphism into a subalgebra of* $\mathrm{Mat}\,(S, 2)$ *such that the program operators map directly to matrix operations.*

Proof. Considering each program operator in turn,

$$
\begin{aligned}
A.x \leftarrow n &\equiv \sum_s s?s[n/A.x]! \\
A.x \leftarrow B.y &\equiv \sum_s s?s[s(B.y)/A.x]! \\
A.x = n &\equiv \sum_{s(A.x)=n} s?s! \\
A.x = B.y &\equiv \sum_{s(A.x)=s(B.y)} s?s! \\
\left(\sum_{s \in \mathrm{dom}\, f} s?f(s)!\right) ; \left(\sum_{t \in \mathrm{dom}\, g} t?g(t)!\right) &\equiv \sum_{s \in \mathrm{dom}\, g \circ f} s?g(f(s))! \\
\left(\sum_{s \in \mathrm{dom}\, f} s?f(s)!\right) +_b \left(\sum_{t \in \mathrm{dom}\, g} t?g(t)!\right) &\equiv \sum_{s \in \mathrm{dom}\, f +_b g} s?(f +_b g)(s)! \\
\left(\sum_{s \in \mathrm{dom}\, f} s?f(s)!\right)^{(b)} &\equiv \sum_{s \in \mathrm{dom}\, f^{(b)}} s?f^{(b)}(s)!
\end{aligned}
$$

So every pure state program can be written in the required form. Turning to the matrix representations, for atomic programs they are uniquely defined. We let r denote the homomorphism into a subalgebra of $\mathrm{Mat}\,(S, 2)$. In particular $r\,(\texttt{false})$ is the zero matrix and $r\,(\texttt{true})$ is the identity.

For composition, let $r(f\,;g) \triangleq r(f) \cdot r(g)$ where $\cdot$ is matrix multiplication over the binary KAT. We have

$$\begin{aligned} r(f\,;g)_{st} = 1 \quad &\iff \quad r(f)_{sf(s)} = 1 \wedge r(g)_{f(s)t} = 1 \\ &\iff \quad [r(f) \cdot r(g)]_{st} = 1. \end{aligned}$$

For the conditional, let B be the diagonal matrix such that $B_{ss} = 1 \iff s \vDash b$, and similarly $\overline{B}$ for the test $\overline{b}$. Then let $r(f +_b g) \triangleq B \cdot r(f) + \overline{B} \cdot r(g)$. Therefore

$$\begin{aligned} r(f +_b g)_{st} = 1 \quad &\iff \quad [s \vDash b \wedge r(f)_{st} = 1] \vee \left[s \vDash \overline{b} \wedge r(g)_{st} = 1\right] \\ &\iff \quad [B_{ss} = 1 \wedge r(f)_{st} = 1] \vee \left[\overline{B}_{ss} = 1 \wedge r(g)_{st} = 1\right] \\ &\iff \quad \left[B \cdot r(f) + \overline{B} \cdot r(g)\right]_{st} = 1. \end{aligned}$$

For iteration, let $r\left(f^{(b)}\right) \triangleq (B \cdot r(f))^\star \cdot \overline{B}$. Consider the least n where $f^n(s) \vDash \overline{b}$. If $n = 0$,

$$\begin{aligned} r\left(f^{(b)}\right)_{st} = 1 \quad &\iff \quad t = s \wedge \overline{B}_{ss} = 1 \\ &\iff \quad t = s \wedge \left[\overline{B} + (B \cdot r(f))^\star \cdot \overline{B}\right]_{ss} = 1 \\ &\iff \quad t = s \wedge \left[(B \cdot r(f))^\star \cdot \overline{B}\right]_{ss} = 1. \end{aligned}$$

Similarly, if $n > 0$, let $s_i = f^i(s)$ then

$$\begin{aligned} r\left(f^{(b)}\right)_{st} = 1 \quad &\iff \quad t = s_n \wedge \left(\forall i < n.\, r\left(f^{i+1}\right)_{s_i s_{i+1}} = 1 \wedge B_{s_i s_i} = 1\right) \wedge \overline{B}_{tt} = 1 \\ &\iff \quad t = s_n \wedge (B \cdot r(f))^n_{st} = 1 \wedge \overline{B}_{tt} = 1 \\ &\iff \quad t = s_n \wedge \left[\overline{B} + (B \cdot r(f))^n \cdot \overline{B}\right]_{st} = 1 \\ &\iff \quad t = s_n \wedge \left[(B \cdot r(f))^\star \cdot \overline{B}\right]_{st} = 1. \end{aligned}$$

If n does not exist then $r\left(f^{(b)}\right)_{st} = 0$. Hence $\overline{B}_{tt} = 0$ and $\left[(B \cdot r(f))^\star \cdot \overline{B}\right]_{st} = 0$. □

4.2.2.3 Combined Semantics

Now that we have developed the semantics for pure packet and pure state programs, we can combine them to obtain the full stateful NetKAT language that can simultaneously reason about packets and state. In [32], a Boolean algebra of mutable state can be freely combined with any KAT in a uniform way such that the Kleene algebra and the Boolean algebra terms commute. This is done by taking the commutative coproduct of the two algebras. Following this approach, we combine our stateless and pure state algebras to form Stateful NetKAT, under the conditions when programs from the two worlds commute with each other. We call the interpretation of Stateful NetKAT the *combined* or *packet filtering semantics*.

Let P be the set of all stateless programs, and Q the set of pure state ones. Let D be a set of commutativity conditions, which we will later discuss in Section 4.5.5. We take the coproduct $P \oplus Q$ modulo D. Stateful NetKAT is the free algebra on this set as generators with operators from the syntax. When writing terms in Stateful NetKAT, we elide the injections from P and Q into the coproduct as they are implicit in the type of tests and actions.

A stateful NetKAT program p is interpreted as a function $[\![p]\!] : S \times H^? \to S \times H^\star + \bot$. As with Equation 4.1, we can extend naturally to the type $[\![p]\!]^\dagger : S \times H^\star + \bot \to S \times H^\star + \bot$ by processing the input string sequentially and concatenating the results:

$$
\begin{aligned}
[\![p]\!]^\dagger(\bot) &\triangleq \bot \\
[\![p]\!]^\dagger(s,h) &\triangleq [\![p]\!](s,h) \\
(h \neq \varepsilon \text{ and } x \neq \varepsilon)\ [\![p]\!]^\dagger(s,hx) &\triangleq \begin{cases} (u,yz), & [\![p]\!](s,h) = (t,y) \wedge [\![p]\!]^\dagger(t,x) = (u,z), \\ \bot, & \text{otherwise} \end{cases}
\end{aligned}
$$

If the context requires clarification of which semantics we are referring to, we write $[\![p]\!]_P : H \to H^\star + \bot$ for specifically the stateless interpretation if $p \in P$. Similarly we write $[\![p]\!]_Q : S \rightharpoonup S$ for the pure state interpretation if $p \in Q$.

The combined interpretation of p is defined inductively. For programs concerning packet fields, $[\![p]\!]\,(s, h)$ calculates the output from h as presented previously in the stateless semantics, while passing through the global state s unchanged. If $h = \varepsilon$, there is no packet to process, so $[\![p]\!]$ acts like the identity. Formally, for $p \in P$,

$$[\![p]\!]\,(s, \varepsilon) \triangleq (s, \varepsilon) \tag{4.4}$$

$$(h \neq \varepsilon)\ [\![p]\!]\,(s, h) \triangleq \begin{cases} \left(s, [\![p]\!]_P\,(h)\right), & [\![p]\!]_P\,(h) \neq \bot, \\ \bot, & \text{otherwise.} \end{cases} \tag{4.5}$$

On the other hand, if $p \in Q$ is a pure state program, it changes the global state as per its stateful interpretation $[\![p]\!]_Q$ while leaving the packet unchanged. The result is $\bot$ if the input state is not in $[\![p]\!]_Q$'s domain:

$$[\![p]\!]\,(s, h) \triangleq \begin{cases} \left([\![p]\!]_Q\,(s), h\right), & s \in \operatorname{dom} [\![p]\!]_Q, \\ \bot, & \text{otherwise.} \end{cases} \tag{4.6}$$

We can then combine programs as follows, while deferring the definition of $p^{(b)}$ to Section

4.2.2.4.

$$
\begin{aligned}
[\![p\,;q]\!]\,(s,h) &\triangleq [\![q]\!]^{\dagger}\,([\![p]\!]\,(s,h)) \\
[\![p \,\&\, q]\!]\,(s,h) &\triangleq \begin{cases} (u, xy)\,, & [\![p]\!]\,(s,h) = (t,x) \wedge [\![q]\!]\,(t,h) = (u,y)\,, \\ \bot, & \text{otherwise} \end{cases} \\
[\![p +_b q]\!]\,(s,h) &\triangleq \begin{cases} [\![p]\!]\,(s,h)\,, & (s,h) \vDash b, \\ [\![q]\!]\,(s,h)\,, & (s,h) \vDash \overline{b}. \end{cases}
\end{aligned}
$$

For tests, the main difficulty is being able to deal with a mix of stateless and pure state terms. Here we let $(s,h) \vDash b \iff s \vDash b$ if b is a pure state test, and similarly $(s,h) \vDash b \iff h \vDash b$ if it is stateless. The success relation $\vDash$ of (s,h) with $b\,;c$, $\overline{b}$, and $b \vee c$ follows naturally. This is for when the test appears as b in $+_b$ and $p^{(b)}$.

If b appears atomically by itself as a program however, $[\![b]\!]$ can still be well-defined from what we have described so far, although it is slightly more involved. There are three possible outcomes, forming a lattice, from the most desirable to the least:

1. $[\![b]\!](s,h) = (s,h)$, if b succeeds.
2. $[\![b]\!](s,h) = (s,\varepsilon)$, if b fails on some stateless test but execution can still continue normally by dropping the packet.
3. $[\![b]\!](s,h) = \bot$, if b fails on some state test.

To define $[\![b]\!]$, first we apply De Morgan's laws so that all negations are pushed to the leaves. These are the atomic stateless or pure state tests within b, and can be interpreted with $[\![-]\!]_P$ and

$[\![-]\!]_Q$ respectively. A conjunction is simply interpreted with the $[\![p\,;q]\!]$ definition from above, while a disjunction $[\![b \vee c]\!]\,(s,h)$ is equal to whichever of $[\![b]\!]\,(s,h)$ or $[\![c]\!]\,(s,h)$ that gives the most desirable outcome from the lattice.

In the case where $[\![b]\!]\,(s,h) = \bot$ if b fails as a state test, we can avoid the possibility of not being able to continue by instead using b as part of $+_b$. The term $\texttt{skip} +_b \texttt{drop}$ drops the packet if and only if b fails. In addition, Theorem 41 for the stateless semantics does *not* hold here in the presence of global state. The program

$$\texttt{if}\ A.x = 1\ \texttt{then}\ A.x \leftarrow 2\ \texttt{else}\ A.x \leftarrow 3$$

is guaranteed to only execute one of its branches, but

$$(A.x = 1\,;A.x \leftarrow 2)\ \&\ (A.x \neq 1\,;A.x \leftarrow 3)$$

results in $\bot$ as one of the state tests will fail.

On the other hand, if b is a packet test, it still behaves very differently as an atomic guard compared to as part of an operator. Consider $b\,;p$ and $p +_b q$. The former retains or drops the packet depending on whether b succeeds, but p is always executed afterwards, potentially changing the state. The latter executes p or q but never both. This is also why we differentiate `skip`/`drop` and `true`/`false`. They exhibit a difference in semantics when standing alone as atomic programs and when part of $+_b$ and $p^{(b)}$. As mentioned previously, it is possible to detect the presence (or absence) of a packet, leveraging an arbitrary field x, with the tests $\texttt{skip} \triangleq x = x$ and $\texttt{drop} \triangleq x \neq x$. The test $\texttt{true} \triangleq A.x = A.x$ succeeds and $\texttt{false} \triangleq A.x \neq A.x$ fails regardless of packet presence.

Even when a Stateful NetKAT term p is composed of pure stateless and state programs, the packet and state computations of $[\![p]\!]$ may not be independent. A simple example is

$$x \leftarrow 1 +_{A.x=1} x \leftarrow 2,$$

whose packet output evidently depends on the state input. In fact, the ; operator also demonstrates this dependence.

Example 45. Consider the term

$$p \triangleq (\texttt{skip} \,\&\, \texttt{skip}) \,;(A.x \leftarrow 2 +_{A.x=1} A.x \leftarrow 1)\,.$$

Let a state s be such that $A.x = 0$. The output state of $[\![p]\!]\,(s, \varepsilon)$ has $A.x = 1$. However, $[\![p]\!]\,(s, h)$ for $h \neq \varepsilon$ has $A.x = 2$. The cause of this behavior is $[\![-]\!]^\dagger$ in the interpretation of ;, which makes the output state dependent at least on the length of the input packet sequence.

Hence it can be seen that in general $[\![p\,;q]\!] \neq [\![q\,;p]\!]$ even if p is stateless and q is pure state. The commutativity conditions of Stateful NetKAT are more complex.

We can leverage the dependence on both the state and packet inputs to define cross assignments and tests between packet fields and state variables. This is allowed by the syntax of Stateful NetKAT but we have not considered them thus far.

Definition 46. An assignment of the state variable $A.x$ from the packet field y is syntactic sugar for

$$A.x \leftarrow y \triangleq A.x \leftarrow n_1 +_{y=n_1} \cdots +_{y=n_{k-1}} A.x \leftarrow n_k,$$

where $n_1, \ldots, n_k$ are all the allowed values of y. The opposite assignment $y \leftarrow A.x$ is defined similarly. On the other hand, a test of $A.x$ against y can be written as

$$A.x = y \triangleq (A.x = n_1 \,;\, y = n_1) \vee \ldots \vee (A.x = n_k \,;\, y = n_k)\,,$$

where the disjunction $b \vee c \triangleq \overline{\overline{b} \,;\, \overline{c}}$. In this case $y = A.x$ has the same definition.

4.2.2.4 Iteration in Stateful NetKAT

So far we have not yet defined the interpretation for iteration $p^{(b)}$. We consider an order on the interpretations of Stateful NetKAT programs. Previously we described an order $\sqsubseteq$ on $H^\star + \bot$ in the stateless semantics and extended it pointwise to $H \to H^\star + \bot$. There was also an order $\leq$ on $S \rightharpoonup S$ in pure state. By taking a construction that is similar to the product of these two orders, we can define an order on $f, g : S \times H^? \to S \times H^\star + \bot$:

$$f \sqsubseteq g \iff \forall\,(s,h)\,.\,f\,(s,h) = \bot \vee f\,(s,h) = g\,(s,h)\,.$$

We will again write ω for the constant $\bot$ function. Note that it is equal to $[\![\texttt{false}]\!]$. The definition of $[\![p^{(b)}]\!]$ is the least solution to the equation $[\![p^{(b)}]\!] = [\![p \,;\, p^{(b)} +_b \texttt{skip}]\!]$ in the above order. Its existence and uniqueness are implied by the following.

Theorem 47. *$(S \times H^? \to S \times H^\star + \bot, \sqsubseteq)$ is a DCPO with ω as the least element.*

It suffices to show that the map τ defined by

$$\tau(W) = \lambda(s,h)\,.\begin{cases} W^\dagger([\![p]\!](s,h)), & (s,h) \vDash b \\ (s,h), & (s,h) \vDash \overline{b} \end{cases}$$

is monotone in $\sqsubseteq$.

Lemma 48. *For all* $W, W' : S \times H^? \to S \times H^\star + \bot$,

$$W \sqsubseteq W' \implies \forall\, (s,x) \in S \times H^\star + \bot.\, W^\dagger\,(s,x) \sqsubseteq W'^\dagger\,(s,x)\,.$$

Proof. By induction on the length of x. The cases where x is null or a single packet follows directly from $W \sqsubseteq W'$. Otherwise, suppose first we do not encounter $\bot$. Then $W\,(s,h) = W'\,(s,h) = (t,y)$ and let

$$\begin{aligned} W^\dagger\,(s,hx) &= (u,yz) \quad \text{where } W^\dagger\,(t,x) = (u,z) \\ W'^\dagger\,(s,hx) &= (u',yz') \quad \text{where } W'^\dagger\,(t,x) = (u',z')\,. \end{aligned}$$

But by the induction hypothesis $W^\dagger\,(t,x)$ must equal $W'^\dagger\,(t,x)$, and so $W^\dagger\,(s,hx) = W'^\dagger\,(s,hx)$. If we do encounter $\bot$, $W\,(s,h)$ must be $\bot$ and so is $W^\dagger\,(s,hx)$. □

It then follows that τ is monotone because

$$\begin{aligned} \tau(W)(s,h) &= \begin{cases} W^\dagger([\![p]\!](s,h)), & (s,h) \vDash b \\ (s,h), & (s,h) \vDash \overline{b} \end{cases} \\ &\sqsubseteq \begin{cases} W'^\dagger([\![p]\!](s,h)), & (s,h) \vDash b \\ (s,h), & (s,h) \vDash \overline{b} \end{cases} \\ &= \tau(W')(s,h). \end{aligned}$$

4.2.2.5 Subsumption of Stateless and Pure State Operators

We end our description of the Stateful NetKAT semantics by highlighting the following *subsumption theorems.* They show the definition of $;$, $\&$, $+_b$, and $p^{(b)}$ in the combined semantics *subsume*

the stateless and pure state definitions of these operators earlier. For example, if p and q are stateless programs, the interpretation of $p\,;q$ in the stateless world should behave the same as $p\,;q$ as a Stateful NetKAT program acting on the packet component. The former $;$ is understood as a stateless operator while the latter is from Stateful NetKAT. It is therefore unnecessary to specify in which semantics an operator such as $;$ is being interpreted, as the single definition in the combined semantics *subsumes* both stateless and pure state versions. The expected behavior from both worlds is automatically embedded in Stateful NetKAT.

Theorem 49. *Let $p, q \in P$ be stateless programs. For all (s, h) we have*

$$\begin{aligned} [\![p\,;q]\!]\,(s,\varepsilon) &= (s,\varepsilon) \\ (h \neq \varepsilon)\ [\![p\,;q]\!]\,(s,h) &= \begin{cases} \left(s, [\![p\,;q]\!]_P\,(h)\right), & [\![p\,;q]\!]_P\,(h) \neq \bot, \\ \bot, & \textit{otherwise.} \end{cases} \end{aligned}$$

The operator $;$ on the left-hand side is from Stateful NetKAT, while on the right it is from the stateless language. The same equations hold if we replace $;$ with $\&$, or with $+_b$ for a stateless test b.

Proof. The equations are similar to 4.4 and 4.5, but the program on the left is a Stateful NetKAT term and it is not in the stateless language per se. The cases all follow from the definitions, with the most interesting one being that of $\&$ when neither p or q results in $\bot$. From the definition in this section,

$$[\![p \,\&\, q]\!]\,(s,h) = (u, xy) \text{ if } [\![p]\!]\,(s,h) = (t,x) \text{ and } [\![q]\!]\,(t,h) = (u,y)\,.$$

If p and q are stateless then necessarily $s = t = u$, hence

$$[\![p \,\&\, q]\!]\,(s,h) = (s, xy) = \left(s, [\![p]\!]_P \cdot [\![q]\!]_P\right) = \left(s, [\![p\,;q]\!]_P\,(h)\right).$$

□

A corresponding result holds for pure state programs.

Theorem 50. *Let $p, q \in Q$. For all (s, h),*

$$[\![p\,;q]\!]\,(s,h) = \begin{cases} \left([\![p\,;q]\!]_Q\,(s)\,,h\right), & s \in \operatorname{dom}[\![p\,;q]\!]_Q, \\ \bot, & \textit{otherwise,} \end{cases}$$

which also holds if we replace $;$ with $+_b$ for a pure state test b.

Proof. Again the equation bears similarity to 4.6. The $+_b$ case is trivial. For composition, assuming $s \in \operatorname{dom}[\![p\,;q]\!]_Q$, we must have $[\![p]\!]\,(s,h) = \left([\![p]\!]_Q\,(s)\,,h\right)$ as p does not change the input packet if it is pure state. Then

$$\begin{aligned} [\![p\,;q]\!]\,(s,h) &= [\![q]\!]^\dagger\,([\![p]\!]\,(s,h)) \\ &= [\![q]\!]^\dagger\left([\![p]\!]_Q\,(s)\,,h\right) \\ &= [\![q]\!]\left([\![p]\!]_Q\,(s)\,,h\right) \\ &= \left([\![q]\!]_Q\left([\![p]\!]_Q\,(s)\right),h\right) \\ &= \left([\![p\,;q]\!]_Q\,(s)\,,h\right). \end{aligned}$$

□

These theorems can also be extended to iteration.

Theorem 51. *Let $p, b \in P$ be a stateless program and test, then*

$$\begin{aligned} [\![p^{(b)}]\!](s,\varepsilon) &= (s,\varepsilon) \\ (h \neq \varepsilon)\ [\![p^{(b)}]\!](s,h) &= \begin{cases} \left(s, [\![p^{(b)}]\!]_P(h)\right), & [\![p^{(b)}]\!]_P(h) \neq \bot, \\ \bot, & \textit{otherwise.} \end{cases} \end{aligned}$$

Proof. Since p and b are stateless, an input state s is not changed in the output. Consider the suborder of $(S \times H^? \to S \times H^\star + \bot, \sqsubseteq)$ consisting of functions that take (s,ε) to (s,ε) and also fix s otherwise. This order is isomorphic to $(H \to H^\star + \bot, \sqsubseteq)$, and the definition of $[\![p^{(b)}]\!]$ here considers the same recursive equation as its stateless counterpart $[\![p^{(b)}]\!]_P$. Hence the output should agree with $[\![p^{(b)}]\!]_P$ in the packet component. □

Theorem 52. *Let $p, b \in Q$ be a pure state program and test. Then*

$$[\![p^{(b)}]\!](s,h) = \begin{cases} \left([\![p^{(b)}]\!]_Q(s), h\right), & s \in \operatorname{dom} [\![p^{(b)}]\!]_Q, \\ \bot, & \textit{otherwise.} \end{cases}$$

Proof. Let $f \triangleq [\![p]\!]_Q$, so $[\![p]\!](s,h)$ equals $(f(s), h)$ if $s \in \operatorname{dom} f$ and it is $\bot$ otherwise. Consider the function map τ defined from p and b as before. For any $W : S \times H^? \to S \times H^\star + \bot$ which fixes the packet component, $W(s,h) = W^\dagger(s,h)$. The action of τ on such a W is therefore

$$\tau(W) = \lambda(s,h).\begin{cases} \bot & (s,h) \vDash b \text{ and } s \notin \operatorname{dom} f \\ W(f(s),h), & (s,h) \vDash b \text{ and } s \in \operatorname{dom} f \\ (s,h), & (s,h) \vDash \overline{b}. \end{cases} \tag{4.7}$$

$\tau(W)$ also fixes the packet component. Now for all $s \in S$ let $n_s \in \mathbb{N}$ be the least number such that $f^{n_s}(s) \vDash \overline{b}$. From the pure state semantics, $[\![p^{(b)}]\!]_Q(s) = f^{n_s}(s)$ if n_s exists, or $s \notin \operatorname{dom} [\![p^{(b)}]\!]_Q$

otherwise. We claim for all $m \in \mathbb{N}$ and (s, h),

$$\tau^{m+1}(\omega)(s,h) = \tau(\omega)\left(f^{\min(m,n_s)}(s), h\right) \tag{4.8}$$

with the caveat that $\min(m, n_s)$ is m if n_s does not exist. This implies our theorem because $[\![p^{(b)}]\!]$ is the supremum

$$\bigsqcup_{0 \leq m} \{\tau^m(\omega)\}.$$

If $m < n_s$ or n_s does not exist,

$$\tau^{m+1}(\omega)(s,h) = \tau(\omega)(f^m(s), h) = \bot.$$

It is certain that $(f^m(s), h) \vDash b$ (otherwise $n_s \leq m$), so $\tau(\omega)(f^m(s), h) = \bot$ due to either $f^m(s) \notin \operatorname{dom} f$ or $\omega(f^m(s), h) = \bot$. Note that h is inconsequential if b is a pure state test. On the other hand, for $m \geq n_s$,

$$\tau^{m+1}(\omega)(s,h) = \tau(\omega)(f^{n_s}(s), h) = (f^{n_s}(s), h)$$

as $(f^{n_s}(s), h) \vDash \overline{b}$ by definition. Hence taking the supremum fixes $[\![p^{(b)}]\!](s,h)$ to $(f^{n_s}(s), h)$ if the latter exists.

Let $W = \tau^m(\omega)$. We must show that $\tau(W)$ from Equation 4.7 gives rise to $\tau^{m+1}(\omega)$ such that Equation 4.8 holds. This is trivial if $m = 0$. Otherwise, inducting on m, we unfold $\tau(W)(s,h)$ using Equation 4.7. There are two easy cases:

If $(s,h) \vDash \overline{b}$ then $n_s = 0$. Hence

$$\tau(W)(s,h) = (s,h) = \left(f^{\min(m,n_s)}(s), h\right) = \tau(\omega)\left(f^{\min(m,n_s)}(s), h\right).$$

If $(s,h) \vDash b$ and $s \notin \text{dom}\, f$ then $\tau(W)(s,h) = \bot$. In this case, certainly $s \notin \text{dom}\, [\![p^{(b)}]\!]_Q$, so n_s does not exist. As explained previously, $\tau(\omega)\left(f^{\min(m,n_s)}(s), h\right)$ is also $\bot$.

The final case is when $(s,h) \vDash b$ and $s \in \text{dom}\, f$. We have $\tau(W)(s,h) = W(f(s),h) = \tau^m(\omega)(f(s),h)$, which we analyze using the induction hypothesis. This is obtained by substituting $m-1$ for m and $f(s)$ for s in Equation 4.8:

$$\tau(W)(s,h) = \tau(\omega)\left(f^{\min\left(m-1,n_{f(s)}\right)}(f(s)), h\right) = \tau(\omega)\left(f^{\min(m-1,n_s-1)}(f(s)), h\right).$$

The right-hand side is again just $\tau(\omega)\left(f^{\min(m,n_s)}(s), h\right)$. □

4.2.3 Independence From Prior History

Just like NetKAT, the Stateless NetKAT semantic definitions do not examine any packet history prior to the head packet. This matches the fact that a real-life switch cannot examine packet data that existed in the past. However, the histories are still kept in order to semantically distinguish packets that evolved differently over time, for example if they traveled over different paths. In addition, there are no Stateful NetKAT terms that produce a new packet from scratch if there is no input packet. These properties can be formalized as:

Definition 53. A function $f : S \times H^? \to S \times H^\star + \bot$ is *history-independent* if, for all global states s and packet histories $\pi : h$ with head packet π, all of the following are true.

- If $f(s,\varepsilon) \neq \bot$ then $f(s,\varepsilon) = (s',\varepsilon)$ for some final state s'.
- If $f(s,\pi:\langle\rangle)$ equals $\bot$ or (s',ε) then $f(s,\pi:h) = f(s,\pi:\langle\rangle)$.

- If $f\left(s, \pi : \langle\rangle\right) = \left(s', \left(\pi_1 : h_1\right) \cdot \cdots \cdot \left(\pi_k : h_k\right)\right)$ then

$$f\left(s, \pi : h\right) = \left(s', \left(\pi_1 : h_1 : h\right) \cdot \cdots \cdot \left(\pi_k : h_k : h\right)\right).$$

From now on we can treat $S \times H^?$ as a finite set, as history-independence implies we only need to consider equivalence classes where packet histories have the same head packet.

4.3 Examples of Stateful NetKAT Reasoning

Before further developing the theory, we give some examples of common networking functionalities that can be easily specified with Stateful NetKAT. These examples were chosen for being difficult to implement in standard NetKAT, while being relatively simple programs in our language. They serve to highlight the added power we gain from adding state.

In the following, we write v^{++} as shorthand for the program that increments v with overflow within some range of values. For example, this can be achieved with

$$v \leftarrow 2 +_{v=1} \left(v \leftarrow 3 +_{v=2} v \leftarrow 1\right)$$

for the range $[1, 3]$.

4.3.1 Load Balancing

The simplest task that distinguishes Stateful NetKAT from standard NetKAT is load balancing the forwarding paths of incoming packets. Suppose switch A maintains a state variable A.nextport

that tracks on which egress port the next packet will be forwarded. Further, packets contain a header field eport that is *metadata*, fields that are not visible to the network at large but are used by the local switch hardware to carry out forwarding decisions. The program that forwards input packets to each egress port in a round-robin fashion can be written as

$$p \triangleq \mathsf{eport} \leftarrow A.\mathsf{nextport}\,;\, A.\mathsf{nextport}^{++}.$$

Although in NetKAT it is possible to use the $+$ operator to specify a combination of forwarding policies, in this case to forward to any one of the available egress ports, it is much harder to express *order*. The semantics $[\![p]\!]^\dagger$ of the the above simple program guarantees that the round-robin scheme is followed for each input packet, even if they are indistinguishable except for the order of arrival.

Just like NetKAT, we can use dup in Stateful NetKAT to distinguish different routes taken by packets. The program

$$p\,;\big(\mathsf{switch} \leftarrow A +_{\mathsf{eport}=1} \mathsf{switch} \leftarrow B\big)\,;\mathsf{eport} \leftarrow 1\,;\mathsf{dup}\,;\mathsf{switch} \leftarrow C$$

is distinguished from

$$p\,;\big(\mathsf{switch} \leftarrow A +_{\mathsf{eport}=1} \mathsf{switch} \leftarrow B\big)\,;\mathsf{eport} \leftarrow 1\,;\mathsf{switch} \leftarrow C.$$

Even if there is a sequence of identical packets at input, the output packet histories would not be identical in the former program.

4.3.2 Port Triggering

Another mechanism that global state facilitates is a one-way flag. Consider a firewall that initially blocks all incoming traffic. Once an outgoing packet is sent on port 42, however, all incoming traffic on that port becomes allowed. Suppose an internal host has IP address $192.168.1.132$, and the switch maintains a Boolean state variable $A.\mathsf{open}$. We can write this policy as

$$\left(\mathsf{port} = 42 +_{A.\mathsf{open}=\mathtt{true}} \mathtt{drop}\right) +_{\mathsf{dest}=192.168.1.132} \left(A.\mathsf{open} \leftarrow \mathtt{true} +_{\mathsf{port}=42} \mathtt{skip}\right),$$

where dest is the packet destination IP address, and all traffic not incoming to the internal host is assumed to be outgoing.

Similarly, we can write an anti-denial-of-service firewall policy where initially port 42 is open, but then becomes closed after n packets have been received:

$$\left(\mathtt{drop} +_{A.\mathsf{count}=n} A.\mathsf{count}^{++}\right) +_{\mathsf{port}=42} \mathtt{skip}.$$

4.3.3 Packet Duplication

Besides the order in which packets are processed, we can also reason about and control their *multiplicity*. The number of packets in the output is controlled by the use of the $\&$ operator. We can easily produce a constant number of output packets from a single input, such as $x \leftarrow 1 \ \& \ x \leftarrow 2$, which duplicates the input packet and sets the field x to 1 in the left copy and to 2 in the right copy. When combined with iteration, however, we can create duplicates based on runtime state values. For example, with a variable $A.\mathsf{count}$ starting at 0,

$$w \triangleq \left(\mathtt{skip} \ \& \ A.\mathsf{count}^{++}\right)^{(A.\mathsf{count} \neq A.\mathsf{copies})}$$

will create A.copies duplicates of the input packet determined at runtime. Note that after duplication with $\&$, the iteration is applied to *both* copies since

$$\begin{aligned}
[\![w]\!](s,h) &= [\![(\texttt{skip} \,\&\, A.\mathsf{count}^{++})\,;w]\!](s,h) \\
&= [\![w]\!]^{\dagger}\left([\![\texttt{skip} \,\&\, A.\mathsf{count}^{++}]\!](s,h)\right) \\
&= [\![w]\!]^{\dagger}(s', h \cdot h) \\
&= [\![w]\!](s', h) \cdot [\![w]\!](s'', h)
\end{aligned}$$

where s' is the state after incrementing A.count once in s, and s'' is the resulting state of $[\![w]\!](s', h)$. However, exactly A.copies are created as the global state threads through the concatenated terms, and w becomes equivalent to `skip` after A.count equals A.copies.

4.3.4 Network Address Translation and Other Applications

Finally, we can take advantage of tests and assignments between state and packet variables to implement Network Address Translation (NAT). A simple scheme is store in the state the origin IP address of an outgoing packet from a port, and any subsequent incoming packets on that port will be forwarded to the stored address:

$$\big(\mathsf{dest} \leftarrow A.\mathsf{fwdaddr} +_{\mathsf{incoming}=\mathsf{true}} A.\mathsf{fwdaddr} \leftarrow \mathsf{origin}\big) +_{\mathsf{port}=42} \texttt{skip}.$$

In general, global state can also work in tandem with more specialized networking hardware to allow more more complex packet processing at high throughput. An example is prior work on push-in first-out queues [59, 8], which proposes forwarding packets based on ordering them

dynamically in a priority queue. Suppose the priority queue is implemented in hardware, and incoming packets are tagged with a unique id. The map from packet ids to priority can be exposed as global state. The programmer can then specify, using Stateful NetKAT, the computation that calculates the priority from each incoming packet and makes updates to the map of priorities.

4.4 Language Model

As with all languages that have a denotational semantics, semantic equivalence induces a congruence relation on terms. For Stateful NetKAT terms e and e', we write $e \equiv e' \triangleq [\![e]\!] = [\![e']\!]$.

Theorem 54. *The relation $\equiv$ is decidable.*

The theorem is implied directly by the fact that $[\![e]\!]$ for all e is a computable function. Intuitively this is true because, due to history-independence, we can treat $[\![e]\!]$ as a function with a finite domain. With appropriate cycle detection, we can always compute $[\![e]\!](s,h)$ in finite time even if e contains loops that result in $\bot$. More formally, consider a relation $(S \times H^?, <)$ such that $(s,h) < (s',h')$ if and only if the computation of $[\![e]\!](s',h')$ requires computing $[\![e]\!](s,h)$ as a dependency. This is a finitely computable relation, and $[\![e]\!](s,h) = \bot$ if and only if $(s,h) < (s,h)$.

Instead of relying on computing the entire function $[\![e]\!]$, however, we would much rather work on the expression e itself. Towards this end, we develop a language model for Stateful NetKAT. Later in Section 4.5, this evolves into a decision procedure by leveraging the theory of Kleene algebra, like in standard NetKAT, that manipulates the terms of the language.

4.4.1 Preliminaries

Theorem 55. *Stateful NetKAT tests form a Boolean algebra.*

The interpretation of a test b within the Boolean algebra is the set of all $S \times H^?$ pairs (s, h) such that $(s, h) \vDash b$. Note again that the packet filtering semantics of some tests may coincide, such as $[\![\texttt{true}]\!] = [\![\texttt{skip}]\!]$, when they appear as primitive programs. They are nevertheless distinct in the Boolean algebra and have different effects when used with the conditional and iteration. As usual for a Boolean algebra, there exists a partial order $\leq$ on tests such that

$$b \leq c \triangleq b \vee c = c,$$

which also corresponds to the subset relation between test interpretations.

Definition 56. Suppose the global state consists of variables $A.x_1, \ldots, A.x_i$, and packet fields are named $x_1, \ldots, x_j$. A *complete state test* t_s is a test of every state variable against some constant valid for each variable:

$$A.x_1 = n_1 ; \ldots ; A.x_i = n_i.$$

A *complete packet test* t_p is either `drop` or a test of every packet field against some constant valid for that field:

$$x_1 = m_1 ; \ldots ; x_j = m_j.$$

A *complete test* is a pair of composed state and packet tests $\alpha = t_s ; t_p$. We denote the set of all complete tests by At.

Theorem 57. *For a complete test α and some test b, if* $\texttt{false} < b \leq \alpha$ *then* $b = \alpha$. *It is also the case that* $\left(\bigvee_{\alpha \in \mathsf{At}} \alpha\right) = \texttt{true}$. *Hence* At *are the atoms of the Boolean algebra.*

This follows directly from each complete test specifying some largest subset of $S \times H^?$ containing packet histories that have the same head packet.

Definition 58. Similar to complete state (packet) tests, *complete state (packet) assignments* are assignments of each variable (field) to some constant:

$$A.x_1 \leftarrow n_1 \,;\, \ldots \,;\, A.x_i \leftarrow n_i.$$

For some complete test α, its corresponding complete state assignment σ_α is simply α's state variable tests written as assignments, and similarly for the corresponding complete packet assignment π_α. The packet "assignment" that corresponds to `drop` is just `drop` itself, as it has the effect of removing the packet if present. The set of all complete state assignments is Σ. The set of all complete packet assignments besides `drop` is Π.

Given the above definitions, we can give a language model for Stateful NetKAT using *guarded Stateful NetKAT strings*. They correspond to valid input-output relations of a Stateful NetKAT program.

Definition 59. A *guarded Stateful NetKAT string* is composed from the alphabet

$$\mathsf{At} \cup \Sigma \cup \Pi \cup \{\langle\!\rangle, \bot\}\,.$$

The test `drop` may appear as part of a complete test in At, but it does not occur on its own as a symbol. Further, guarded Stateful NetKAT strings match the regular expression

$$I \triangleq \mathsf{At} \cdot \bot + \mathsf{At} \cdot (\langle\!\rangle \cdot \Pi^\star \cdot \Pi)^\star \cdot \Sigma.$$

Firstly, the string is *guarded* by a complete test. This specifies the input state-packet that will be processed. The string then describes the output that will be computed, taking one of three forms:

- $\mathsf{At} \cdot \bot$ means the computation will not terminate.
- $\mathsf{At} \cdot \Sigma$ means no output packet will be produced.
- $\mathsf{At} \cdot \between \cdot \Pi^\star \cdot \Pi \cdot \dots \cdot \Sigma$ means a non-empty sequence of packet histories will be produced. The symbol $\between$ is a delimiter that separates individual packet histories in the output sequence. Each packet history can be unbounded in size through concatenating packets expressed as members of Π. Note that the expression $(\Pi \cdot \mathtt{dup})^\star \cdot \Pi$ is how the output part of NetKAT guarded strings are specified in [9], but here we can elide all the occurrences of dup. The ordering of symbols in the string corresponds to the order of outputs of a Stateful NetKAT program on the input specified by an element of At.

Note that in the latter two forms where there is some meaningful output, the string always ends with some member of Σ, which indicates the global state at the end of the computation. We denote the set of all guarded Stateful NetKAT strings by I. Just as input guards At correspond to subsets of the packet filtering semantics domain $S \times H^?$, the outputs specified by guarded strings correspond to elements of the co-domain $S \times H^\star + \bot$.

Definition 60. A set of guarded strings $X \subseteq I$ is *total* if every possible guard appears exactly once. That is

$$\forall \alpha.\, \exists! x.\, \alpha \cdot x \in X.$$

If X is total then it gives a deterministic set of outputs for every possible state-packet input.

We can now interpret a Stateful NetKAT program p in this model, developed in the next sections. We denote this interpretation as $G(p)$ and it is defined inductively from subsets of I. Along the way we will also prove:

Theorem 61. *$G(p)$ is always total.*

4.4.2 Primitives and Conditional

For conciseness we will elide "$\alpha \in \mathsf{At}$" in set comprehensions when we define sets of guarded strings if α ranges over all complete tests. For packet and state assignments, we modify the output complete assignments as appropriate:

$$\begin{aligned} G(x \leftarrow v) &\triangleq \{\alpha \cdot \sigma_\alpha \mid \pi_\alpha = \mathtt{drop}\} \cup \{\alpha \cdot \langle\!\rangle \cdot \pi_\alpha [\alpha(v)/x] \cdot \sigma_\alpha \mid \pi_\alpha \neq \mathtt{drop}\} \\ G(A.x \leftarrow v) &\triangleq \{\alpha \cdot \sigma_\alpha [\alpha(v)/A.x] \mid \pi_\alpha = \mathtt{drop}\} \\ &\cup \{\alpha \cdot \langle\!\rangle \cdot \pi_\alpha \cdot \sigma_\alpha [\alpha(v)/A.x] \mid \pi_\alpha \neq \mathtt{drop}\}. \end{aligned}$$

The term $\alpha(v)$ is v if v is a constant, or it is the value being tested against v in α if v is a packet field or state variable. The terms $\pi_\alpha [\alpha(v)/x]$ and $\sigma_\alpha [\alpha(v)/A.x]$ denote the result of substituting the right-hand side of the assignment to x (resp. $A.x$) in the complete assignment with $\alpha(v)$. If v is a packet field and $\pi_\alpha = \mathtt{drop}$, i.e. no input packet exists, then $\sigma_\alpha [\alpha(v)/A.x] = \sigma_\alpha$.

For dup, we take the input packet as the output if it exists, but also append a copy to the history:

$$G(\mathtt{dup}) \triangleq \{\alpha \cdot \sigma_\alpha \mid \pi_\alpha = \mathtt{drop}\} \cup \{\alpha \cdot \langle\!\rangle \cdot \pi_\alpha \cdot \pi_\alpha \cdot \sigma_\alpha \mid \pi_\alpha \neq \mathtt{drop}\}.$$

For a guarded string, the head packet is at the *rightmost* position π' in the substring $\pi \cdot \cdots \cdot \pi'$. The interpretations of tests, assignments and dup are evidently total as exactly one string is given for each $\alpha \in \mathsf{At}$.

Sets of guarded strings can be combined by the conditional:

$$\begin{aligned} G(p +_b q) &\triangleq G(p) +_b G(q) \\ &\triangleq \{\alpha \cdot x \mid \alpha \cdot x \in G(p), \alpha \leq b\} \cup \left\{\beta \cdot y \mid \beta \cdot y \in G(q), \beta \leq \overline{b}\right\}. \end{aligned}$$

We retain only the strings in $G(p)$ that have guards satisfying b, and those in $G(q)$ whose guards satisfy $\overline{b}$. The two subsets have mutually exclusive guards, and from the totality of $G(p)$ and $G(q)$ we must have every possible guard appearing exactly once.

For a positive packet test b, consider each $\alpha \in \mathsf{At}$. If $\alpha \leq b$ then the input packet and state are passed to the output, otherwise the input packet is dropped, i.e.

$$G(b) \triangleq \{\alpha \cdot \langle\!\rangle \cdot \pi_\alpha \cdot \sigma_\alpha \mid \alpha \leq b, \pi_\alpha \neq \mathtt{drop}\} \cup \{\alpha \cdot \sigma_\alpha \mid \alpha \not\leq b \text{ or } \pi_\alpha = \mathtt{drop}\}.$$

If b is instead a state test, the result is $\bot$ if $\alpha \not\leq b$:

$$\begin{aligned} G(b) \triangleq\ & \{\alpha \cdot \langle\!\rangle \cdot \pi_\alpha \cdot \sigma_\alpha \mid \alpha \leq b, \pi_\alpha \neq \mathtt{drop}\} \cup \{\alpha \cdot \sigma_\alpha \mid \alpha \leq b, \pi_\alpha = \mathtt{drop}\} \\ & \cup \{\alpha \cdot \bot \mid \alpha \not\leq b\}. \end{aligned}$$

To compose tests, as in packet filter semantics, we first push negations to the leaves. We can define $G\left(\overline{b}\right)$ as the set with the opposite outcomes as $G(b)$ by swapping the $a \leq b$ and $a \not\leq b$ conditions. Conjunctions can be built from the interpretations of ; that will appear shortly. For disjunction, let

$$G(b \vee c) \triangleq \{\alpha \cdot (x \sqcup y) \mid \alpha \cdot x \in G(b), \alpha \cdot y \in G(c)\}$$

where $x \sqcup y$ is the supremum of x and y in the lattice $\bot < \sigma_\alpha <\!\!\!\cdot\, \pi_\alpha \cdot \sigma_\alpha$.

4.4.3 Compositions

To interpret parallel composition, we can define a special kind of concatenation $\blacklozenge$ on guarded strings.

Definition 62. Consider some $\alpha, \beta \in \mathrm{At}$ and $\sigma, \sigma' \in \Sigma$. Let

$$\begin{aligned} \alpha \cdot \bot \blacklozenge \beta \cdot y \cdot \sigma' &\triangleq \alpha \cdot \bot, \\ \alpha \cdot x \cdot \sigma \blacklozenge \beta \cdot y \cdot \sigma' &\triangleq \begin{cases} \alpha \cdot x \cdot y \cdot \sigma', & \sigma_\beta = \sigma \text{ and } \pi_\beta = \pi_\alpha, \\ \text{undefined}, & \text{otherwise.} \end{cases} \end{aligned}$$

Note that x is possibly ε. For $X, Y \subseteq I$, $X \blacklozenge Y$ is pairwise concatenation between members of X and Y

$$X \blacklozenge Y \triangleq \{x \blacklozenge y \mid x \in X, y \in Y\}.$$

Theorem 63. *If X and Y are total then $X \blacklozenge Y$ is total.*

Proof. As Y is total, for each $\alpha \cdot x \in X$ there is in general exactly one $\beta \cdot y \in Y$ such that $\alpha \cdot x \blacklozenge \beta \cdot y$ is defined. The exception is $\alpha \cdot \bot$, which itself is the unique result of the concatenation. The concatenated string is still guarded by α, so the resulting set is total because X is total. □

For parallel composition, the language model interpretation is simply

$$G(p \,\&\, q) \triangleq G(p) \blacklozenge G(q)$$

and it is total assuming that is true for both $G(p)$ and $G(q)$.

Theorem 64. $\blacklozenge$ *is associative.*

Proof. The cases involving $\bot$ are straightforward. Otherwise, consider when the following equations hold:

$$\begin{aligned} \alpha \cdot x \cdot \sigma \blacklozenge \beta \cdot y \cdot \sigma' &= \alpha \cdot x \cdot y \cdot \sigma' \\ \beta \cdot y \cdot \sigma' \blacklozenge \gamma \cdot z \cdot \sigma'' &= \beta \cdot y \cdot z \cdot \sigma''. \end{aligned}$$

The first equation implies $\sigma_\beta = \sigma$ and $\pi_\beta = \pi_\alpha$. The second equation implies $\sigma_\gamma = \sigma'$ and $\pi_\gamma = \pi_\beta$, so $\pi_\gamma = \pi_\alpha$. Hence

$$\begin{aligned} (\alpha \cdot x \cdot \sigma \blacklozenge \beta \cdot y \cdot \sigma') \blacklozenge \gamma \cdot z \cdot \sigma'' &= \alpha \cdot x \cdot y \cdot \sigma' \blacklozenge \gamma \cdot z \cdot \sigma'' \\ &= \alpha \cdot x \cdot y \cdot z \cdot \sigma'' \\ &= \alpha \cdot x \cdot \sigma \blacklozenge (\beta \cdot y \cdot z \cdot \sigma'') \\ &= \alpha \cdot x \cdot \sigma \blacklozenge (\beta \cdot y \cdot \sigma' \blacklozenge \gamma \cdot z \cdot \sigma'')\,. \end{aligned}$$

□

For $G\,(p\,;q)$, we need to mimic the monadic composition where $[\![q]\!]$ is computed on each packet output of $[\![p]\!]$ in sequence. This is done by introducing another type of concatenation $x \diamond Y$ whose result is a set. Firstly:

Definition 65. For an indexed set of strings $\{x_i\}_{1 \le i \le k}$, we write for their concatenation

$$\bigotimes_{i=1}^{k} x_i \triangleq x_1 \cdot \cdots \cdot x_k.$$

This is then used to define the $\{\!\!\{$-delimited concatenation

$$\bigbraces_{i=1}^{k} x_i \triangleq \bigotimes_{i=1}^{k} (\{\!\!\{ \cdot x_i) = \{\!\!\{ \cdot x_1 \cdot \cdots \cdot \{\!\!\{ \cdot x_k.$$

If any $x_i = \varepsilon$ then the section $\{\!\!\{ \cdot x_i$ is omitted.

Now, suppose x is a guarded string, but in contrast to $\blacklozenge$, Y is a set of guarded strings. For the first two forms of x,

$$\alpha \cdot \bot \diamond Y \triangleq \{\alpha \cdot \bot\} \tag{4.9}$$

$$\alpha \cdot \sigma \diamond Y \triangleq \left\{\alpha \cdot y \mid \beta \cdot y \in Y, \sigma_\beta = \sigma, \pi_\beta = \texttt{drop}\right\}. \tag{4.10}$$

However, if $x = \alpha \cdot \left(\bigbraces_{i=1}^{k} x_i \cdot \pi_i\right) \cdot \sigma$ where each $x_i \cdot \pi_i$ is a string in $\Pi^\star \cdot \Pi$, we need to consider Y at each delimited packet history in x in turn. Each string in $x \diamond Y$ is generated by the following process. Assuming we do not encounter any strings involving $\bot$, define a string $z_1 \cdot \sigma_1$ as follows for the guard β_1 where $\sigma_{\beta_1} = \sigma$ and $\pi_{\beta_1} = \pi_1$:

$$z_1 \cdot \sigma_1 \triangleq \begin{cases} \alpha \cdot \sigma_1, & \beta_1 \cdot \sigma_1 \in Y, \\ \alpha \cdot \left(\bigbraces_{j=1}^{\ell} x_1 \cdot y_j\right) \cdot \sigma_1, & \beta_1 \cdot \left(\bigbraces_{j=1}^{\ell} y_j\right) \cdot \sigma_1 \in Y. \end{cases} \tag{4.11}$$

Then define inductively for the guard β_i where $\sigma_{\beta_i} = \sigma_{i-1}$ and $\pi_{\beta_i} = \pi_i$:

$$z_i \cdot \sigma_i \triangleq \begin{cases} z_{i-1} \cdot \sigma_i, & \beta_i \cdot \sigma_i \in Y, \\ z_{i-1} \cdot \left(\bigbraces_{j=1}^{\ell} x_i \cdot y_j\right) \cdot \sigma_i, & \beta_i \cdot \left(\bigbraces_{j=1}^{\ell} y_j\right) \cdot \sigma_i \in Y. \end{cases}$$

The generated string is then $z_k \cdot \sigma_k$. The set $x \diamond Y$ contains such generated strings for all combinations of $\beta_i \cdot y_i \cdot \sigma_i \in Y$ that satisfy the above definitions. If at any point we encounter a string

$\beta_i \cdot \bot \in Y$, the generated string is $\alpha \cdot \bot \in x \diamond Y$. On the other hand, if at any point we do not find a $\beta_i \cdot y_i \cdot \sigma_i \in Y$ satisfying the conditions on σ_{β_i} and π_{β_i}, the result is undefined.

Similar to $\blacklozenge$, we can also write

$$X \diamond Y \triangleq \bigcup_{\alpha \cdot x \in X} (\alpha \cdot x) \diamond Y$$

for the set of guarded strings X.

Theorem 66. *If X and Y are total then $X \diamond Y$ is total.*

Proof. If Y is total, for any $\alpha \cdot x \in X$, $\alpha \cdot x \diamond Y$ is a singleton set of a string guarded by α. If X is total then every possible α appears exactly once. □

We can thus define

$$G\,(p\,;q) \triangleq G\,(p) \diamond G\,(q)\,.$$

To see why this definition works, we only need to consider the non-trivial case $x = \alpha \cdot \left(\bigdiamond_{i=1}^{k} x_i \cdot \pi_i\right) \cdot \sigma \in G\,(p)$. Note that π_i and x_i are the i-th head packet and history respectively in x, and σ is the final state after executing p. To compose with q, we operate on the first packet π_1 while in state σ. This specifies β_1. The computation of q is equivalent to replacing this first packet with the entire output sequence in $G\,(q)$ that is guarded by β_1, while retaining the history x_1 in each output packet. At the end we land in state σ_1. The whole process is repeated for each packet history $x_i \cdot \pi_i$ in x, accumulating the result in the output string, except each time we start with the previous global state σ_{i-1}. At each step $z_i \cdot \sigma_i$ is well-defined as there is exactly one string guarded by β_i in the total set $G\,(q)$.

Theorem 67. $\diamond$ *is associative.*

Towards a proof of this theorem, let $\rhd$ be a binary operation with type

$$\rhd\colon \Pi^\star \cdot \Pi \to (\wr \cdot \Pi^\star \cdot \Pi)^\star \to (\wr \cdot \Pi^\star \cdot \Pi)^\star.$$

It has the definition

$$\begin{aligned} x \cdot \pi \rhd \varepsilon &\triangleq \varepsilon \\ x \cdot \pi \rhd \bigwr_{j=1}^{\ell} y_j &\triangleq \bigwr_{j=1}^{\ell} (x \cdot y_j)\,. \end{aligned}$$

In other words $x \cdot \pi \rhd y$ takes the headless history x and prepends it to each packet history in y.

We can in fact condense the inductive process for the non-trivial case of $\diamond$ into a single short lemma using $\rhd$:

Lemma 68. *Let $x = \alpha \cdot \left(\wr_{i=1}^{k} x_i \cdot \pi_i\right) \cdot \sigma_0$, and Y be a set of guarded strings. Suppose there is an indexed subset of Y called Y' such that $Y' = \{\beta_i \cdot y_i \cdot \sigma_i\}_{1 \le i \le k}$ satisfies $\sigma_{\beta_i} = \sigma_{i-1}$ and $\pi_{\beta_i} = \pi_i$ for all i. Assigning multiple indices to the same member of Y is allowed. Consider the string*

$$\alpha \cdot \bigotimes_{i=1}^{k} (x_i \cdot \pi_i \rhd y_i) \cdot \sigma_k.$$

This string is one element of $x \diamond Y$. The set $x \diamond Y$ consists of all such strings, corresponding to the different ways of picking Y' such that the required conditions on β's and σ's hold.

Proof. The proof is similar to that of Theorem 64. We only need to consider each individual string in X, and the simple cases of $\diamond$ in Equations 4.9 and 4.10 are straightforward. For $x = \alpha \cdot \left(\wr_{i=1}^{k} x_i \cdot \pi_i\right) \cdot \sigma_0 \in X$, suppose the concatenation with Y and Z is non-empty. Without loss of generality, let us pick a particular indexed $Y' \subseteq Y$ such that $Y' = \left\{\beta_i \cdot \left(\wr_{j=1}^{\ell_i} y_{ij} \cdot \pi'_{ij}\right) \cdot \sigma_i\right\}_{1 \le i \le k}$

satisfies the conditions in Lemma 68. Similarly, for every $1 \leq i \leq k$ there must exist a $Z'_i \subseteq Z$, where $Z'_i = \left\{\gamma_{ij} \cdot z_{ij} \cdot \sigma'_{ij}\right\}_{1 \leq j \leq \ell_i}$, such that the required conditions with the i-th string of Y' are satisfied. □

This lemma can be used to prove the associativity of $\diamond$. Ignoring other strings that could be generated for different picks of Y' and Z', as per the lemma we have

$$\begin{aligned} x \diamond Y' &= \left\{ \alpha \cdot \bigotimes_{i=1}^{k} \left(x_i \cdot \pi_i \rhd \bigcirc\!\!\!\!\!\!\bigcirc_{j=1}^{\ell_i} y_{ij} \cdot \pi'_{ij} \right) \cdot \sigma_k \right\} \\ &= \left\{ \alpha \cdot \bigotimes_{i=1}^{k} \left(\bigcirc\!\!\!\!\!\!\bigcirc_{j=1}^{\ell_i} x_i \cdot y_{ij} \cdot \pi'_{ij} \right) \cdot \sigma_k \right\} \end{aligned}$$

and

$$\text{for all } i,\ \beta_i \cdot \left(\bigcirc\!\!\!\!\!\!\bigcirc_{j=1}^{\ell_i} x_i \cdot y_{ij} \cdot \pi'_{ij} \right) \cdot \sigma_i \diamond Z'_i = \left\{ \beta_i \cdot \bigotimes_{j=1}^{\ell_i} \left(x_i \cdot y_{ij} \cdot \pi'_{ij} \rhd z_{ij} \right) \cdot \sigma'_{\ell_i} \right\}.$$

Hence

$$\begin{aligned} (x \diamond Y') \diamond Z &= \left\{ \alpha \cdot \bigotimes_{i=1}^{k} \bigotimes_{j=1}^{\ell_i} \left(x_i \cdot y_{ij} \cdot \pi'_{ij} \rhd z_{ij} \right) \cdot \sigma'_{\ell_k} \right\} \\ &= \left\{ \alpha \cdot \bigotimes_{i=1}^{k} \left[x_i \cdot \pi_i \rhd \bigotimes_{j=1}^{\ell_i} \left(y_{ij} \cdot \pi'_{ij} \rhd z_{ij} \right) \right] \cdot \sigma'_{\ell_k} \right\} \\ &= x \diamond (Y' \diamond Z)\,. \end{aligned}$$

Theorem 69. *$G\,(\texttt{true})$ is an identity for $\diamond$ and $G\,(\texttt{false})$ is an annihilator for $\diamond$ and $\blacklozenge$.*

4.4.4 Iteration

Similar to the packet filtering semantics for `while` b `do` p, the interpretation in the language model is

$$G\left(p^{(b)}\right) \triangleq \text{ the least subset of } I \text{ s.t. } G\left(p^{(b)}\right) = G\left(\left(p\,;p^{(b)}\right) +_b \texttt{skip}\right).$$

The right-hand side is equal to

$$\left(G\left(p\right) \diamond G\left(p^{(b)}\right)\right) +_b G\left(\texttt{skip}\right).$$

This set must be total as we have shown it is the case for all other operators. To define the least subset, we first give an order $\sqsubseteq$ on guarded strings. The relation can only hold among strings with the same guard α, and $\alpha \cdot x \sqsubseteq \alpha \cdot y$ if and only if $x = \bot$ or $x = y$. We can then extend this to total sets of guarded strings $X, Y \subseteq I$ as

$$X \sqsubseteq Y \iff \forall \alpha \in \mathsf{At}, \alpha \cdot x \in X, \alpha \cdot y \in Y. \, \alpha \cdot x \sqsubseteq \alpha \cdot y.$$

The least solution of $G\left(p^{(b)}\right)$ is with respect to this order.

Theorem 70. *The order $\sqsubseteq$ over total subsets of I is a DCPO with least element $G\left(\texttt{false}\right) = \{\alpha \cdot \bot\}$. Further, the map τ on a total $W \subseteq I$ given by*

$$\tau\left(W\right) \triangleq \left(G\left(p\right) \diamond W\right) +_b G\left(\texttt{skip}\right)$$

for some p and b is always monotone in $\sqsubseteq$. Hence there is a unique least solution for $G\left(p^{(b)}\right)$.

4.4.5 Isomorphism with Packet Filtering Semantics

We conclude our presentation by pointing out that there are classes of strings in I that can never appear in the language model for well-formed Stateful NetKAT terms. The following is one example.

Theorem 71. *Strings of the form $\alpha \cdot \langle\!\rangle \cdot \pi \cdot \cdots \cdot \sigma$, where $\pi_\alpha = \texttt{drop}$, exist in I but not in $G(p)$ for any term p.*

Proof. Consider the definition of G on the primitive terms. If a guard α has $\pi_\alpha = \texttt{drop}$, the correponding guarded string is always in the form $\alpha \cdot x$ for some $x \in \Sigma + \bot$. The construction of the language model for a term that is p and q connected by an operator builds inductively on $G(p)$ and $G(q)$. By the induction hypothesis, any string in $G(p)$ with guard α and $\pi_\alpha = \texttt{drop}$ must be $\alpha \cdot x$ for $x \in \Sigma + \bot$, and similarly for those in $G(q)$. The interpretation $G(p +_b q)$ only takes strings verbatim from $G(p)$ and $G(q)$. For $G(p \,\&\, q)$ and $G(p\,;q)$, a string $\alpha \cdot x \in G(p)$ is only concatenated or composed with $\beta \cdot y \in G(q)$ if $\pi_\alpha = \pi_\beta$, and if this is $\texttt{drop}$ then both x and y are in $\Sigma + \bot$. The definition of $G\left(p^{(b)}\right)$ follows from the other operators. □

This corresponds to the packet filtering semantics given previously, in which packets cannot be created out of thin air. There must be an input packet for there to be a non-empty output sequence. In fact, there exists an isomorphism between the language model of Stateful NetKAT and those semantics, therefore demonstrating the language model is sound and complete. Let $(s, h) \in S \times H^?$ be an arbitrary state and optional packet history pair, and let α be the corresponding complete test. That is, α's complete state test asserts the global state is equal to s. Its

complete packet test is `drop` if $h = \varepsilon$, or it asserts the input packet is the head packet of h. Again due to history-independence, it is sufficient to specify only the head packet with α, instead of the full input packet history. We prove the following facts.

Lemma 72. $(s, h) \vDash b \iff \alpha \leq b$.

Recall from Theorem 55 that the interpretation of a test b is the subset of $S \times H^?$ whose members satisfy the $\vDash$ relation with b, and $\leq$ corresponds to the subset relation. The guard α identifies the singleton set $\{(s, h)\}$, so the lemma follows.

Theorem 73. *For any Stateful NetKAT term p, the following properties hold, characterizing the isomorphism between $[\![p]\!]$ and $G(p)$.*

1. $[\![p]\!](s, h) = \bot \iff \alpha \cdot \bot \in G(p)$.
2. $[\![p]\!](s, h) = (s', \varepsilon) \iff \alpha \cdot \sigma_{s'} \in G(p)$, *where $\sigma_{s'}$ is s' as a complete state assignment.*
3. $[\![p]\!](s, h) = (s', h_1 \cdot \dots \cdot h_k) \iff \alpha \cdot \left(\bigdiamond_{i=1}^{k} h_i\right) \cdot \sigma_{s'} \in G(p)$, *where each packet history $h_i = \pi_1 : \dots : \pi_j$ is expressed in the guarded string as $\pi_1 \cdot \dots \cdot \pi_j$.*

Proof. By induction on the structure of p. In particular the recursive equation for $G\left(p^{(b)}\right)$ is the same as that for $[\![p^{(b)}]\!]$, and we consider an order $\sqsubseteq$ on total subsets of I that is also isomorphic to the order considered in the packet filtering semantics. Note that $[\![p]\!](s, h)$ is well-defined, and correspondingly there is always exactly one string guarded by α in $G(p)$ due to totality. □

4.5 Decision Procedure

So far we have seen that the equational theory of Stateful NetKAT is decidable, as the complete input-output relation of the semantics can be enumerated. Instead of essentially computing the entire sets $G(p)$ and $G(q)$ to decide $p \equiv q$, a decision procedure should ideally operate on the terms themselves. Previous literature on NetKAT developed decision procedures for the equational theory by leveraging the theory of Kleene algebra and coalgebra. In the next sections, we will demonstrate how the language model of Stateful NetKAT allows this approach to be followed for the portion of the language without the $\&$ operator. The process is in two steps:

1. We define a Kleene algebra called *regularized Stateful NetKAT*. The interpretation of its terms are sets of guarded Stateful NetKAT strings. Every Stateful NetKAT term can be *regularized* as a corresponding term in this Kleene algebra with the same language model interpretation. This translation is similar to but more involved than "reduced NetKAT" in [9], as Stateful NetKAT is not by itself a Kleene algebra with tests.
2. Equations in regularized Stateful NetKAT can be decided with classical tools, as any Kleene algebra can be viewed as expressions over regular sets, in this case our language model. There is then a canonical homomorphism to the final coalgebra corresponding to finite automata, between which we can efficiently decide equivalence [43].

Trivially, each set $G(p)$ is regular, as it can be represented as the sum over all of its elements. So already every Stateful NetKAT term p corresponds to some regular expression such that the sets of guarded strings they represent coincide. This is analogous to computing $[\![p]\!]$ entirely.

However, the insight of regularized Stateful NetKAT is that $p^{(b)}$ can be compactly translated using the Kleene asterate.

In addition, we propose a coalgebraic description for an automaton recognizing guarded Stateful NetKAT strings, thereby taking the first steps towards a decision procedure based on bisimilarity for the full language including $\&$.

4.5.1 Regularized Stateful NetKAT

The reader may have noticed that G behaves like a homomorphism from Stateful NetKAT terms to total subsets of I in the following ways:

$$G\left(p +_b q\right) = G\left(p\right) +_b G\left(q\right) \qquad G\left(p \,\&\, q\right) = G\left(p\right) \blacklozenge G\left(q\right) \qquad G\left(p\,;q\right) = G\left(p\right) \diamond G\left(q\right).$$

Combined with Theorem 69, these hint at an algebraic structure over 2^I. The operators are $+_b$, $\diamond$ and $\blacklozenge$. The sets $G\left(\texttt{false}\right)$ and $G\left(\texttt{true}\right)$ are the zero and one elements respectively. Our aim is to make sense of this structure through the more familiar theory of Kleene algebra.

Definition 74. Given some instantiation of Stateful NetKAT, the corresponding *regularized Stateful NetKAT* is a Kleene algebra

$$\left(\mathsf{At} \cup \Sigma \cup \Pi \cup \left\{\texttt{dup}, \texttt{false}, \texttt{true}\right\}, +, \,;\,, ^\star, \texttt{false}, \texttt{true}\right)$$

that is equipped with an additional binary operator $\&$. As before, At, Σ, and Π are the sets of complete tests, complete state assignments, and complete packet assignments respectively. Note that the $+$ operator here is not restricted by a test condition. Regularized Stateful NetKAT expressions are terms in this language, including $\&$, and with all Kleene algebra axioms imposed.

The interpretation of regularized Stateful NetKAT expressions is a map to sets of regular strings composed from the alphabet $\mathsf{At} \cup \Sigma \cup \Pi \cup \{(\!|\}$ .The strings are in guarded form and match the regular expression $\mathsf{At} \cdot (\!| \cdot \Pi^\star \cdot \Pi)^\star \cdot \Sigma$. This is the same specification as before for the language model, except we have removed $\bot$, and strings of the form $\alpha \cdot \bot$ will not appear. Although the presentation of the language model includes strings $\alpha \cdot \bot$ for consistency with the input-output relations of the packet filtering semantics, this turns out to be extraneous. We can represent computations that will not terminate on input α with the *absence* of any strings guarded by α.

Definition 75. For a set of guarded strings X, we write $X_{\setminus \bot}$ for X but with all strings of the form $\alpha \cdot \bot$ removed. Thus $G(p)_{\setminus \bot}$ represents the possible computations of p that will terminate with a valid output.

We proceed to define the the interpretation $(\!| r |\!)$ of a regularized Stateful NetKAT expression r. It is in fact closely related to the definition of G. For any $r \in \mathsf{At} \cup \Sigma \cup \Pi \cup \{\mathtt{dup}, \mathtt{drop}, \mathtt{false}, \mathtt{true}\}$, it is a valid Stateful NetKAT term, so let $(\!| r |\!) \triangleq G(r)_{\setminus \bot}$. In particular $(\!| \mathtt{false} |\!) \triangleq \emptyset$. For the operators, $+$ is simply set union, while $\&$ and $;$ correspond to the different types of concatenation as defined previously:

$$(\!| r + r' |\!) \triangleq (\!| r |\!) \cup (\!| r' |\!) \qquad (\!| r \,\&\, r' |\!) \triangleq (\!| r |\!) \blacklozenge (\!| r' |\!) \qquad (\!| r \,;\, r' |\!) \triangleq (\!| r |\!) \diamond (\!| r' |\!).$$

To define the * operator, firstly for a set of guarded strings X, let

$$X^0 \triangleq (\!| \mathtt{true} |\!) \qquad X^{i+1} \triangleq X^i \diamond X \qquad X^\star \triangleq \bigcup_{i \geq 0} X^i. \tag{4.12}$$

We can therefore define $(\!| r^\star |\!) \triangleq (\!| r |\!)^\star$.

It is tempting at this point to claim that $(\!| - |\!)$ is a Kleene algebra homomorphism of some sort. This is in fact not the case. As $+$ is required to distribute over $;$, we must have $x \,;\, (y + z) =$

$(x\,;y)+(x\,;z)$. However, consider

$$X \diamond (Y \cup Z) \stackrel{?}{=} (X \diamond Y) \cup (X \diamond Z)\,.$$

This equality may not hold if there is some string in X that requires strings from both Y and Z for the concatenation to be defined, but not Y or Z considered individually. In general, we do not place any restrictions on the regularized Stateful NetKAT interpretations, such as totality in Definition 60. A set modeling an expression can contain multiple strings guarded by some α or none. The set can even be infinite with the use of *. We can recover all of the nicer properties of the Stateful NetKAT language model if we only consider terms in the regularized language translated from valid Stateful NetKAT programs. We call this translation *regularization*, and it is described in the next section.

4.5.2 Regularization

The aim of regularization is to write a Stateful NetKAT expression as a regularized term while preserving the represented set of guarded strings.

Theorem 76. *For every Stateful NetKAT program p, there exists a term $r\,(p)$ in regularized Stateful NetKAT such that $G\,(p)_{\setminus\perp} = (\!|\, r\,(p) \,|\!)$. We call $r\,(p)$ the regularization of p.*

The set of guarded strings $(\!|\, r\,(p) \,|\!)$ is $G\,(p)$ but with all strings of the form $\alpha \cdot \perp$ removed because they are annihilators in any $\blacklozenge$ and $\diamond$ concatenation.

The first obstacle we have to overcome in regularization is that Stateful NetKAT terms involve tests and assignments that are not complete. This can be solved using the following fact.

Lemma 77. *Every Stateful NetKAT term q, besides* `false`*, that is entirely primitive tests and assignments composed with* $;$ *can be rewritten in the form*

$$\hat{q} \triangleq (\pi_1;\sigma_1) +_{\alpha_1} \ldots (\pi_i;\sigma_i) +_{\alpha_i} \cdots +_{\alpha_{k-1}} (\pi_k;\sigma_k)$$

such that $\alpha_i \in \mathsf{At}$, $\pi_i \in \Pi + \mathtt{drop}$, $\sigma_i \in \Sigma$, *and* $G(q) = G(\hat{q})$. *Each* α_i *is distinct but the guarded summation covers all possible cases, i.e., there exists* $\alpha_k \in \mathsf{At}$ *where* $\left(\bigvee_{i=1}^{k} \alpha_i\right) = \mathtt{true}$.

The proof is by induction on q and calculating $G(q)$, whose strings must be of the form $\alpha \cdot \sigma$ or $\alpha \cdot \langle\!\rangle \cdot \pi \cdot \sigma$. The computation of $G(q)$ is straightforward as q is entirely composed of primitive tests and assignments. Each string in the set corresponds straightforwardly to a subterm in $\hat{q}$. We can therefore extend this to eliminate incomplete compositions of tests and assignments in a general term p without changing its semantics. Note that after this transformation, b in $+_b$ and $-^{(b)}$ can still be incomplete, as we only eliminate incomplete tests and assignments appearing as primitive terms. Also, it is not necessary to transform appearances of `true`, even though it can be written as a summation.

The next step is to regularize p to obtain $r(p)$, which is defined homomorphically:

$$r(p) \triangleq p \text{ where } p \in \Sigma \cup \Pi \cup \{\mathtt{dup}, \mathtt{false}, \mathtt{true}\}$$

$$r(p\,;q) \triangleq r(p)\;;r(q) \qquad r(p \,\&\, q) \triangleq r(p) \,\&\, r(q)\,.$$

Now for some possibly incomplete test b, consider

$$B \triangleq +_{\beta \leq b} \beta, \qquad \overline{B} \triangleq +_{\beta \not\leq b} \beta$$

where β ranges over At. Note the iterated $+$ here is the unguarded sum operator in regularized Stateful NetKAT. The empty sum is `false`. For example, if $b = \mathtt{true}$, then $B \triangleq +_{\beta \in \mathsf{At}} \beta$ and

$\overline{B} \triangleq \texttt{false}$. In particular, this is how drop is regularized, as it is just an incomplete test and does not appear directly in the regularized language: $r(\texttt{drop}) = +_{\beta \leq \texttt{drop}} \beta$. More generally, the regularization of conditional and iteration terms dependent on b can be written as the traditional [25] encodings

$$r(p +_b q) \triangleq [B\,; r(p)] + \left[\overline{B}\,; r(q)\right] \qquad r\left(p^{(b)}\right) \triangleq (B\,; p)^\star\,; \overline{B}.$$

4.5.3 Equational Theory of the Regularized Language

Suppose for now we consider only Stateful NetKAT terms without the $\&$ operator. Then the regularization of those terms will also not contain $\&$.

Theorem 78. *For Stateful NetKAT terms p and q without $\&$, $p \equiv q$ if and only if $r(p) \equiv r(q)$.*

Equivalence between $r(p)$ and $r(q)$ in the regularized language (without $\&$) solely arises from the Kleene algebra axioms. To see why this claim is true, recall that there is a homomorphism from Stateful NetKAT terms to the language model $G(-)$, and $G(-)_{\setminus\perp} = (\!|\, r(-) \,|\!)$. As homomorphisms preserve equations, $(\!|\, r(p) \,|\!) = (\!|\, r(q) \,|\!)$. We also need the following fact, despite the previous claim that $(\!|-|\!)$ without restrictions is not a Kleene algebra homomorphism.

Theorem 79. *Consider the subalgebra of regularized Stateful NetKAT generated by regularization. The interpretation $(\!|-|\!)$ on this subalgebra is a homomorphism to the Kleene algebra of sets of regular strings over the alphabet $\mathrm{At} \cup \Sigma \cup \Pi \cup \{\text{Ⓧ}\}$. Concatenation in this latter algebra is $\diamond$, sum is set union, and the asterate is defined as per Equations 4.12.*

Only considering the subalgebra generated by regularization makes $(\!|-|\!)$ well-behaved, as the sets of guarded strings are total. This means concatenation is always well-defined, and sums consist of summands with all the possible guards.

Finally, any Kleene algebra can be viewed as expressions over regular sets. In the present case, regularized Stateful NetKAT terms are regular expressions for our language model, from which there is a canonical homomorphism to the final coalgebra. In practice, deciding the equivalence between regularized Stateful NetKAT expressions then reduces to converting to finite automata and classical methods [43] for efficiently deciding their equivalence.

Unfortunately, the above decision process only works for Stateful NetKAT terms without $\&$. However, this is still useful, as for example almost all the programs given in Section 4.3 are in this form. With $\&$, deciding equivalence is much harder. Taking first steps toward this, in Section 4.5.4 we give a coalgebraic description of an automaton that recognizes guarded Stateful NetKAT strings.

4.5.4 Coalgebra

Now that we can map Stateful NetKAT terms into a Kleene algebra while preserving the language model, we can leverage the rich existing work surrounding decision procedures for KAs. In particular, we follow the approach from [28] to define a coalgebra for Stateful NetKAT.

Definition 80. A *Stateful NeKAT coalgebra* is a triple (S, d, e) consisting of a set of states S, a *continuation map* $d : \mathsf{At} \to (\Pi + \bullet) \to S \to S$, and an *observation map* $e : \mathsf{At} \to \Sigma \to S \to 2$. The co-domain 2 denotes the two-element set $\{0, 1\}$, while $\bullet$ is a distinguished element. We write

$d_{\alpha\pi}$ for the continuation map partially applied with $\alpha \in \mathsf{At}$ and $\pi \in \Pi + \bullet$. Similarly, we write $e_{\alpha\sigma}$ for the observation map with $\sigma \in \Sigma$. It follows that a *Stateful NetKAT automaton* is such a coalgebra with a distinguished start state $s \in S$. The automaton consumes guarded strings from the $\perp$-free subset of the language model as defined in Definition 59, in other words, the model of regularized Stateful NetKAT.

The intuition is that each transition of a Stateful NetKAT automaton consumes a single packet from the start of the guarded string $\alpha \cdot x \cdot \sigma$ and leaves a residual string $\beta \cdot y \cdot \sigma$. In particular:

- The $d_{\alpha\bullet}$ transition attempts to consume any historical packet π from the string prefix $\alpha \cdot \{\!\{ \cdot \pi \cdot \pi'$, and leaves behind $\alpha \cdot \{\!\{ \cdot \pi'$ as the residual prefix. If this succeeds, the automaton lands in the new state determined by $d_{\alpha\bullet}$.
- On the other hand, $d_{\alpha\pi}$ consumes some history-free packet $\pi \in \Pi$ from the prefixes $\alpha \cdot \{\!\{ \cdot \pi \cdot \{\!\{$ or $\alpha \cdot \{\!\{ \cdot \pi \cdot \sigma$, and leaves behind $\beta \cdot \{\!\{$ or $\beta \cdot \sigma$ respectively. In this case $\sigma_\beta = \sigma_\alpha$ and $\pi_\beta = \pi$.
- The observation map $e_{\alpha\sigma}$ determines whether the automaton should accept the string $\alpha \cdot \sigma$ in the current state.

Intuitively, $d_{\alpha\bullet}$ captures the fact that Stateful NetKAT does not directly reason with historical packet data, but packet histories are distinct in the model. On the other hand, $d_{\alpha\pi}$ captures how the state transitions after one packet has been output by the Stateful NetKAT program. As explained previously, we now only deal with guarded strings representing terminating computations. If some guard α causes computation to not halt, strings headed by it will be absent from

the language model. A NetKAT automaton will not accept such strings. Language equivalence between Stateful NetKAT automata would therefore be perfect for our purpose of determining equivalence of Stateful NetKAT terms, as it will ensure equality of the input-output relations.

A Stateful NetKAT coalgebra can be explained as a coalgebra for the set endofunctor

$$F\,X = X^{\mathsf{At}\times(\Pi+\bullet)} \times 2^{\mathsf{At}\times\Sigma}$$

and the continuation and observation maps are the structural map of the coalgebra $(d, e) : X \to F\,X$. For the $\bot$-free guarded strings I, the acceptance predicate $\mathsf{Accept} : S \times I \to 2$ is defined coinductively as below. In all cases β is such that $\sigma_\beta = \sigma_\alpha$ and $\pi_\beta = \pi$.

$$\begin{aligned}
\mathsf{Accept}\,(t, \alpha\cdot \langle\!\langle \cdot\pi\cdot \langle\!\langle \cdot x) &\triangleq \mathsf{Accept}\,(d_{\alpha\pi}\,(t)\,, \beta\cdot \langle\!\langle \cdot x)\\
\mathsf{Accept}\,(t, \alpha\cdot \langle\!\langle \cdot\pi\cdot\sigma) &\triangleq \mathsf{Accept}\,(d_{\alpha\pi}\,(t)\,, \beta\cdot\sigma)\\
\mathsf{Accept}\,(t, \alpha\cdot \langle\!\langle \cdot\pi\cdot\pi'\cdot x) &\triangleq \mathsf{Accept}\,(d_{\alpha\bullet}\,(t)\,, \alpha\cdot \langle\!\langle \cdot\pi'\cdot x)\\
\mathsf{Accept}\,(t, \alpha\cdot\sigma) &\triangleq e_{\alpha\sigma}\,(t)\,.
\end{aligned}$$

In future work we hope to explore decision procedures for the equivalence of Stateful NetKAT automata based on bisimilarity. In particular, we could use derivatives [10, 19] to define d and e from Stateful NetKAT terms.

4.5.5 Axiomatization

Another approach towards finding a decision procedure for Stateful NetKAT is to give a consistent and complete axiomatization for the language. The axioms are in fact a superset of the

commutativity conditions D in our coproduct $P \oplus Q$ modulo D, as they set out terms that are equivalent. In this section we give a set of consistent axioms for Stateful NetKAT. Similar to the semantics, it is convenient to consider each fragment of the language in turn.

4.5.5.1 Stateless Axioms

As the stateless part of Stateful NetKAT is based on standard NetKAT, naturally all axioms from the latter should apply. The most interesting extra operator we introduce is $\&$, and we can give it the following axioms:

$$\texttt{drop} \mathbin{\&} p \equiv p \mathbin{\&} \texttt{drop} \equiv p \qquad (p \mathbin{\&} q) +_b (r \mathbin{\&} t) \equiv (p +_b r) \mathbin{\&} (q +_b t)$$

$$p \equiv p' \implies p \mathbin{\&} q \equiv p' \mathbin{\&} q \qquad q \equiv q' \implies p \mathbin{\&} q \equiv p \mathbin{\&} q'$$

$$p\,;(q \mathbin{\&} r) \equiv (p\,;q) \mathbin{\&} (p\,;r) \qquad (p \mathbin{\&} q)\,;r \equiv (p\,;r) \mathbin{\&} (q\,;r)$$

In addition, there are axioms relating to $+_b$ arising from the fact the tests form a Boolean algebra, such as $p +_b q \equiv q +_{\overline{b}} p$.

4.5.5.2 Pure State Axioms

Recall that for the pure state language, the interpretations are partial functions $S \rightharpoonup S$, or alternatively functional matrices. There is also an ordering on interpretations $f \leq g$ defined in

Equation 4.3. This means that, among other axioms, we have

$$bf \leq f +_b g \qquad \overline{b}g \leq f +_b g \qquad bf \leq h \wedge \overline{b}g \leq h \implies f +_b g \leq h$$

$$(f +_b g)h = fh +_b gh \qquad fc(g +_b h) = \begin{cases} fcg, & \text{if } c \text{ is an atom and } c \leq b \\ fch, & \text{if } c \text{ is an atom and } c \leq \overline{b} \end{cases}$$

$$ff^{(b)} +_b \mathtt{true} \leq f^{(b)} \qquad fh +_b \mathtt{true} \leq h \implies f^{(b)} \leq h.$$

As usual for a Boolean algebra, c is an atom if it is the least non-false element: $b \vee c = c$ if and only if $b = \mathtt{false}$ or $b = c$.

Also, for constants m, n and constants or variables u, v,

$$A.x \leftarrow u\,; B.y \leftarrow v \equiv B.y \leftarrow v\,; A.x \leftarrow u,\ A.x \neq B.y, u \neq B.y, v \neq A.x$$

$$A.x \leftarrow u\,; A.x \leftarrow v \equiv A.x \leftarrow v,\ v \neq A.x$$

$$A.x \leftarrow u\,; A.x \leftarrow A.x \equiv A.x \leftarrow u$$

$$A.x \leftarrow u \equiv A.x \leftarrow u\,; A.x = u$$

$$A.x = u \equiv A.x = u\,; A.x \leftarrow u$$

$$A.x \leftarrow u\,; B.y = v \equiv B.y = v\,; A.x \leftarrow u,\ A.x \neq B.y, v \neq A.x$$

$$A.x = m\,; A.x = n \equiv \mathtt{false},\ m \neq n$$

$$\sum_n A.x = n \equiv \mathtt{true}.$$

4.5.5.3 Full Language

The least we could say about the axioms for full Stateful NetKAT is that the compound tests and assignments from the two halves of the language commute. That is to say, suppose p and q are stateless and pure state programs respectively that are solely tests or solely assignments composed with ;. It is true that $p\,;q \equiv q\,;p$. With the addition of other operators however, the situation quickly becomes unclear. We have already seen examples in Section 4.2.2.3 where the composition of stateless and pure state programs creates subtle interplay in the semantics. The operator $\&$, which originates from the stateless language, is especially troublesome when combined with state programs. For example, consider

$$(A.x \leftarrow 1\,;A.y \leftarrow 2)\ \&\ A.z \leftarrow 3 \equiv A.x \leftarrow 1\ \&\ (A.y \leftarrow 2\,;A.z \leftarrow 3)\,,$$

yet neither the left nor the right branches of $\&$ are separately equivalent. In the full language $\&$ also does not have clear distributivity properties similar to those in the stateless axioms. While it is straightforward to see all the axioms we have presented so far are consistent in Stateful NetKAT, we leave the task of finding a complete deductive system to future work.

Readers familiar with the work done by [61] may realize it is possible to express Stateful NetKAT as a guarded Kleene algebra with tests (GKAT) with an additional $\&$ operator. In fact, the regularization process described in previous sections is similar to how GKATs embed into the theory of Kleene algebra. The complete axiomatic system for GKAT may provide guidance in finding a full set of axioms for our work.

4.6 Future and Related Work

The mixture of stateless, packet data reasoning with mutable state creates subtle interplays that we hoped to capture within a single language. We have made many decisions in the combined semantics of Stateful NetKAT that aim to be as consistent as possible with the stateless and pure state fragments of the language while preserving generality. There is much further work that can be done in perfecting the metatheory of the language, including as mentioned the exploration of a coalgebraic decision procedure and axiomatization. Besides the examples we have given, it would be interesting to see if Stateful NetKAT can be implemented and used in practice to specify real-life packet forwarding programs.

There have been many extensions to NetKAT since its presentation. A recent, fruitful line of work has been the addition of probabilistic choice to the language [27, 62, 65], and it remains to be seen if mutable state can work alongside. In past work, reasoning about mutable state in switches has been attempted by work such as [51, 46, 11]. Formal semantics in this area tend to be operational with a view towards implementation and compilation. Our work is based on a principled foundation with a denotational semantics that is able to reason about program equivalence, in order to better inform compilation and verification. In particular, adding mutable state to NetKAT has been considered by [46], although that was to facilitate the specific application of network event structures and not unrestricted, general purpose state. It was also the approach taken by [11]. Both of these extensions retained the inherently parallel and non-deterministic $+$ operator from NetKAT, and as a result they were restricted in how they resolved conflicting changes in state. Stateful NetKAT takes the contrasting approach of assuming packet processing can always be serialized into a deterministic, global order. A sound theoretical foundation involv-

ing non-deterministic and parallel processing could draw ideas from work done by [34, 35, 33] on concurrent Kleene algebra. Finally, we also consulted the work by [14] on Kleene Algebra Modulo Theories, which considered state in a limited sense along other possible client algebraic theories that can be combined with Kleene algebra.

CHAPTER 5

FURTHER APPLICATIONS

There is an ongoing interest in the network community for incorporating more programmability into the dataplane. Besides hardware switches that may implement match-action, one particular area of interest is to improve methods for programming software switches, such as Open vSwitch (OVS) [53]. The main use case of a software switch is in hypervisors, software that creates and manages virtual machines and provides virtualization to emulate the appropriate machine platform. In software switches, network traffic is routed to and from virtual machines that the hypervisors manage. To program these switches, protocol developers write network programs in domain-specific languages (DSLs) such as P4 [17, 50], specifically designed to easily express packet processing logic. A target-specific compiler generates the final instructions executed by the software switch using the input program. PISCES [58], a P4 programmable version of OVS, is a good example of a switch that adopts this model of programmability.

However, we argue that if a compiler has fine-grained control over program instructions in the target software switch than what is available today, performance benefits can be gained for network programs. While existing software switches are designed and hand-tuned to support a particular set of features, they expose only a limited set of interfaces (e.g., OpenFlow [15]) for external entities (e.g., controllers) to modify or fine-tune their functionality. This corresponds to the fact that, given their nature, DSLs are highly expressive in specifying packet forwarding logic but expressiveness is limited for fine-tuning the target-specific parameters of the architecture. Thus, it is very difficult for compilers to specify low-level behaviors, such as defining memory and CPU cache access patterns or instruction parallelization behaviors in the software switch.

The goal of this chapter is to make the case that improvements in performance, flexibility, and feature support are attainable if the target switch exposes more low-level interfaces. We offer the following contributions towards substantiating this claim:

- Review of the Vector Packet Processor (VPP) [67, 12], a software switch that expose low-level interfaces. We explain how VPP is able to obtain better performance against competing switches, such as OVS, with access to the low-level architecture (Section 5.1).
- The design and implementation of Programmable Vector Packet Processor (PVPP), a software switch built using the VPP framework that can be programmed with P4_{14}, the first version of the P4 DSL [17, 50] (Section 5.2).
- Compiler optimizations of PVPP exploiting the low-level interfaces that VPP exposes (Section 5.3).
- An evaluation demonstrating the increased performance of PVPP programs as more optimizations are made. We also present preliminary results comparing the performance of PVPP against the existing packet-forwarding features in VPP [67] and PISCES [58] (Section 5.4).

5.1 VPP: A Flexible Software Switch Target

We begin by giving an overview of the Vector Packet Processing (VPP) framework, the software switch that PVPP is built upon. The VPP platform [67, 12] is an extensible packet processing framework that provides a fully-featured, highly-optimized forwarding engine designed to run

on general purpose, commodity CPUs. The VPP platform runs completely in user-space by leveraging Data Plane Development Kit (DPDK) device drivers [3]. DPDK is a set of libraries that allows *kernel bypass*, the offloading of packet processing to user-space programs instead of being handled in the kernel as it was traditionally. This enables greater control and performance as the packet processing functionality no longer has to be introduced as a kernel module, for the most part bypassing kernel code.

In VPP's vector processing model, the forwarding path is decomposed into a collection of processing nodes that are organized as a directed graph. Each node has the responsibility of processing an entire vector of packets, making local modifications and deciding the next node for every packet in the vector. Vectors of packets are formed at the input nodes of the graph by processing as many available packets from the receiving network interface driver as possible. The main reason for having each node work on a vector of packets at a time is to optimize instruction and data cache locality across the entire vector, minimizing the number of expensive cache misses that may occur. In addition, as each node iterates through a vector, the node will prefetch the data for the next packet in the vector, thereby hiding the memory latency associated with data-read dependencies.

The ability to split packet processing into a graph of nodes consequently enables pipelined execution for VPP. On multi-core systems, processing performance can improve by allowing multiple threads to exercise the same node in the graph. Alternatively, the graph can split across CPU cores, forming a threaded-pipeline model.

An implementation of a network program in VPP consists of one or more nodes that process vectors of packets, where each node contains the code for a logical stage in the program. Since

there is no boundary restriction on logic contained in each node, there is a high flexibility in deciding how a program is divided into VPP nodes. Each implementation can choose a division that optimizes some parameter, such as cache locality.

The current VPP codebase contains a set of highly efficient hand-tuned implementations of common network protocols and features. These correspond to a collection of packet processing nodes that are loaded by default as the standard VPP installation starts up. In these implementations, a distinctive pattern, typical to most nodes, is the unrolling of instructions to process multiple packets in a single iteration of the vector-traversal loop. Packets processed in the same iteration are operated on near-simultaneously by interleaving the instructions relevant to each packet before moving on to the next instruction. This design is to maximize the utilization of CPU registers, since each packet is only likely to require the use of some of the registers available on the processor. Memory locality and instruction cache hits are also improved, as processing multiple packets in parallel are very likely to perform the same set of instructions and refer to similar memory locations.

VPP allows for modularity and extensibility in the packet processing pipeline through the use of *plugins*, which let users introduce new nodes with custom packet processing logic. Further, a plugin has the full freedom to rearrange or delete existing nodes in the graph. A plugin is even able to replace nodes responsible for packet input and output from the device, thereby potentially supplanting the entire processing graph with nodes introduced by the plugin. VPP plugins are typically developed and compiled separately from each other and from the core VPP code.

An example of VPP graph structure with is shown in Figure 5.1. The `dpdk-input` and `dpdk-output` nodes handle passing packets to and from the DPDK library. The packets may

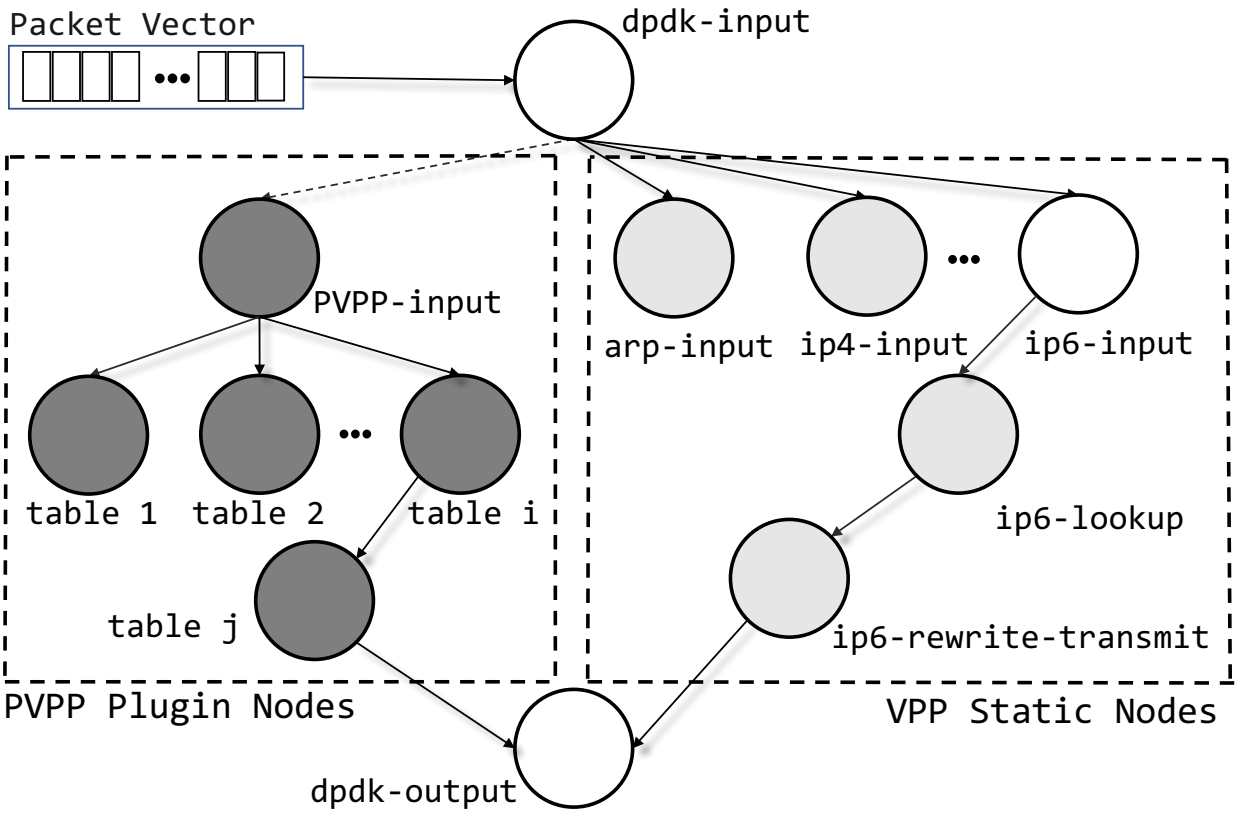

Figure 5.1: VPP node graph structure with PVPP plugin.

proceed to the static nodes on the right, consisting of a set of hand-coded processes for network protocols that VPP already supports. Alternatively, they could be processed by the plugin nodes on the left added by PVPP for functionality compiled from P4.

VPP has demonstrated outstanding performance with its existing network implementations. In particular, it has achieved line-rate performance when running common forwarding programs on commodity x86 CPUs [69]. It also attains better performance compared to other software switches such as OVS [68]. The efficiencies that make these results possible are obtained by manipulating the underlying low-level architecture in various ways, such as processing multiple

packets simultaneously in a single packet vector iteration. However, a disadvantage of operating on packets at low-level is that it is difficult to produce correct implementations of packet forwarding logic. A programmer who wishes to produce efficient implementations in VPP must possess specialized knowledge in networks, operating systems, memory management and compiler optimization techniques, among others. The hand-coded implementations that currently exist in the VPP code base are difficult to evolve as the underlying processor evolves. An increase in the number of registers available, for example, could potentially improve performance, but existing code that is hand-tuned for a set number of registers must also be retrofitted manually in order to take advantage of this change.

Ideally, we would like to automatically engage the low-level interfaces available in VPP through the use of a compiler. This would show that it is possible for a software switch to expose its underlying architecture, bringing performance and other benefits, while not requiring the programmer to possess the knowledge need for an efficient implementation. To achieve this goal, we present the state of our ongoing work on PVPP.

5.2 PVPP Architecture

We now discuss PVPP's architecture in detail, which works with the P4_{14} version of the P4 language. We first discuss methods for embedding P4 functionality as a VPP plugin. Then, we discuss the design choices and optimizations implemented in a P4-to-PVPP compiler that generates the plugin.

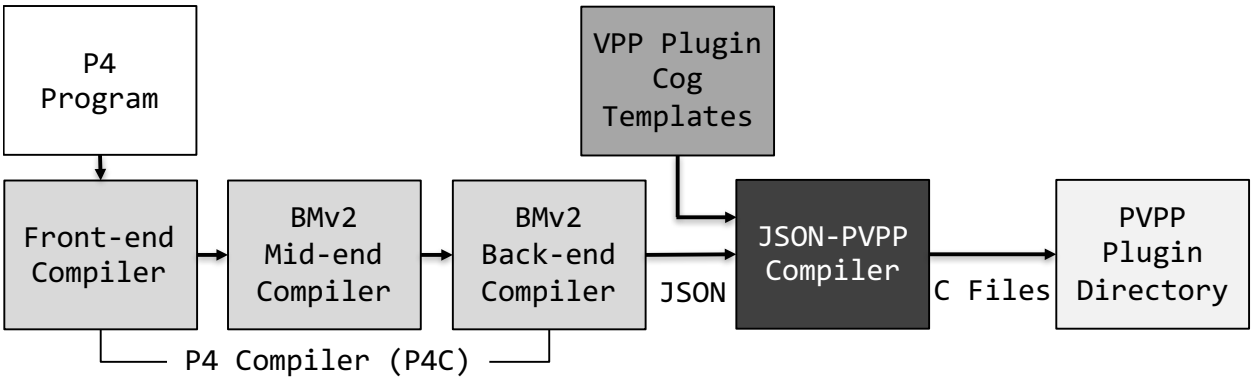

Figure 5.2: PVPP compiler architecture.
The architecture consists of two parts: P4C and JSON to PVPP.

5.2.1 Separate Compilation via VPP Plugin

PVPP integrates an input P4 program into VPP as a plugin. Embedding the entire P4-related logic into a plugin has many advantages. One major advantage is that compiling a new P4 program as a different VPP plugin does not require recompilation of the whole VPP framework, as plugins are designed to compile and load separately. This is a clear advantage over the approach taken by PISCES [58]. PISCES requires recompilation of the entire OVS codebase for each new P4 program, since the code generated by the P4-to-OVS compiler is linked statically and compiled with the whole OVS baseline source code. Another advantage is in the distribution of compiled code to multiple target systems. A plugin is compiled only once before it is distributed to different VPP targets, even if those targets are running different versions of VPP or configurations.

5.2.2 P4-to-PVPP Compiler

The compilation of P4-to-PVPP code is a two-step process. The first step runs the open-source P4 compiler (P4C) [6], which takes the input P4 program and generates an intermediate representation in JavaScript Object Notation (JSON) format. The overall structure of P4C is shown in the left half of Figure 5.2. In the second step, we feed this intermediate representation to our JSON-to-VPP compiler to generate a customized VPP plugin with single-node or multiple-node implementations. The single-node implementation creates a plugin with only one node that performs the entire parse, match, and action logic defined in the input P4 program. In contrast, the multiple-node implementation creates a plugin with a set of nodes that are each responsible for a subset of the parse, match, and action logic. In our initial multiple-node implementation, PVPP creates a separate node corresponding to each P4 table, where the node performs match and actions for that table. The overall process of compiling JSON to PVPP is shown in the right half of Figure 5.2.

Stage 1: P4 to JSON. P4C first transforms the input P4 program into an internal intermediate representation [7] by running it through the front-end compiler of P4C. The front-end compiler is target-independent and is mainly responsible for P4 program validation, type checking, and performing target-independent optimizations. Given the intermediate output, P4C runs it through the target-dependent mid-end and back-end to perform optimizations driven by target-dependent policies and generates an optimized final representation. This final representation can be in various forms, such as code snippets or data objects; in PVPP's case, JSON output is generated.

Stage 2: JSON to PVPP. Given the JSON output from P4C, the custom Python-based compiler generates all the necessary C code to build a VPP plugin. Specifically, the JSON representation is input into the Cog [2] template engine. Cog is a code generator that is able to produce completed output text from templates by executing Python code annotated within placeholder sections. The output is merged with other static files to build a VPP plugin.

5.3 PVPP Compiler Optimizations

We discuss the compiler optimizations that exploit the low-level details that are accessible in VPP plugins.

Reducing metadata access and pointer dereferences. One of the most important compiler optimizations is to implement an efficient memory management scheme. To do so, the compiled code is designed so that memory dereferences are eliminated whenever possible. One place where this is evident is in the storage of metadata. A good approach we found to optimize metadata storage is to flatten and merge all P4 metadata structures into a single C structure. Although this is wasteful in space, as not all types of metadata will be valid for every possible path of the packet processing pipeline, we found significant performance improvements compared to handling metadata at multiple locations that a more direct translation of P4 declarations would produce.

Similarly, PVPP represents all action-table relationships as pre-computed pointers. Because PVPP determines the next table by the action taken according to the matched rule of the current table, it would know in advance the pointers to tables and actions at compile time. Therefore, we

can store the relevant pointers in a rule at the time it is inserted into a table, reducing the amount of pointer arithmetic needed during rule matching stage.

Reducing number of tables and metadata. Naïvely, P4C produces a final JSON output containing a number of redundant tables and metadata. For example, P4C generates a redundant table with no match rule for performing interface output selection. Such redundant tables result in extraneous loops of table- and action-lookups for our single-node implementation, and they result in iterations through the extraneous set of nodes in the multiple-node implementation. Thus, we add an optimization to remove redundant tables and metadata that do not serve any VPP-specific function in the JSON output prior to feeding the output to the final stage of the compiler.

Multiple-node pipeline implementation. The most unique characteristic of the VPP framework is that packets traverse through a graph of processing nodes. Unfortunately, finding the best division of tasks in a packet processing pipeline into a set of nodes in a directed graph is non-trivial. We add an optimization to allow users to annotate the P4 program to specify whether they want to create a plugin with a single node or multiple nodes. For a match-action pipeline defined by a P4 program, the obvious division we implement is to separate each table into different nodes. However, evaluation shows that the multiple-node implementation performs slightly worse than the single-node implementation for our benchmark programs. We discuss the reasons behind this performance overhead and other methods for optimizing the division of tasks in the next section.

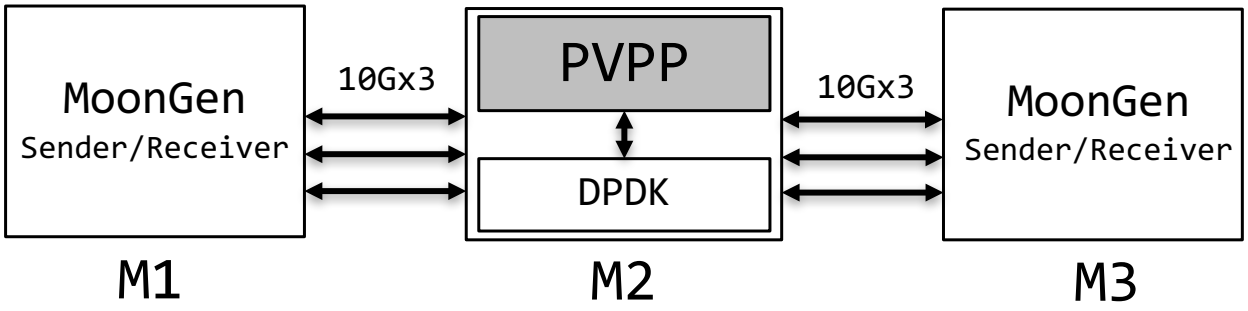

Figure 5.3: Topology of the PVPP experimental setup.

5.4 Evaluation

We now discuss preliminary performance and scalability results using our benchmark application. Then, we discuss the comparison results between PVPP and PISCES.

5.4.1 Experimental Setup

Figure 5.3 shows the topology of our setup for evaluating the forwarding performance of PVPP. We used three PowerEdge R730xd servers with two 8-core, 16-thread Intel Xeon E5-2640 v3 2.6GHz CPUs running the Proxmox Virtual Environment Kernel version 4.2.6-1-pve [54], an open-source server virtualization platform that uses virtual switches to connect VMs. These machines were equipped with one dual-port and one quad-port Intel X710 10 Gbps network interface card (NIC). We configured two of these machines, namely `M1` and `M3`, with MoonGen [24], for sending and receiving 64-byte packets, respectively, at 14.88 million packets per second (Mpps). We connected these six interfaces to a third machine, `M2`, running PVPP, causing `M2` to process a maximum of 60 Gbps of traffic.

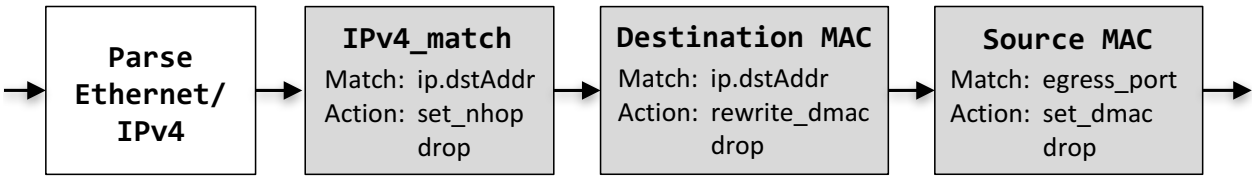

Figure 5.4: Control flow of the PVPP benchmark application.
The white box corresponds to parsers and grey boxes correspond to tables.

Optimization	**Single** Mpps	**Single** Increment (%)	**Multiple** Mpps	**Multiple** Increment (%)
Unoptimized	7.860	N/A	7.051	N/A
Removing Redundant Tables	9.248	+1.388 (+17.7%)	8.381	+1.330 (+18.9%)
Reducing Metadata Access	9.508	+0.260 (+2.81%)	8.501	+0.120 (+1.43%)
Multiple Packet Processing	9.508	+0.000 (+0.00%)	8.800	+0.299 (+3.52%)
Reducing Pointer Dereferences	10.008	+0.500 (+5.26%)	9.023	+0.223 (+2.53%)
Caching Interface Mapping	10.209	+0.201 (+2.01%)	9.197	+0.174 (+1.93%)

Table 5.1: Incremental improvements of each optimizations for PVPP, single- vs. multiple-node.

5.4.2 Baseline End-to-End Performance

We first measured the throughput of our benchmark application to establish the baseline performance with no optimization. The overview of the application is shown in Figure 5.4. In this application, the packets received at ingress are processed through PVPP's parser, allocating memory addresses for the IP and Ethernet headers. Then, the `IPv4_match` table matches on the destination IP address of the packet, performs a TTL decrement, and writes the next hop IP address and the egress port based on the installed rules. The packet then goes through two separate tables rewriting the destination and source MAC address. Finally, the packet is sent out to the interface connected to `M3`. The first line of Table 5.1 shows the baseline throughput for the given benchmark application.

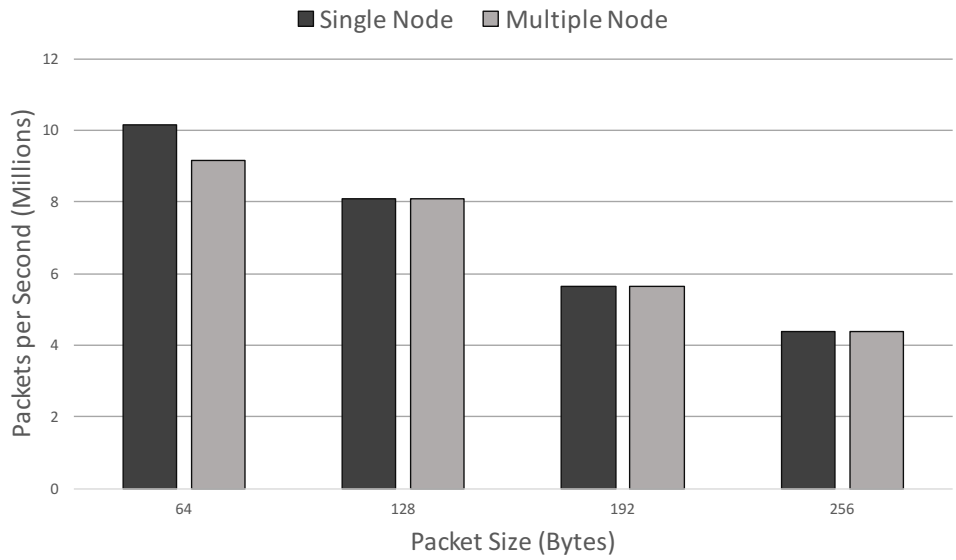

Figure 5.5: Forwarding performance of PVPP for the benchmark application across one 10Gbps interface, in Mpps.

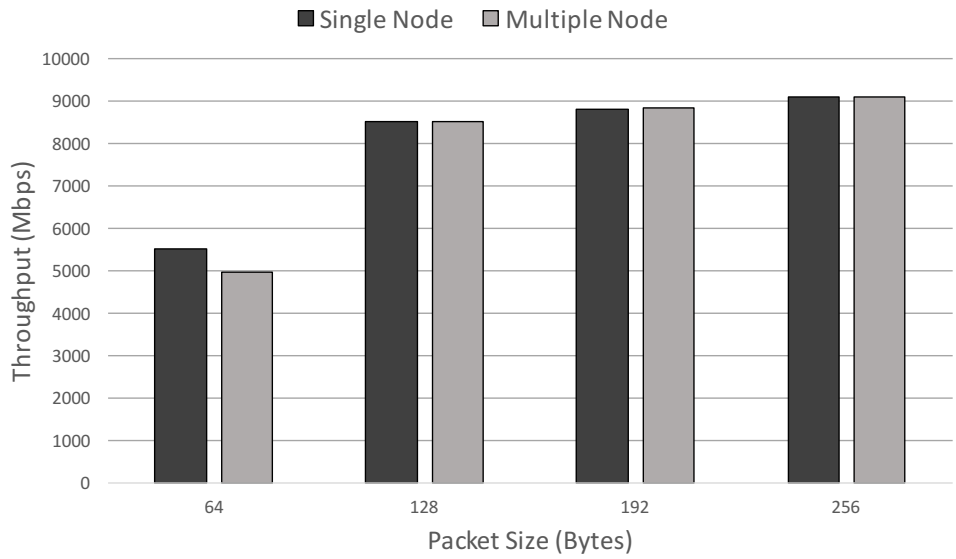

Figure 5.6: Forwarding performance of PVPP for the benchmark application across one 10Gbps interface, the same benchmark as previously but in Mbps.

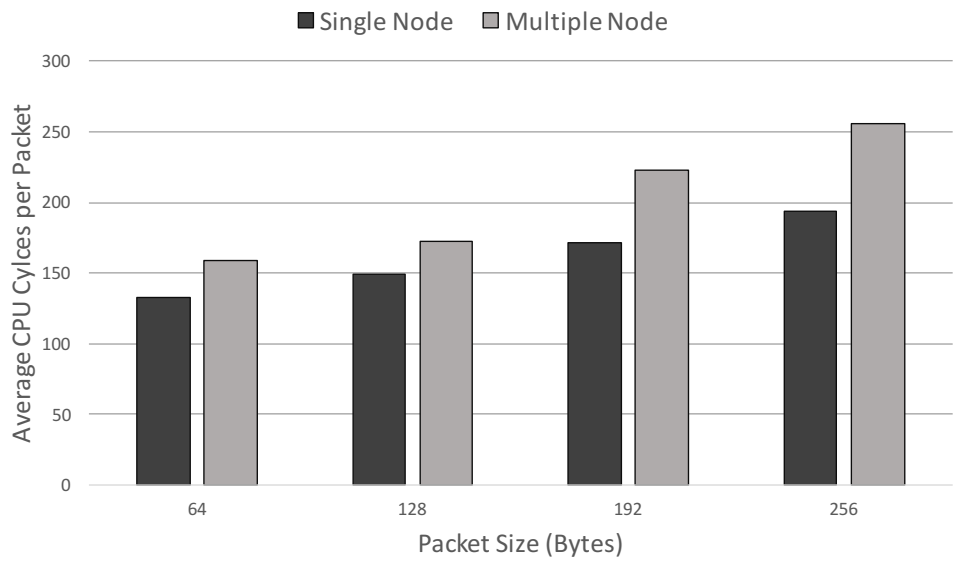

Figure 5.7: Number of CPU cycles consumed per packet during PVPP benchmark.

5.4.3 Optimized End-to-End Performance

We now show the effects of the optimizations that we discussed in Section 5.3. The measurements were obtained from running the same benchmark application described in Section 5.4.2. Table 5.1 shows the incremental improvement in throughput for each of the optimizations. We saw the greatest improvements when reducing the number of match-action tables, which has a similar effect of removing a processing node in the multiple-node implementation. Reducing the number of pointer dereferences provided further improvements, since pointer dereferencing requires a large number of CPU cycles.

One interesting observation to emphasize is the multiple packet-processing optimization for the single-node implementation. Recall that in Section 5.1, we observed that VPP nodes often processed multiple packets in parallel within a single vector traversal loop iteration. We im-

plemented the same optimization, such that the compiler was able to generate a target network program that automatically unrolls the loop and processes two packets during a single iteration. Interestingly, this optimization in the single-node implementation did not show any improvement. We hypothesize that this was the result of a long series of operations that a packet must go through when the entire pipeline is fit into one node, thus requiring a large number of registers for processing each packet. Also, the single-node implementation cannot fully reap the benefits of memory locality or instruction cache hits because the packets have to be processed through the entire pipeline—straining the cache—before processing the next packet.

With all the optimizations, PVPP's throughputwas 10.209 Mpps for the single-node implementation and 9.197 Mpps for the multiple-node implementation. The throughput of vanilla VPP for the equivalent application was 10.748 Mpps, which was 5.25% higher than the single-node implementation and 16.9% higher than the multiple-node implementation.

Figures 5.5, 5.6, and 5.7 show the detailed end-to-end performance of optimized PVPP for our benchmark application. Note that the measurement for CPU cycles only included the cycles that were spent on the packet processing and did not include the cycles spent for DPDK interactions and packet output. The experiment showed an overhead of about 10% for both the Mpps and Mbps measurements for a 64-byte packet and about 30% overhead for the number of CPU cycles spent across all the packet sizes for the multiple-node implementation versus the single-node implementation. This overhead was mainly the result of additional VPP-related operations for moving packets between different nodes.

Another interesting observation was the throughput versus CPU cycles. The measurements showed that the increase in the CPU cycles spent per packet with the larger packets are greater

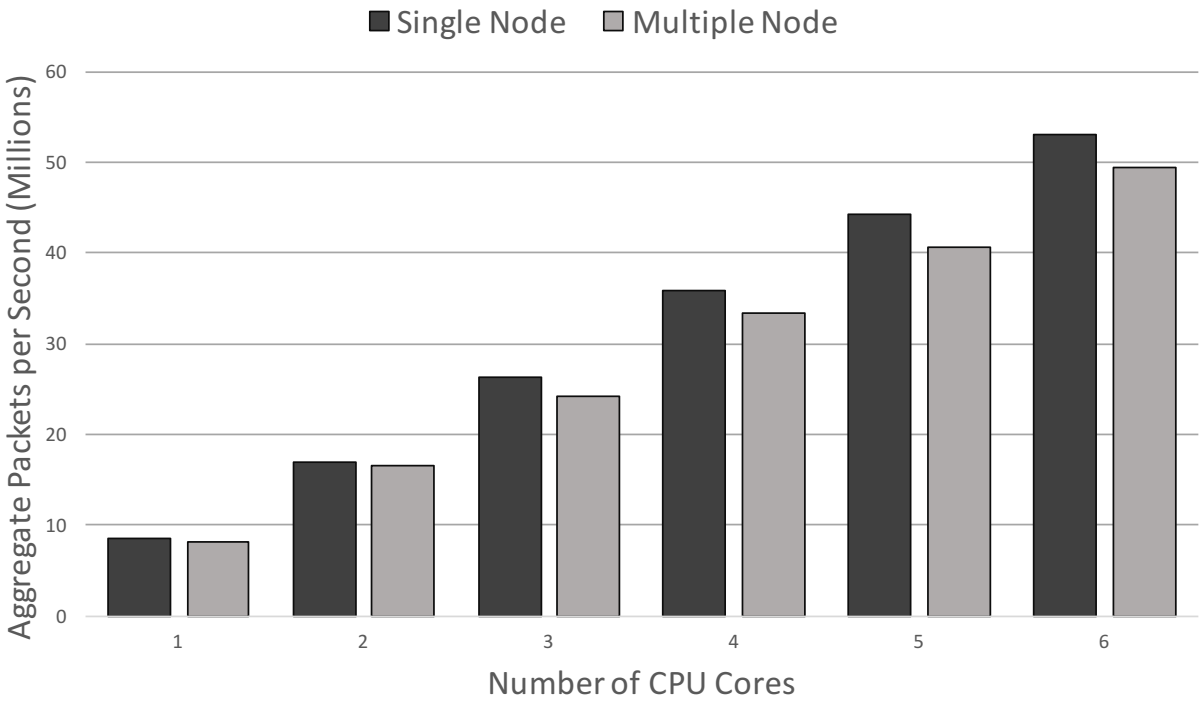

Figure 5.8: Effect of the number of cores used for PVPP on throughput (Mbps). Effect on Mpps is the same. Mpps = Mbps/(8 $\times$ 64).

for the multiple-node implementation versus the single-node implementation. However, the throughput measurements of the two implementations counter-intuitively showed a comparable increase. This result was mainly due to the fact that PVPP becomes I/O-bound rather than CPU-bound for larger packet sizes, thus the increase in CPU cycles spent was not reflected in the throughput measurements.

CPU Cycles	PVPP	PISCES (with microflow)	PISCES (no microflow)
End-to-End	132.9	100.6	166.0

Table 5.2: Average number of CPU cycles consumed for processing a 64-byte packet in PVPP and PISCES with and without microflow cache.

5.4.4 Multi-Core Performance

We also measured the throughput of our benchmark application while varying the number of CPU cores and the number of input interfaces used by PVPP to quantify the scalability of throughput versus the number of CPUs used. Only one CPU processed the packet from one designated interface.

For this experiment, we started bidirectional traffic between the three interface pairs, generating 64-byte packets at the maximum rate of 60 Gbps. This resulted in packets flowing through six ingress ports that PVPP must process. The procedure of adding an interface accompanies launching another VPP thread on a new CPU core that processed the packets arriving at the newly added ingress interface. Figure 5.8 shows the relationship between the performance of PVPP with a varying number of CPU cores. We observed that throughput increases linearly with the addition of CPU cores as expected.

5.4.5 Comparing PVPP and PISCES

We compare the forwarding performance between PVPP and PISCES [58], with and without micro-flow cache turned on. The PISCES underlying switch target OVS relies on caches to achieve

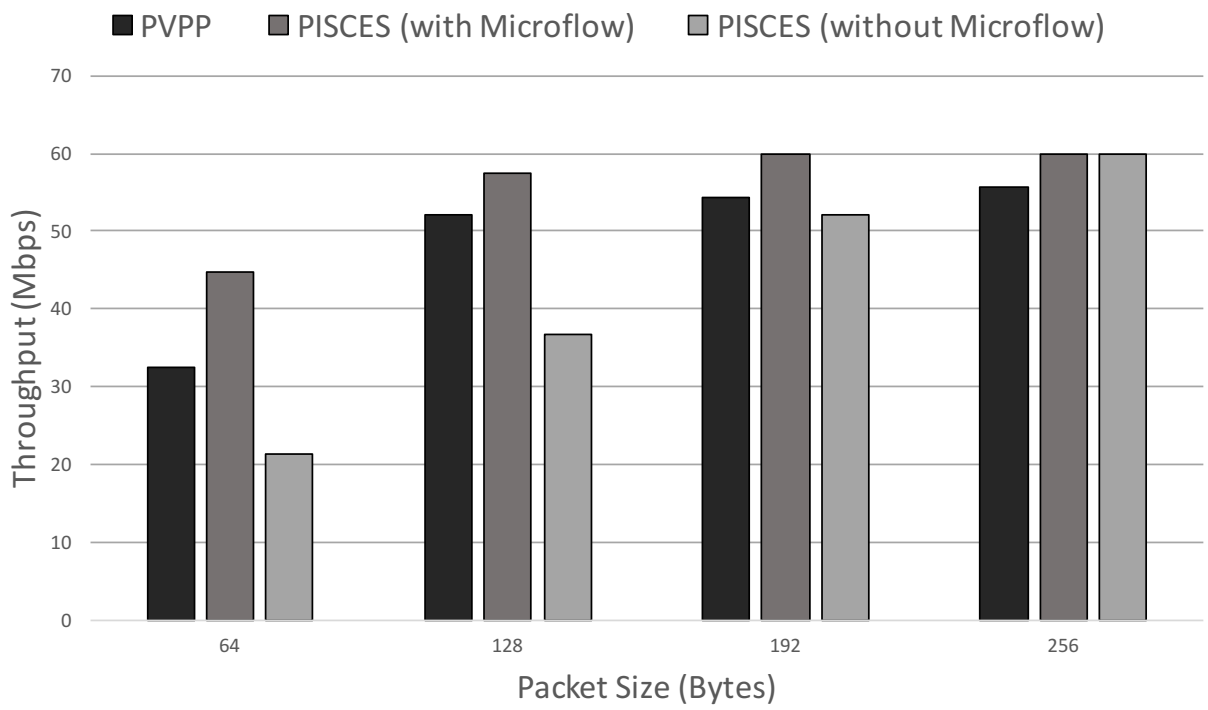

Figure 5.9: Throughput comparison between the single-node implementation of PVPP and PISCES with and without microflow cache.

good forwarding performance. The primary OVS cache is its mega-flow cache, which is a feature not present in PVPP-compiled code. This is where OVS combines the results of tables that a packet visits in the match-action pipeline into a single flow rule by (lazily) computing the cross-product of the tables, caching the result for fast recall later [53]. Each result of the cross-product corresponds to one rule in the mega-flow cache. The mega-flow cache wildcard matches on headers fields and hence it can still have a significant toll on performance. Thus, OVS also includes a micro-flow cache, an exact-match cache, which is a map from a packet's IP five-tuple to a mega-flow cache entry. The five-tuple consists of the the source and destination addresses and port numbers, in addition to an identifier for the transport layer protocol, and therefore it identifies

the network flow that the packet belongs.

For comparison, we used a simple (switch) program that matches on the destination MAC and sends the packet to the appropriate egress interface. We send various sized packets to all six interfaces using six CPU cores, each pinned to a distinct ingress interface, resulting in a total of 60 Gbps traffic.

Figure 5.9 shows the comparisons. PVPP, with a small overhead, performs comparably with PISCES with micro-flow cache enabled. PVPP performs better than PISCES with micro-flow cache disabled. Table 5.2 shows the comparisons of end-to-end CPU cycles with different cache configurations. A caveat to note is that this is not an exact comparison because of the difference in implementations of the two switches. Nevertheless, CPU cycles spent per packet are comparable, confirming the similarities in throughputs.

5.5 Discussion and Future Work

PVPP presents a promising argument that low-level interfaces available in a software switch can be utilized to generate a more efficient P4-programmed switch. In order to further strengthen the argument and to provide a more functional P4 programmable software switch, we are planning to complement our work with the following action items.

Automated and optimal node splits. The current scheme for generating multiple nodes for a PVPP plugin is to let the compiler split a P4 program by creating a node for each P4 table. Our

evaluation shows that this method can lead to worse performance. As part of our future work, we intend to create schemes using analysis of input P4 programs to generate more intelligent node splits for better performance.

Handling multiple packets. As mentioned in Sections 5.1 and 5.4, processing two packets per loop iteration is often beneficial. We believe that further unrolling may yield better performance, particularly for match-action tables with a small number of instructions. PVPP provides the opportunity to explore this space by developing a P4 program analyzer that determines the optimal number of packets to process in parallel.

Extending P4 feature support. PVPP currently lacks some features that P4 supports such as dataplane states (i.e. registers, counters or meters). One advantage of VPP is the lack of OVS-like cache structures that enables a relatively straightforward implementation of the dataplane states. One possible design suggestion to implement dataplane states in PVPP is to compile all match-action tables that refer to the stateful features into a single node, with the state allocated and referenced from only within that node. This approach avoids adding costly locks that are needed if more than one node references the same state. However, supporting parallel accesses on the same states across multiple nodes requires a more sophisticated design to ensure correct stateful operations.

CHAPTER 6

CONCLUSION

In this dissertation, we have presented different approaches to formalizing the match-action paradigm. We have also introduced an extension of the high-level network specification language NetKAT to model state in the data plane. Finally, we have presented a high-level description of the VPP software switch and the results of an analysis of the performance of a compiled P4 program to that platform.

We believe our results obtained from formalization bear theoretical interest. In addition, if they are taken further, they could lead to the discovery of new compiler optimizations targeting match-action. However, applying theoretical ideas to understand current SDN architectures is not limited to this one paradigm, nor is it limited to the methods we have presented in the past chapters. For example, we could attempt to verify how much each of the claimed sources of high performance in VPP [12] actually contributes to throughput. There still remains rigorous micro-benchmarking that can be done, aided by theoretical knowledge of the relevant hardware systems.

For Stateful NetKAT, there is also a lot more interesting work that can be done. Besides completing the metatheory efforts set out in the Chapter 4, the language can be implemented and programs tested in practice. Another idea is to allow an unguarded $+$ operator for non-deterministic behavior. However, the metatheory is expected to be very difficult in that case, as we must keep track of a set of non-deterministic outcomes for each cell of the input/output string of packets. Perhaps again the recent work on concurrent flavors of KATs [34, 35, 33] can help

here. Finally, we sketched a promising start towards a complete axiomatization and coalgebraic decision procedure that would be able to decide the equational theory of the full Stateful NetKAT language.

BIBLIOGRAPHY

[1] Cisco Catalyst 9000 Wireless and Switching Family. https://www.cisco.com/c/en/us/solutions/enterprise-networks/catalyst-9000.html.

[2] Cog. http://nedbatchelder.com/code/cog/.

[3] DPDK: Data Plane Development Kit. http://dpdk.org.

[4] Intel Tofino Series Programmable Ethernet Switch ASIC. https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-series/tofino.html.

[5] Juniper Networks EX Series Ethernet Switches. https://www.juniper.net/us/en/products-services/switching/ex-series.

[6] P4 Compiler (P4C). https://github.com/p4lang/p4c-bm.

[7] P4 Intermediate Representation. https://github.com/p4lang/p4c/.

[8] Albert Gran Alcoz, Alexander Dietmüller, and Laurent Vanbever. SP-PIFO: Approximating push-in first-out behaviors using strict-priority queues. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 59–76, Santa Clara, CA, February 2020. USENIX Association.

[9] Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker. NetKAT: Semantic foundations for networks. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 113–126, New York, NY, USA, 2014. ACM.

[10] Valentin Antimirov. Partial derivatives of regular expressions and finite automata constructions. In Ernst W. Mayr and Claude Puech, editors, *STACS 95*, pages 455–466, Berlin, Heidelberg, 1995. Springer Berlin Heidelberg.

[11] Mina Tahmasbi Arashloo, Yaron Koral, Michael Greenberg, Jennifer Rexford, and David Walker. SNAP: Stateful network-wide abstractions for packet processing. In *Proceedings of*

the 2016 ACM SIGCOMM Conference, SIGCOMM '16, page 29–43, New York, NY, USA, 2016. Association for Computing Machinery.

[12] D. Barach, L. Linguaglossa, D. Marion, P. Pfister, S. Pontarelli, and D. Rossi. High-speed software data plane via vectorized packet processing. *IEEE Communications Magazine*, 56(12):97–103, 2018.

[13] D. Barrington and N. Immerman. Time, hardware, and uniformity. *Proceedings of IEEE 9th Annual Conference on Structure in Complexity Theory*, pages 176–185, 1994.

[14] Ryan Beckett, Eric Campbell, and Michael Greenberg. Kleene algebra modulo theories. *CoRR*, abs/1707.02894, 2017.

[15] Andrea Bianco, Robert Birke, Luca Giraudo, and Manuel Palacin. OpenFlow Switching: Data Plane Performance. In *IEEE International Conference on Communications (ICC)*, 2010.

[16] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. P4: Programming protocol-independent packet processors. *SIGCOMM Comput. Commun. Rev.*, 44(3):87–95, July 2014.

[17] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. P4 Programming Protocol-independent Packet Processors. *ACM SIGCOMM Computer Communication Review (CCR)*, July 2014.

[18] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, pages 99–110, New York, NY, USA, 2013. ACM.

[19] Janusz A. Brzozowski. Derivatives of regular expressions. *J. ACM*, 11(4):481–494, October 1964.

[20] Marco Canini, Daniele Venzano, Peter Perešíni, Dejan Kostić, and Jennifer Rexford. A NICE way to test openflow applications. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 127–140, San Jose, CA, April 2012. USENIX Association.

[21] Sean Choi, Xiang Long, Muhammad Shahbaz, Skip Booth, Andy Keep, John Marshall, and Changhoon Kim. The case for a flexible low-level backend for software data planes. In *Proceedings of the First Asia-Pacific Workshop on Networking*, APNet'17, page 71–77, New York, NY, USA, 2017. Association for Computing Machinery.

[22] Sean Choi, Xiang Long, Muhammad Shahbaz, Skip Booth, Andy Keep, John Marshall, and Changhoon Kim. PVPP: A programmable vector packet processor. In *Proceedings of the Symposium on SDN Research*, SOSR '17, pages 197–198, New York, NY, USA, 2017. ACM.

[23] J.H. Conway. *Regular algebra and finite machines*. Chapman and Hall mathematics series. Chapman and Hall, 1971.

[24] Paul Emmerich, Sebastian Gallenmüller, Daniel Raumer, Florian Wohlfart, and Georg Carle. MoonGen: A Scriptable High-Speed Packet Generator. In *ACM The Internet Measurement Conference (IMC)*, 2015.

[25] Michael J. Fischer and Richard E. Ladner. Propositional dynamic logic of regular programs. *Journal of Computer and System Sciences*, 18(2):194–211, 1979.

[26] Nate Foster, Rob Harrison, Michael J. Freedman, Christopher Monsanto, Jennifer Rexford, Alec Story, and David Walker. Frenetic: A network programming language. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*, ICFP '11, page 279–291, New York, NY, USA, 2011. Association for Computing Machinery.

[27] Nate Foster, Dexter Kozen, Konstantinos Mamouras, Mark Reitblatt, and Alexandra Silva. Probabilistic netkat. In Peter Thiemann, editor, *Programming Languages and Systems*, pages 282–309, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

[28] Nate Foster, Dexter Kozen, Matthew Milano, Alexandra Silva, and Laure Thompson. A coalgebraic decision procedure for NetKAT. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pages 343–355, New York, NY, USA, 2015. ACM.

[29] Merrick Furst, James B. Saxe, and Michael Sipser. Parity, circuits, and the polynomial-time hierarchy. *Mathematical systems theory*, 17(1):13–27, Dec 1984.

[30] Michael R. Garey and David S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1990.

[31] Malvin Gattinger and Jana Wagemaker. Towards an analysis of dynamic gossip in netkat. In Jules Desharnais, Walter Guttmann, and Stef Joosten, editors, *Relational and Algebraic Methods in Computer Science*, pages 280–297, Cham, 2018. Springer International Publishing.

[32] Niels Bjørn Bugge Grathwohl, Dexter Kozen, and Konstantinos Mamouras. KAT + B! In *Proceedings of the Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, CSL-LICS '14, New York, NY, USA, 2014. Association for Computing Machinery.

[33] Tobias Kappé, Paul Brunet, Alexandra Silva, Jana Wagemaker, and Fabio Zanasi. Concurrent Kleene algebra with observations: From hypotheses to completeness. In Jean Goubault-Larrecq and Barbara König, editors, *Foundations of Software Science and Computation Structures*, pages 381–400, Cham, 2020. Springer International Publishing.

[34] Tobias Kappé, Paul Brunet, Alexandra Silva, and Fabio Zanasi. Concurrent Kleene algebra: Free model and completeness. In Amal Ahmed, editor, *Programming Languages and Systems*, pages 856–882, Cham, 2018. Springer International Publishing.

[35] Tobias Kappé, Paul Brunet, Bas Luttik, Alexandra Silva, and Fabio Zanasi. On series-parallel pomset languages: Rationality, context-freeness and automata. *Journal of Logical and Algebraic Methods in Programming*, 103:130–153, 2019.

[36] Peyman Kazemian. *Header Space Analysis*. PhD thesis, Stanford University, 2013.

[37] Peyman Kazemian, George Varghese, and Nick McKeown. Header space analysis: Static checking for networks. In *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 113–126, San Jose, CA, 2012. USENIX.

[38] Ahmed Khurshid, Xuan Zou, Wenxuan Zhou, Matthew Caesar, and P. Brighten Godfrey. Veriflow: Verifying network-wide invariants in real time. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 15–27, Lombard, IL, April 2013. USENIX Association.

[39] Dexter Kozen. On Kleene algebras and closed semirings. Mathematical foundations of computer science, Proc. 15th Symp., MFCS '90, Banská Bystrica/Czech. 1990, Lect. Notes Comput. Sci. 452, 26-47 (1990)., 1990.

[40] Dexter Kozen. A completeness theorem for Kleene algebras and the algebra of regular events. *Information and Computation*, 110(2):366–390, 1994.

[41] Dexter Kozen. Kleene algebra with tests. *ACM Trans. Program. Lang. Syst.*, 19(3):427–443, May 1997.

[42] Dexter Kozen and Konstantinos Mamouras. Kleene algebra with equations. In Javier Esparza, Pierre Fraigniaud, Thore Husfeldt, and Elias Koutsoupias, editors, *Automata, Languages, and Programming*, pages 280–292, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[43] Dexter C. Kozen. *Automata and Computability*. Springer-Verlag, Berlin, Heidelberg, 1st edition, 1997.

[44] Karthik Lakshminarayanan, Anand Rangarajan, and Srinivasan Venkatachary. Algorithms for advanced packet classification with ternary CAMs. In *Proceedings of the 2005 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '05, pages 193–204, New York, NY, USA, 2005. ACM.

[45] Haohui Mai, Ahmed Khurshid, Rachit Agarwal, Matthew Caesar, P. Brighten Godfrey, and Samuel Talmadge King. Debugging the data plane with anteater. *SIGCOMM Comput. Commun. Rev.*, 41(4):290–301, August 2011.

[46] Jedidiah McClurg, Hossein Hojjat, Nate Foster, and Pavol Černý. Event-driven network programming. *SIGPLAN Not.*, 51(6):369–385, June 2016.

[47] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. OpenFlow: Enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 38(2):69–74, March 2008.

[48] C. R. Meiners, A. X. Liu, and E. Torng. Topological transformation approaches to TCAM-based packet classification. *IEEE/ACM Transactions on Networking*, 19(1):237–250, Feb 2011.

[49] Chad R. Meiners, Jignesh Patel, Eric Norige, Eric Torng, and Alex X. Liu. Fast regular expression matching using small TCAMs for network intrusion detection and prevention systems. In *Proceedings of the 19th USENIX Conference on Security*, USENIX Security'10, pages 8–8, Berkeley, CA, USA, 2010. USENIX Association.

[50] P4 Language Specification. `http://p4.org/spec/`.

[51] Aurojit Panda, Ori Lahav, Katerina Argyraki, Mooly Sagiv, and Scott Shenker. Verifying reachability in networks with mutable datapaths. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 699–718, Boston, MA, March 2017. USENIX Association.

[52] Rina Panigrahy and Samar Sharma. Reducing TCAM power consumption and increasing throughput. In *Proceedings of the 10th Symposium on High Performance Interconnects HOT Interconnects*, HOTI '02, pages 107–, Washington, DC, USA, 2002. IEEE Computer Society.

[53] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan J. Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Jonathan Stringer, Pravin Shelar, Keith Amidon, and Martín Casado. The Design and Implementation of Open vSwitch. In *USENIX Conference on Networked Systems Design and Implementation (NSDI)*, 2015.

[54] Proxmox Virtual Environment. `https://www.proxmox.com`.

[55] Mark Reitblatt, Marco Canini, Arjun Guha, and Nate Foster. FatTire: Declarative fault tolerance for software-defined networks. In *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, HotSDN '13, page 109–114, New York, NY, USA, 2013. Association for Computing Machinery.

[56] Ori Rottenstreich, Isaac Keslassy, Avinatan Hassidim, Haim Kaplan, and Ely Porat. On finding an optimal TCAM encoding scheme for packet classification. *2013 Proceedings IEEE INFOCOM*, pages 2049–2057, 2013.

[57] Devavrat Shah and Pankaj Gupta. Fast updating algorithms for TCAMs. *IEEE Micro*, 21(1):36–47, January 2001.

[58] Muhammad Shahbaz, Sean Choi, Ben Pfaff, Changhoon Kim, Nick Feamster, Nick McKeown, and Jennifer Rexford. PISCES: A programmable, protocol-independent software switch. In

Proceedings of the 2016 ACM SIGCOMM Conference, SIGCOMM '16, pages 525–538, New York, NY, USA, 2016. ACM.

[59] Anirudh Sivaraman, Suvinay Subramanian, Mohammad Alizadeh, Sharad Chole, Shang-Tse Chuang, Anurag Agrawal, Hari Balakrishnan, Tom Edsall, Sachin Katti, and Nick McKeown. Programmable packet scheduling at line rate. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, page 44–57, New York, NY, USA, 2016. Association for Computing Machinery.

[60] Steffen Smolka, Spiridon Eliopoulos, Nate Foster, and Arjun Guha. A fast compiler for NetKAT. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, ICFP 2015, pages 328–341, New York, NY, USA, 2015. ACM.

[61] 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. *Proc. ACM Program. Lang.*, 4(POPL), December 2019.

[62] Steffen Smolka, Praveen Kumar, David M. Kahn, Nate Foster, Justin Hsu, Dexter Kozen, and Alexandra Silva. Scalable verification of probabilistic networks. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, page 190–203, New York, NY, USA, 2019. Association for Computing Machinery.

[63] Robert Soulé, Shrutarshi Basu, Parisa Jalili Marandi, Fernando Pedone, Robert Kleinberg, Emin Gun Sirer, and Nate Foster. Merlin: A language for provisioning network resources. In *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies*, CoNEXT '14, page 213–226, New York, NY, USA, 2014. Association for Computing Machinery.

[64] Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5(2):285 – 309, 1955.

[65] Alexander Vandenbroucke and Tom Schrijvers. P$\lambda\omega$nk: Functional probabilistic NetKAT. *Proc. ACM Program. Lang.*, 4(POPL), December 2019.

[66] Andreas Voellmy and Paul Hudak. Nettle: Taking the sting out of programming network routers. In Ricardo Rocha and John Launchbury, editors, *Practical Aspects of Declarative Languages*, pages 235–249, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[67] Vector Packet Processing (VPP) Platform. https://fd.io.

[68] Validating Cisco's NFV Infrastructure Pt. 1. http://www.lightreading.com/nfv/nfv-tests-and-trials/validating-ciscos-nfv-infrastructure-pt-1/d/d-id/718684.

[69] VPP Performance Tests. https://docs.fd.io/csit/rls1701/report/vpp_performance_tests/.

[70] Hongyi Zeng, Shidong Zhang, Fei Ye, Vimalkumar Jeyakumar, Mickey Ju, Junda Liu, Nick McKeown, and Amin Vahdat. Libra: Divide and conquer to verify forwarding tables in huge networks. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 87–99, Seattle, WA, April 2014. USENIX Association.