Universiteit Leiden
The Netherlands

# Master Computer Science

Validation & Simulation of Software-Defined Network Specifications in Probabilistic NetKAT

Floyd Remmerswaal

Supervisors:
Henning Basold, Marcello Bonsangue & Alfons Laarman

MASTER THESIS

Leiden Institute of Advanced Computer Science (LIACS)
www.liacs.leidenuniv.nl

22/12/2023

**Abstract**

Software-Defined Networking has become an important field of study for academia, while at the same time showing success in real-world settings. ProbNetKAT allows one to model interesting problems, such as fault tolerance in networks or expected congestion on specific links, by providing a probabilistic language for Software-Defined Networking. Previous research has not yet produced a compiler for this language. This thesis contributes an implementation of ProbNetKAT in Haskell, as well as a compiler that allows the running of ProbNetKAT programs in the ns-3 network simulator. The operational semantics of nondeterminism in ProbNetKAT requires transforming the intermediate automaton to a specific normal form, which we have implemented. The Haskell implementation enables probabilistic inference on ProbNetKAT programs.

# Contents

# 1 Introduction

In the past two decades, *Software-Defined Networking* (SDN) has introduced a new era of dynamic and flexible network management. Where previously, updating the way the network propagates packets was determined in the switches and routers themselves, SDN centralizes control in software-based controllers. This paradigm shift provides network administrators with more flexible control over the network, enabling them to adapt and optimize network behavior in real-time, responding to changing demands and conditions.

Traditional SDN has proven itself with real-world successes, for example, in cloud and Internet of Things settings. Whereas most SDN languages have some form of nondeterminism or probabilistic branching, the probabilistic SDN language ProbNetKAT combines both probabilistic choice and nondeterminism to allow handling of uncertainties inherent in real-world network environments. The probabilistic and nondeterministic semantics of ProbNetKAT can be used to model, for example, unreliable network connections that may randomly drop packets. Having a working implementation of this language allows for research into probabilistic networking protocols, which this thesis aims to provide.

The main contributions of this thesis are a Haskell implementation of the semantics of ProbNetKAT in terms of probability distributions, as well as a compiler for ProbNetKAT programs into a form usable by the discrete network simulator ns-3. The former enables probabilistic inference on ProbNetKAT programs and, for example, analysis of probabilistic forwarding protocols for a modeled network. The latter facilitates the evaluation of ProbNetKAT programs within a simulated network environment by visualizing the effects of the program on the behavior of packets in the network. In this thesis, any time inference is written, it is to be taken as referring to probabilistic inference specifically.

## 1.1 Thesis Overview

We describe the background of ProbNetKAT in Section 2. Section 3 discusses the semantics of ProbNetKAT and inference on ProbNetKAT programs. In Section 4, we look at the process of compiling ProbNetKAT programs and some potential problems that arise. Section 5 contains details on the implementation of the topics discussed in Sections 3 and 4. We give our conclusion in Section 6 and future work in Section 7.

## 1.2 Related Work

Previous work has been done on Probabilistic NetKAT and its predecessors. Smolka et al. [Smo+15] introduce and discuss a NetKAT compiler. This work uses an intermediary automaton, aptly called a NetKAT automaton, which has the same expressive power as the NetKAT language itself. In our implementation, discussed in more detail in later sections, we also convert ProbNetKAT programs into automata. The workings of the two types of automata are similar, but Smolka et al. include a global and a local compilation step. The automaton is used for the global compilation, whereas the local compilation uses Forwarding Decision Diagrams (a generalization of BDDs) to output forwarding tables. We do not directly produce forwarding tables in our compilation.

However, although ProbNetKAT has advantages over NetKAT, it comes with its own caveats. Kahn [Kah17] shows that there are certain undecidable problems for probabilistic network programming with some results specifically for ProbNetKAT. These results follow from the embedding of the undecidable *Post-correspondence* problem and probabilistic finite automata in the language.

Another article discussing decidability for problems regarding ProbNetKAT programs is *Scalable verification of probabilistic networks* by Smolka et al. [Smo+18]. They show that for ProbNetKAT without histories, the problem of determining program equivalence is decidable. General results without restrictions are not yet known. A notable difference between their work and ours is this restriction, as we do support histories in the probabilistic inference.

In *Scalable Verification of Probabilistic Networks*, Smolka et al. [Smo+19] present an implementation of the guarded and history-free fragment of ProbNetKAT, named McNetKAT, which is capable of verifying probabilistic network programs of thousands of nodes.

Where ProbNetKAT extends the NetKAT language, Vandenbroucke and Schrijvers [VS19] in turn propose a functional extension of Probabilistic NetKAT called P$\lambda\omega$NK, allowing higher-order functions that could make writing SDN programs easier.

Because ProbNetKAT has both probabilistic and nonodeterministic branching, the automata we use also have these features. Sokolova [Sok11] have studied a variety of probabilistic systems coalgebraically. Systems with nondeterministic branching followed by probabilistic branching are called *Segala systems*. After normalization, our automata are more closely related to *bundle systems*. The main difference is that we keep track of the number of repetitions in nondeterministic branching, akin to using multi-sets as described by Jacobs [Jac21]. Of course, our original (not normalized) automata can arbitrarily mix nondeterministic and probabilistic choice which makes them *Pneuli-Zuck systems*, rather than Segala systems.

# 2 Background

ProbNetKAT is based on existing work on the NetKAT SDN language, which itself is an extension on the Kleene Algebra With Tests. This section provides an overview of ProbNetKAT's background in relation to Software-Defined Networking (SDN) and its predecessors.

## 2.1 Software Defined Networking

In traditional networking, switches forward network packets according to forwarding tables. Changes in the structure of the network therefore require updating the forwarding tables of all affected switches. This process not only becomes impractical in large and dynamic networks but also hinders the flexibility required to adapt to changing demands.

The main idea behind SDN is to separate the *data plane* and the *control plane* of the network. The data plane is the network layer that consists of switches that perform forwards based on forwarding tables, while the control plane is made up of programs and servers that determine and manage these tables. In SDN, you centrally program the behavior of the network and push the changes to the switches.

## 2.2 NetKAT

Based on Kleene Algebra with Tests (KAT), Anderson et al. [And+14] proposed NetKAT as an SDN language. The language extends KAT with primitives for reasoning about networking. NetKAT is a Kleene Algebra with Tests with extra assumptions and constants. The essential object in NetKAT is the network *packet*, which consists of a number of *fields*. A field can be thought of as a variable, with a name and a value. A *history* is an ordered list of packets, conceptually keeping track of changes to the packet as it moves through the network. The first item of the list denotes the current state of the packet. In NetKAT, the constant *dup* denotes creating a copy of the head of the history, growing the history. The other additions to NetKAT are *assignment* (to fields), *tests* (on fields), *drop*, *skip*, sequential composition ($\cdot$), and parallel composition ($+$). The Kleene star is still present from KAT, and denotes zero or more repetitions. Tests behave like a *skip* or no-op when the test is passed or as a drop if the test fails.

As stated before, NetKAT is a Kleene Algebra with Tests, which is a tuple $(K, B, +, \cdot, *, 0, 1, \neg)$ such that [And+14]:

- $(K, +, \cdot, *, 0, 1)$ is a Kleene algebra
- $(B, +, \cdot, \neg, 0, 1)$ is a Boolean algebra
- $(B, +, \cdot, 0, 1)$ is a structural subalgebra of $(K, +, \cdot, 0, 1)$

NetKAT satisfies these structural demands, but these demands are not enough to define NetKAT. The operations discussed above (*dup*, for example) require more structure, the axioms of which are given by Anderson et al. [And+14] but will not be discussed in their entirety in this thesis. In terms of denotational semantics, NetKAT programs denote functions of type $H \to \mathcal{P}(H)$, mapping histories to sets of histories. Dropping maps the history to the empty set.

## 2.3 ProbNetKAT

Building on the basis of NetKAT, Foster et al. [Fos+16] proposed to extend NetKAT to include probabilistic and nondeterministic branching. This allows modeling behavior such as a faulty network link that sometimes drops packets. This article serves as the main inspiration for our research. Foster et al. [Fos+16] proved that the extension from NetKAT to ProbNetKAT is conservative, which means that any ProbNetKAT program that does not have probabilistic choices has the same semantics as in NetKAT. However, some symbols differ as the parallel composition in NetKAT is given by + and by & in ProbNetKAT. ProbNetKAT extends NetKAT with the random choice operation $p \oplus_r q$ for some ProbNetKAT expressions $p$ and $q$. The argument $r \in [0, 1]$ denotes the probability for the left side of the operator; the probability for the right side is then $1 - r$. When $r = 0.5$ this can be omitted for brevity.

If we want to model, for example, a faulty network link in ProbNetKAT, we can do so by incorporating a chance of dropping the packet in the network topology. A very simple example can be seen in Figure 1. Between switches $S1$ and $S2$ there is a 10% chance that the packet is lost. We cannot model this with NetKAT, but in ProbNetKAT we can simply encode the edge as $sw = S1; sw \leftarrow S2 \oplus_{0.9} drop$. A slightly more complex example will be discussed in more detail in Section 3.2.

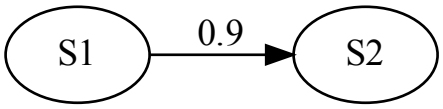


Figure 1: Very simple network example. Packets travelling from $S_1$ to $S_2$ have probability 0.9 of succeeding and 0.1 of being dropped.

# 3 Probabilistic Inference

## 3.1 Semantics of ProbNetKAT

The semantics of ProbNetKAT is defined by Foster et al. [Fos+16] in terms of Markov kernels [Kle20, Def. 8.25] . To understand the meaning of ProbNetKAT programs, we do not have to fully understand this mathematical basis. Instead, we can think of ProbNetKAT programs as taking in a network packet (formally, a history) and outputting a distribution on sets of histories. Recall from Section 2 that a *history* is defined to be a non-empty sequence of packets, keeping track of the path of the packet through the network. The atomic operations used in ProbNetKAT have fairly simple semantics, as given below. In the definition, $\delta$ denotes the Dirac measure, and $a \in 2^H$ is a set of histories. The type of the semantics is $2^H \to \mathcal{B} \to \mathbb{R}$. For an input $a \in 2^H$, the program $p$ produces an output following the distribution given by $[\![p]\!](a)$. $\mathcal{B} \subseteq 2^{(2^H)}$ are the Borel sets [Kle20, Def. 1.21] of the topology generated by basic clopen sets on the powerset $2^H$. The definitions given by Foster et al. [Fos+16] are as follows:

$$
\begin{aligned}
[\![p]\!] &: 2^H \to \mathcal{B} \to \mathbb{R} \\
[\![x \leftarrow n]\!](a) &= \delta_{\{\pi[n/x]:\eta \mid \pi:\eta \in a\}} \\
[\![x = n]\!](a) &= \delta_{\{\pi:\eta \mid \pi:\eta \in a, \pi(x)=n\}} \\
[\![dup]\!](a) &= \delta_{\{\pi:\pi:\eta \mid \pi:\eta \in a\}} \\
[\![skip]\!](a) &= \delta_a \\
[\![drop]\!](a) &= \delta_\theta
\end{aligned}
$$

As an example, we can see that the *drop* instruction in ProbNetKAT maps any incoming packet to a distribution that has probability 1 for the empty set and 0 everywhere else. Chaining these atomic operations is done with the sequential composition operator (;). Its semantics is given by:

$$[\![p; q]\!](a) = [\![q]\!]([\![p]\!](a))$$

When programs $p$ and $q$ are composed sequentially, the output of $p$ becomes the input of $q$. This is nontrivial, as the output type of $[\![p]\!](a)$ is a distribution, rather than a set of histories, as its input is. Foster et al. [Fos+16] extend the definition of $[\![q]\!]$ by integration as shown below.

$$[\![q]\!](\mu) \triangleq \lambda A. \int_{a \in 2^H} [\![q]\!](a, A) \cdot \mu(da), \text{for } \mu \text{ a probability measure on } 2^H$$

Essentially, the required composition is the Kleisli composition. Another interesting part comes from the nondeterministic (or parallel) ('&') and probabilistic composition ('$\oplus$') operators. The probabilistic composition takes an argument $r$ that denotes the probability that the first argument is used. Semantically, we multiply this $r$ by the distribution produced by $[\![p]\!]$ to scale the distribution. Of course, the second option has probability $1 - r$. For convenience, when $r = 0.5$, we can omit this parameter and simply write $\oplus$. Combining both nondeterminism and probabilistic choice results in

some problems with the operational semantics, which will be discussed in more detail in Section 4. Intuitively, the parallel composition means that both subprograms are executed, and the results are combined. In this sense, *drop* behaves like the identity for parallel composition. Similarly, *skip* is the identity for sequential composition. Foster et al. [Fos+16] give the following semantics for these compositions:

$$[\![p \& q]\!](a) = [\![p]\!](a) \& [\![q]\!](a)$$
$$[\![p \oplus_r q]\!](a) = r[\![p]\!](a) + (1-r)[\![q]\!](a)$$

We still need a definition for "&" on distributions as that is missing. Foster et al. [Fos+16] give the following definition for combining distributions. Given measures $\mu$ and $\nu$, both measures are sampled independently, after which we take the union of these results (both subsets of histories).

$$(\mu \& \nu)(A) \triangleq (\mu \times \nu)(\{(a,b) | a \cup b \in A\})$$

Finally, the Kleene star ('*') operation denotes iteration on a program. Giving its semantics is not trivial. To obtain a definition, Foster et al. [Fos+16] construct an infinite stochastic process starting with $c_0 \in 2^H$. They create the sequence by defining $c_{n+1}$ as the result of sampling $2^H$ according to the distribution $[\![p]\!](c_n)$, giving the infinite sequence $c_0, c_1, c_2, \cdots \in (2^H)^\omega$. Then they ask if the union of the resulting sequence $\bigcup_n c_n$ is in $A$ ($A \in \mathcal{B}$. We take the probability that $A$ will be sampled from this union to be equal to $[\![p*]\!](c_0, A)$.

## 3.2 Probabilistic Inference

Probabilistic programming languages, such as ProbNetKAT, inherently deal with distributions rather than concrete values. We often want to reach some conclusions despite the uncertainty that these distributions introduce in our decision-making process, such as predicting the chance that our probabilistic protocol works as intended or estimating the congestion in a network link. Probabilistic inference refers to taking a given prior probability distribution and attempting to estimate the outcome by updating our knowledge. As an example, we could take a network topology and the protocol and model it by implementing both in ProbNetKAT. We can assign to every network edge a probability reflecting the chance that this edge passes packets successfully. This program then models the behavior of the entire system, and we can draw samples from it. Depending on the situation, we might be able to completely enumerate all possible execution paths.

To expand on this example and see the possibilities enabled by implementing the semantics of ProbNetKAT, we look at a fault tolerance example by Foster et al. [Fos+16]. Figure 2 shows the network topology used in the example.

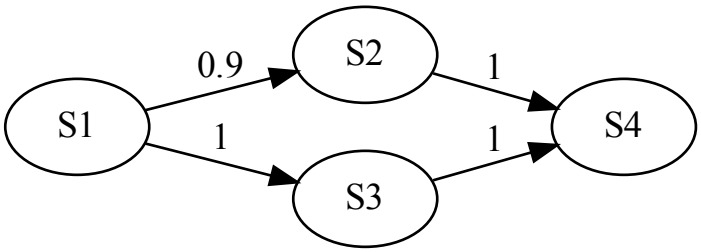


Figure 2: Example of a network topology with one link having a 10% fail rate.

We have four network switches, marked $S1$ through $S4$. All links except $S1 \to S2$ have probability 1 of working correctly, while $S1 \to S2$ only has probability 0.9, which means that it fails 10% of the time. In ProbNetKAT we would encode this topology as follows:

$$\begin{aligned} t \triangleq & (sw = S_1; pt = 2; ((sw \leftarrow S_2; pt \leftarrow 1) \oplus_{.9} drop)) \\ & \& \ (sw = S_1; pt = 3; sw \leftarrow S_3; pt \leftarrow 1) \\ & \& \ (sw = S_2; pt = 4; sw \leftarrow S_4; pt \leftarrow 2) \\ & \& \ (sw = S_3; pt = 4; sw \leftarrow S_4; pt \leftarrow 3) \end{aligned}$$

One possible, although admittedly not very clever, policy for forwarding packets in this network would be to send all traffic from $S1$ to $S4$ through $S2$. This policy in ProbNetKAT looks like this:

$$p \triangleq (sw = S_1; pt \leftarrow 2) \& (sw = S_2; pt \leftarrow 4)$$

If the packet is in $S_1$, we set the port of the packet to 2, making the topology forward it to $S2$. Similarly, if the packet is in $S2$, we set the port to 4. Combining both policy and topology into a single ProbNetKAT program would be $(p; t)^*$, alternating between the policy and the network topology. Finally, we would like to know how many packets end up in $S_4$. To this end, we add an egress predicate to the program $e \triangleq sw = S_4$. The complete program is then $(p; t)^*; e$. When the Haskell implementation of ProbNetKAT is given this program with an input, it will show the output distribution of the program. In this case, the output is as follows.

- 90.00% : [[(4,2)]]
- 10.00% : []

For the input $[[(1, 0)]]$. We can see that we have a probability of 90% that the packet ends up in $(4, 2)$ and a probability of 10% that the packet is dropped.

If we want to see the history of the packet and see how it traveled through the network, we can edit the topology to always *dup* the packet before editing the port.

$$\begin{aligned} t' \triangleq & (sw = S_1; pt = 2; dup; ((sw \leftarrow S_2; pt \leftarrow 1) \oplus_{.9} drop)) \\ & \& \ (sw = S_1; pt = 3; dup; sw \leftarrow S_3; pt \leftarrow 1) \\ & \& \ (sw = S_2; pt = 4; dup; sw \leftarrow S_4; pt \leftarrow 2) \\ & \& \ (sw = S_3; pt = 4; dup; sw \leftarrow S_4; pt \leftarrow 3) \end{aligned}$$

The output of the network is as follows.

- 90.00% : [[(4,2),(2,4),(1,2)]]
- 10.00% : []

If we change the forwarding protocol to send any packet randomly through $S2$ or $S3$, we can improve on this 90%. The protocol $p'$ as defined below implements this change.

$$p' \triangleq (sw = S_1; (pt \leftarrow 2 \oplus pt \leftarrow)) \& (sw = S_2; pt \leftarrow 4) \& (sw = S_3; pt \leftarrow 4)$$

The output distribution reflects the fact that packets passing through $S2$ have a chance of failing. We are left with only 5% dropped packets. We can conclude that policy $p'$ has better fault tolerance than policy $p$ for this specific network topology.

- 50.00% : [[(4,3)]]
- 45.00% : [[(4,2)]]
- 5.00% : []

It is the convention that the port number denotes the switch number of the other side of the connection. So port 3 of switch 1 is the port connecting to switch 3. Similarly, the packet is received on port 1 of switch 3.

### 3.3 Semantics in Haskell

As previously stated, the semantics of ProbNetKAT were implemented in Haskell. First, we define the type of a *packet* to be a tuple of two integers, the first being the switch and the second the port. A *history* is simply an (ordered) list of packets. We handle distributions on sets of histories. We restrict packets to only consisting of these two values, as that makes the semantics easier to implement and test, as the dimension of the tuple is constant. However, any payload could be quite easily added as additional fields or by changing a packet to be a collection (set or otherwise) of pairs of strings and integers. This would match more closely the definition of [Fos+16], but for simplicity, we have not done so.

One can think of ProbNetKAT programs as taking in sets of packets and outputting distributions on SH. We use Monad-Bayes, a library that implements probabilistic programming in Haskell by Scibior [Sci23] for our probabilistic monads.

A Kleisli arrow type takes the form Kleisli $m\ a\ b$, corresponding to a function of type $a \to mb$ for types $a$ and $b$ and some monad $m$ (a probabilistic monad in our case). Using these Kleisli arrows, we can perform the composition of subexpressions with Kleisli composition. This means that we can define the semantics of individual operations in terms of mapping histories to distributions of sets of histories and have the Kleisli composition deal with using distributions as input. We use the following types, where $m$ denotes a probabilistic monad from the monad-bayes package ("*Enumeration*", for example).

```
type Packet = (Integer, Integer) -- sw is the first element, pt is the second
type History = [Packet]
type SH = Set History
type KSH m = Kleisli m SH SH
```

The semantics of the atomic actions are then implemented by lifting basic functions into Kleisli arrows. Taking the assignment, for example, to change the switch on a packet, we simply map a packet to a packet with the correct switch value. We take that function and map it to the input set. The resulting mapping is then lifted to a Kleisli arrow with *arr*. The 'assign port' function is essentially the same. Drop maps to a singleton set containing only the empty set; skip is the identity function (or rather the identity arrow). Testing involves filtering the sets based on the given test and mapping a packet to the empty history ('dropping') if it fails the test.

```
changeSw :: Integer -> History -> History
changeSw _ [] = []
changeSw i ((_,y):xs) = (i,y) : xs

assignSw :: MonadDistribution m => Integer -> KSH m
assignSw s = arr $ Set.map (changeSw s)
```

Parallel composition means that we take the union of executing both input arrows, which is achieved by liftA2. The sequential composition is implemented simply with >>>, which is the left-to-right composition for Kleisli arrows. This accomplishes exactly what we want: the output of the left-hand side is the input for the right-hand side. The probabilistic composition takes a probability $r$ and draws a Bernoulli random variable that is 1 with probability $r$. This results in a probabilistic Kleisli arrow.

```
par :: MonadDistribution m => KSH m -> KSH m -> KSH m
par = liftA2 Set.union

seq :: MonadDistribution m => KSH m  -> KSH m  -> KSH m
seq = (>>>)

prob :: MonadDistribution m => Double -> KSH m -> KSH m -> KSH m
prob r f g = Kleisli $ \h -> do
  x <- bernoulli r
  if x then runKleisli f h else runKleisli g h
```

Finally, we have to implement the semantics of the Kleene star. Foster et al. [Fos+16] give an approximation for the Kleene star by performing a finite number of repetitions. We use the fact

that $[p^*]$ is approximated by $[p^{(M)}]$ for sufficiently large $M$ [Fos+16]. We define $p^{(0)} = skip$ and $p^{n+1} = skip \,\&\, p\,;\, p^{(n)}$. This is easily implemented in Haskell if the previous composition operators have already been implemented. The 'depth' of the approximation makes the resulting computation scale exponentially with the number of possible paths. The depth is configurable with the constant value kleeneDepth in our code (with default value 5).

```
kleeneApprox :: MonadDistribution m => Integer -> KSH m -> KSH m
kleeneApprox 0 _ = skip
kleeneApprox n p = skip `par` seq p (kleeneApprox (n-1) p)

kleene :: MonadDistribution m => KSH m -> KSH m
kleene = kleeneApprox kleeneDepth
```

When we look at the example discussed in Section 3.2, we can already see a very noticeable difference between $p$ and $p'$. Using depth 10 for the Kleene star approximation on modest hardware, the former works without issue, while the latter gets killed by the operating system for using too many resources. Although Foster et al. [Fos+16] prove that we can approximate $[p^*]$ mathematically, in practice it might not always be feasible.

# 4 Compilation

In order to run ProbNetKAT programs on the network simulator, we have to transform the program into something the simulator nodes can run. The NS-3 simulator, as well as the behavior of the simulated nodes, is written in C++. We transform the abstract syntax tree of a ProbNetKAT program into an automaton. The nodes in the network run the ProbNetKAT program by adding a field to the ProbNetKAT header (attached to the packet) that behaves like a program counter. Before we can execute the program we have to transform it to a form that can be executed in the ns-3 simulator. To this end the program is converted to an automaton. Therefore, in the ProbNetKAT header we keep track of where in the automaton (node number) the packet currently is. This automaton is structurally similar to the expression tree from which it is derived, with some key differences.

In the automaton, nodes contain atomic instructions (*dup*, *drop*, *skip*, assignment, or test) or denote a branch (either probabilistic or parallel). Atomic instructions have zero or one outgoing edge. The probabilistic and parallel nodes have (at least) two outgoing edges. In the case of the probabilistic nodes, the outgoing edges also have a weight, denoting the probability for that branch to be taken. For the parallel composition nodes, the outgoing edges have no weight. How these are handled is explained in more detail in Section 4.2.1.

Note that there are no sequential composition nodes left in the automaton. Atomic operations that are sequentially composed form a chain in the automaton. If the sequential composition is of sub-trees that are more complex and feature probabilistic or nondeterministic branching, the resulting automaton is more complex than a chain as well. The right sub-tree of the sequential composition must be placed at the end of all branches of the left sub-tree. This is further complicated by the Kleene star, which ends up as edges to earlier nodes in the automaton. This means that we cannot use a tree datastructure, and a more general graph structure must be used.

Figure 3 shows the transition from abstract syntax tree to automaton for a program $(sw \leftarrow 1; skip)^*$. For illustration purposes, this program is very simple.


(a) Abstract syntax tree representation.

(b) Automaton representation.

Figure 3: Program $(sw \leftarrow 1; skip)^*$ represented as a tree and as an automaton (graph).

## 4.1 ns-3

The target for our compilation of ProbNetKAT is the network simulator ns-3. This discrete event network simulator is written in C++ and allows one to write code that will be executed on the simulated network nodes. Therefore, we want to compile ProbNetKAT programs to a C++ snippet that can be compiled into the simulation. Implementation details will be discussed in more detail in Section 5.

We simulate a given network topology and send packets through this network. As the purely mathematical semantics of ProbNetKAT does not have a sense of the physical nature of networks (for example, network switches and latency), we have to make some decisions when we want to actually *run* ProbNetKAT programs.

An important point is that there is no 'send' instruction in ProbNetKAT. This means that we have to choose a different way of interpreting when a switch that implements a ProbNetKAT program should forward a packet. We have chosen to use the *dup* command to mean 'send the current packet to the switch indicated by the switch number in the packet. As *dup* allows you to keep track of the way a packet moves through the network and is therefore naturally called at the positions in the program where a packet moves between switches, it is a reasonably natural fit.

The result of compiling a ProbNetKAT program is a C++ program snippet that contains instructions to build the graph (automaton). The simulated network nodes interpret the program by finding the node in the automaton where the execution of the program is left.

## 4.2 Probability First Normal Form

### 4.2.1 Operational meaning of &

To give operational meaning to parallel composition, we allow the network nodes to choose one of the possible paths from a parallel composition node in the automaton. If a packet happens to be dropped somewhere further in its lifetime, the node that drops the package communicates this to the rest of the network via the control node. The specific path taken in the automaton is marked, and subsequent packets at the same node at the same stage of their execution will take the other path (provided that the other path has not been marked yet).

However, this approach is not enough, as the result of either path from the parallel composition node can be random. See Figure 4 for a visualization of the problem. If by chance in the left subtree the *drop* path is taken, our approach would mark the root node, making sure that a similar packet will take the right subtree. If the next packet also ends up being dropped, we will have both sides marked. To prevent this, we must prevent probabilistic choices after parallel branches. To this end, we convert all programs to a Probability First Normal Form.

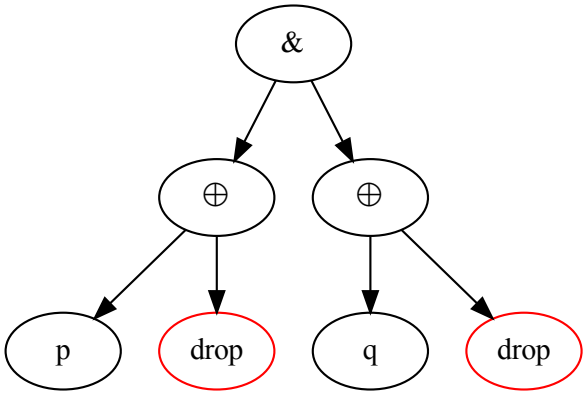


Figure 4: Automaton showing a problem that arises from combining probabilistic and nondeterministic branching. The red *drop* nodes are problematic for the operational meaning of the parallel composition.

### 4.2.2 Converting to Probability First

In Probability First Normal Form we do not allow probabilistic branching after any nondeterministic branching. In general, we want to have all probabilistic branching done at the root of the automaton. We swap the parallel and probabilistic composition and extend the automaton to include all possible execution paths. This can be seen in Figure 5. At first glance, it is obvious that the automaton grows very quickly when converted to this Normal Form.

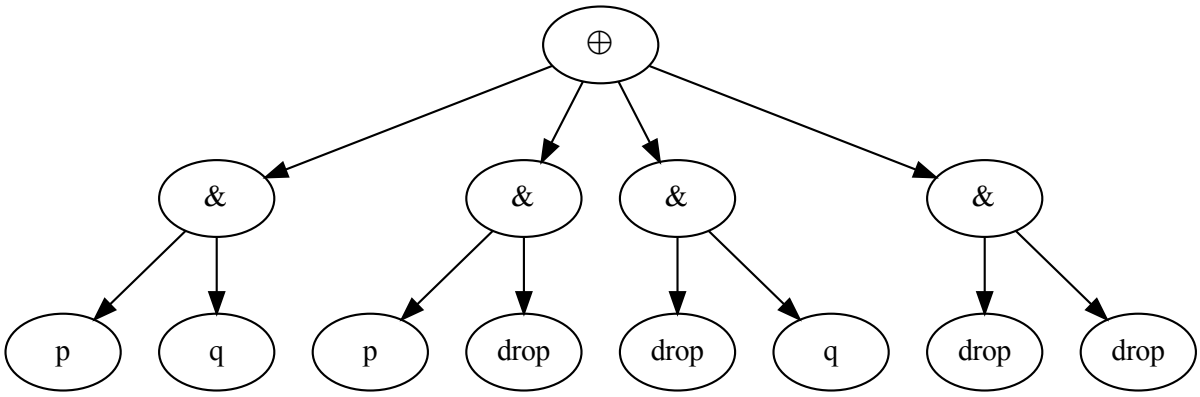


Figure 5: The same program as in Figure 4, but in Probability First Normal Form. The semantics of the programs are equivalent, but this version fixes the operational problems with the parallel composition.

The implementation of normalization is discussed in more detail in Section 5.5. An important consideration is also that we should keep track of the parallel branches that a packet has taken in the automaton. These could be nested, and if a packet is dropped, we should mark the last parallel branch that was taken. Of course, this could be further optimized by marking higher branches if both execution paths end up marked.

# 5 Implementation

The project consists of one main Haskell program along with a number of modules. Figure 6 depicts the flow of the program and the module responsible for each part. All source code is publicly available on GitHub: `https://github.com/floydremmerswaal/probnetkat`. The program parses the command-line input and handles reading the input files. The parser is generated with a parser generator, discussed in more detail in Section 5.1. This parser generates an abstract syntax tree of the input program (if the input is valid). This syntax tree is transformed into a Kleisli arrow as discussed in Section 3, or compiled into C++ instructions as described in Section 4. After we have constructed the Kleisli arrow representing the program, we can easily calculate the output distribution of running the program on an input distribution.

Also included in this project are several programs written to allow our generated C++ code to be ran in the ns-3 simulator. This is explained in more detail in Section 5.6.

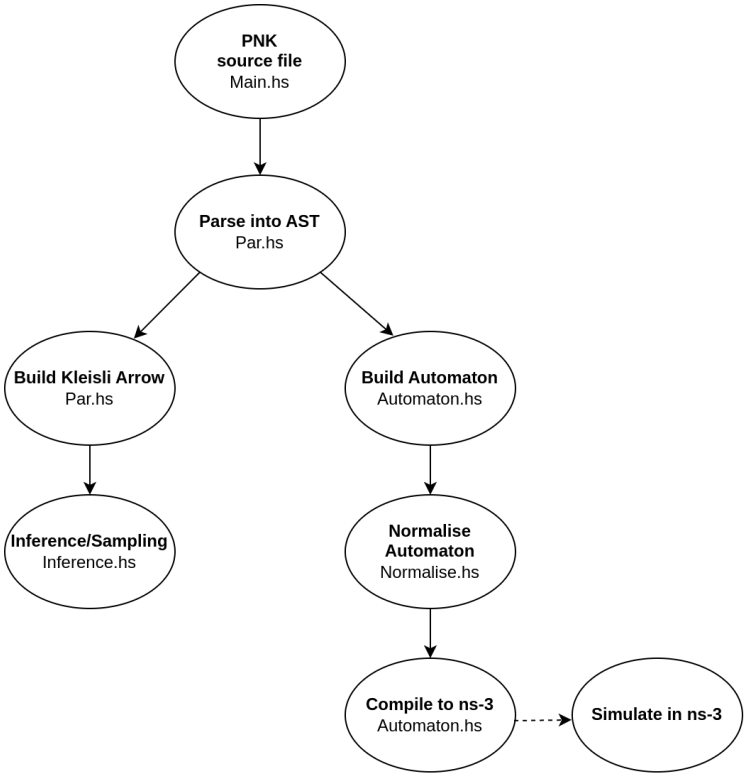


Figure 6: Diagram showing the workings of the project. The regular text under the bold text refers to the module that contains the code that accomplishes the step. The dotted line towards the simulation step means that it is a step that has to be done manually by the user.

## 5.1 Generating the parser

In order to parse ProbNetKAT programs, we need a parser. The compiler front-end generator BNFC takes in a BNF and produces a parser in the desired language, as well as datatypes for abstract syntax trees. We have chosen to target Haskell. BNFC targets the Alex lexer generator and the Happy parser generator.

The BNF used to parse ProbNetKAT can be seen in Figure 7. As explained previously, we have chosen to restrict packets to contain two fields, the switch and the port number. To ease implementation, the assignments and tests for these fields are encoded as different expressions instead of a general assignment and test expression that has both a name and a value argument. We have included both line (anything after "//") and multiline (anything between "/*" and "*/") comments to allow one to annotate their ProbNetKAT programs.

```
entrypoints Exp;

EAssSw.  Exp3    ::= "sw <-" Integer;
EAssPt.  Exp3    ::= "pt <-" Integer;
ESwEq.   Exp3    ::= "sw =" Integer;
EPtEq.   Exp3    ::= "pt =" Integer;
EDup.    Exp3    ::= "dup";
ESkip.   Exp3    ::= "skip";
EDrop.   Exp3    ::= "drop";

ESeq.    Exp2    ::= Exp2 ";" Exp3;
EProbD.  Exp1    ::= Exp1 "+" Exp2;
EProb.   Exp1    ::= Exp1 "+[" Double "]" Exp2;
EPar.    Exp     ::= Exp "&" Exp1;

EKleene. Exp     ::= Exp "*";

coercions Exp 3;

comment "//" ;
comment "/*" "*/" ;
```

Figure 7: BNF grammar for ProbNetKAT to be used by BNFC.

## 5.2 Build the Kleisli Arrow

When the lexer and parser are finished, we are left with the abstract syntax tree that represents the program. To build the Kleisli arrow used for inference, we can recursively walk through this tree, as seen in the Haskell code below. When the semantics of the actions and compositions are defined, we simply use these functions for the corresponding nodes in the tree. The result is a single arrow representing the entire computation described by the ProbNetKAT program. Note that the parser differentiates between probabilistic composition with and without argument ($EProb$ and $EProbD$) to allow users to simply write $\oplus$ when $r = 0.5$, instead of $\oplus_{0.5}$.

```
transExp :: MonadDistribution m => Exp -> Kleisli m SH SH
transExp x = case x of
  EDup                    -> dup
  ESkip                   -> skip
  EDrop                   -> drop
  EAssSw  integer         -> assignSw integer
  EAssPt  integer         -> assignPt integer
  ESwEq   integer         -> testSw integer
  EPtEq   integer         -> testPt integer
  EKleene exp1            -> kleene (transExp exp1)
  ESeq    exp1 exp2       -> seq (transExp exp1) (transExp exp2)
  EPar    exp1 exp2       -> par (transExp exp1) (transExp exp2)
  EProbD  exp1 exp2       -> prob 0.5 (transExp exp1) (transExp exp2)
  EProb   exp1 double exp2  -> prob double (transExp exp1)  (transExp exp2)
```

## 5.3 Probabilistic Inference

The Haskell library Monad-Bayes implements probabilistic monads that allow us to do inference and sampling. We use the Enumerate monad to get a complete and exact result of the resulting distribution after passing the input packets through the program by enumerating all possible paths. Alternatively, we can use the SampleIO monad (with fixed or random seeding) to generate any number of samples we want. Both options are available to the user via different command-line options. To show how this works, we reuse the example in Section 3.2. Recall that we have a network consisting of four nodes, $S_1 - S_4$. The link between $S_1$ and $S_2$ is only 90% reliable. To perform the inference step shown for that example, we first write the program in a file:

```
// total network is (p;t)*;e
// policy (p)
((((sw = 1; pt <- 2) & (sw = 2; pt <- 4));
// topology (t)
((sw = 1; pt = 2; dup; (( sw <- 2 ; pt <- 1) +[0.9] drop)) &
 (sw = 1; pt = 3; dup; sw <- 3; pt <- 1) &
 (sw = 2; pt = 4; dup; sw <- 4; pt <- 2) &
 (sw = 3; pt = 4; dup; sw <- 4; pt <- 3)))*)
// egress (e), or what ends up in switch 4?
; (sw = 4)
```

This file is available in the repository as *test/infer.pnk*. We then run the inference on it by running the program on this input file with the input [[(1,0)]], which means that the packet starts in switch 1. We use option *-i* to run exact inference.

```
stack run -- -i test/infer.pnk "[[(1,1)]]"
```

The output of the program shows the resulting distribution:

```
90.00% : [[(4,2)]]
10.00% : []
```

## 5.4 Building the Automaton

In the compilation step, we convert the abstract syntax tree into an automaton, represented by a graph data structure. We use the Functional Graphing Library, originally motivated by the author Martin Erwig in his paper *Inductive Graphs and Functional Graph Algorithms* [Erw01]. This package allows users to construct arbitrary graphs, internally represented by a *patricia tree*.

The graph is constructed by visiting the AST nodes in a depth-first manner. Due to the structure of the tree, any leaf node is an atomic instruction, and all other nodes are either one of the compositions or the Kleene star. When a sequential composition is encountered, we first recursively construct the graph for the left child, after which we can do the same for the right child. In an (AST) leaf node, we create an edge from the (graph) parent to the generated (graph) node. To do this, the recursive function call takes the node number to be used by the child, and the parent node number. In a similar manner, the probabilistic and nondeterministic branchings are dealt with. However, they are simpler, as they retain the structure from the AST and do not need to connect their subtrees together. To handle the Kleene star correctly, the recursive function has another argument that signifies if the current context is in the subtree of a Kleene star. This is necessary as we have to loop back to an earlier state in the automaton. We also need to know which state to loop back to. A notable exception to this is that even if we are currently processing the subtree of a Kleene star, the left hand side of a sequential composition should not loop back to the parent node of the Kleene star. Only the last expression ('most right hand') has an edge back up. The resulting graph can be operationally problematic, as discussed earlier in Section 4.2. We will discuss the solution to these problems in the next section.

## 5.5 Normalizing the Automaton

The normalization step assumes that an automaton already exists as specified in previous sections. To convert the graph to a normalized version, we use an intermediate tree representation again. This spanning tree contains the weights of the edges in a list inside the nodes instead of on the edges.

A state is maintained during the transformation, remembering the next node to generate (an incremented integer) and a map from integers to lists of nodes that maps a node number to a list of nodes that have been generated from that node. This map starts empty. While walking through the tree recursively, nondeterministic choices are merged down, and the probabilities of probabilistic choices are multiplied and normalized. After we have recursively built up the new spanning tree, it is converted back into an automaton, which is now normalized.

The resulting automaton is converted into C++ instructions by listing all nodes and edges of the graph, resulting in the construction of an equivalent data structure in the C++ program. Some optimizations are definitely still possible, but beyond the scope of this thesis. This could include, for example, detecting equivalent sub-graphs and merging execution paths.

## 5.6 Ns-3 structure

Ns-3 allows us to program our own simulations. In the simulation, we first generate the network topology. In our tests, we have always used a network consisting of $n$ nodes, all connected to every other node. These nodes run a program that we also provide. This program handles sending and receiving the network packets and also runs the ProbNetKAT program. The packets themselves are UDP packets with some additional data, notably the *switch* and *port* numbers of the ProbNetKAT packet, as well as the ProbNetKAT automaton node that should be executed next.

In total, we actually use $n + 1$ nodes in the simulated network and use that extra ingress node to send initial packets into the network. The simulation program contains a specification on when and where the initial packets should be sent. Alternatively, one can specify the rate at which packets should be sent for the entire duration of the network simulation. Of course, more intricate ingress policies could be programmed into the simulation.

After running the simulation, it can be visualized with the NetAnim program provided by ns-3. The simulated nodes are visualized in a ring in the bottom right, whereas the ingress node is placed in the top left to avoid confusion. An example of a visualization can be seen in Figure 8. In the figure, we see a network packet moving from node 0 to node 1. We built our implementation on top of the UDP implementation in ns-3, which is why the packet shows up as a UDP packet.

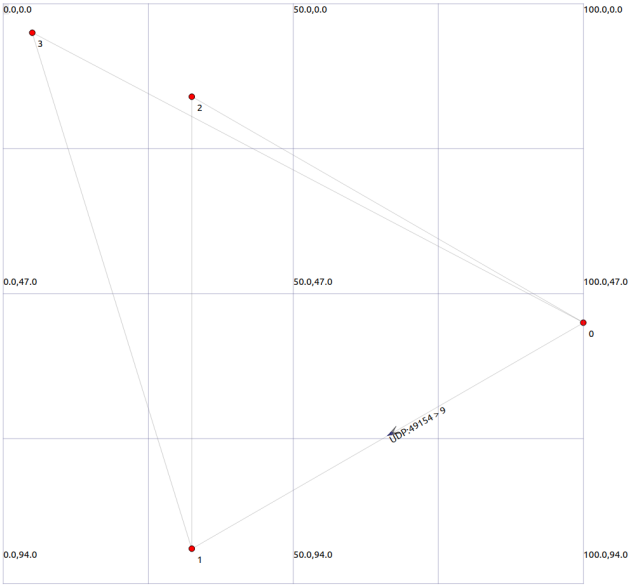


Figure 8: Visualisation of a network simulation running a compiled ProbNetKAT program.

The simulation automatically stops when there are no packets left on the network and there are no packets scheduled to be sent either. Alternatively, the simulation is run for a predetermined period of time. As ProbNetKAT has probabilistic branching, we use the random number generator provided by ns-3. Both the seed and the run numbers for this generator can be changed; a fixed seed is the default for reproducibility, but a random seed could also be used.

# 6 Conclusion

In this thesis, we have discussed the Software Defined Networking language Probabilistic NetKAT. We set out to find a way to run ProbNetKAT programs and perform probabilistic inferences on these programs. To this end, a parser has been created, and the semantics of the language has been implemented in Haskell, enabling inference on packets and programs.

In addition, a compiler has been implemented that transforms a ProbNetKAT program into an automaton. This automaton can be compiled into a C++ program that the network simulator ns-3 can run. The combination of both probabilistic and nondeterministic behavior in ProbNetKAT introduces problems for the operational semantics, which are solved by transforming the automata into a Probabilistic Normal Form. This normal form retains the semantic meaning of the original automaton but makes sure that all probabilistic choices are made before any parallel branching can occur. This normalization step is also implemented in our Haskell project. The simulated network nodes in ns-3 run an interpreter for the ProbNetKAT automaton, while the packets sent through the network are adapted UDP packets. The simulation can be visualized using the existing NetAnim ns-3 animator.

# 7 Future Work

The generated automata, both regular and normalized, may be larger than necessary. For example, generating the graph for a program $p; (a\&b); q$ would place copies of $q$ underneath both branches of the parallel composition. A more space-efficient way to handle this would merge the paths. It would be interesting to find a way to minimize these automata, especially Probability First Normal Form, as normalized graphs can become quite large. In previous work, Smolka et al. [Smo+15] have used a generalization of Binary Decision Diagrams called Forwarding Decision Diagrams to optimize their automata and the resulting forwarding tables. A similar approach might be fruitful for ProbNetKAT, perhaps using probabilistic BDD/FDDs.

Another avenue to explore would be integrating with SDN APIs such as OpenFlow, or working towards running ProbNetKAT programs on physical network hardware.

Finally, our Kleene star implementation for inference uses an approximation. Further work could involve finding a way to implement the semantics without approximation. This might be done by iterating until a fixed point is reached.

# 8 Acknowledgements

I would like to thank my thesis supervisor Henning Basold, for his patience and advice during the entire project. Also, a great thank you to Marcello Bonsangue for being my second supervisor. I have also had tremendous support from friends and family and have enjoyed the coffee breaks during the many library study sessions, for which I am very grateful. Finally, special thanks to Marleen for helping me with my planning, and getting me through the year in general.

# References


[And+14] Carolyn Jane Anderson et al. "NetKAT: semantic foundations for networks". en. In: *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. San Diego California USA: ACM, Jan. 2014, pp. 113–126. ISBN: 978-1-4503-2544-8. DOI: 10.1145/2535838.2535862. URL: https://dl.acm.org/doi/10.1145/2535838.2535862 (visited on 02/03/2023).

[Erw01] Martin Erwig. "Inductive graphs and functional graph algorithms". en. In: *Journal of Functional Programming* 11.5 (Sept. 2001). Publisher: Cambridge University Press, pp. 467–492. ISSN: 1469-7653, 0956-7968. DOI: 10.1017/S0956796801004075. URL: https://www.cambridge.org/core/journals/journal-of-functional-programming/article/inductive-graphs-and-functional-graph-algorithms/2210F7C31A34EA4CF5008ED9E7B4EF62 (visited on 12/14/2023).

[Fos+16] Nate Foster et al. "Probabilistic NetKAT". en. In: *Programming Languages and Systems*. Ed. by Peter Thiemann. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2016, pp. 282–309. ISBN: 978-3-662-49498-1. DOI: 10.1007/978-3-662-49498-1_12.

[Jac21] Bart Jacobs. "From Multisets over Distributions to Distributions over Multisets". In: *2021 36th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. Rome, Italy: IEEE, June 2021, pp. 1–13. ISBN: 978-1-66544-895-6. DOI: 10.1109/LICS52264.2021.9470678. URL: https://ieeexplore.ieee.org/document/9470678/ (visited on 12/21/2023).

[Kah17] David M. Kahn. "Undecidable Problems for Probabilistic Network Programming". en. In: (2017). Artwork Size: 17 pages Medium: application/pdf Publisher: Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik GmbH, Wadern/Saarbruecken, Germany, 17 pages. DOI: 10.4230/LIPICS.MFCS.2017.68. URL: http://drops.dagstuhl.de/opus/volltexte/2017/8096/ (visited on 12/13/2023).

[Kle20] Achim Klenke. *Probability Theory: A Comprehensive Course*. en. Universitext. Cham: Springer International Publishing, 2020. ISBN: 978-3-030-56401-8 978-3-030-56402-5. DOI: 10.1007/978-3-030-56402-5. URL: https://link.springer.com/10.1007/978-3-030-56402-5 (visited on 12/19/2023).

[Sci23] Adam Scibior. *Monad-Bayes*. Oct. 2023. URL: https://hackage.haskell.org/package/monad-bayes.

[Smo+15] Steffen Smolka et al. "A fast compiler for NetKAT". en. In: *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*. Vancouver BC Canada: ACM, Aug. 2015, pp. 328–341. ISBN: 978-1-4503-3669-7. DOI: 10.1145/2784731.2784761. URL: https://dl.acm.org/doi/10.1145/2784731.2784761 (visited on 12/13/2023).

[Smo+18] Steffen Smolka et al. *Probabilistic Program Equivalence for NetKAT*. arXiv:1707.02772 [cs]. Mar. 2018. URL: http://arxiv.org/abs/1707.02772 (visited on 12/13/2023).

[Smo+19] Steffen Smolka et al. "Scalable verification of probabilistic networks". In: *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2019. New York, NY, USA: Association for Computing Machinery, June 2019, pp. 190–203. ISBN: 978-1-4503-6712-7. DOI: 10.1145/3314221.3314639. URL: https://doi.org/10.1145/3314221.3314639 (visited on 03/15/2023).

[Sok11] Ana Sokolova. "Probabilistic systems coalgebraically: A survey". en. In: *Theoretical Computer Science* 412.38 (Sept. 2011), pp. 5095–5110. ISSN: 03043975. DOI: 10.1016/j.tcs.2011.05.008. URL: https://linkinghub.elsevier.com/retrieve/pii/S0304397511003902 (visited on 12/21/2023).

[VS19] Alexander Vandenbroucke and Tom Schrijvers. "PNK: functional probabilistic NetKAT". In: *Proceedings of the ACM on Programming Languages* 4 (Dec. 2019), pp. 1–27. DOI: 10.1145/3371107.