Harnessing Probabilistic Programming for Network Problems

Alexander Vandenbroucke

What do I work on?

Programming Languages:
Practice and Theory

KU LEUVEN

What do I work on?

functional programming

Tabling-monad in Haskell

logic programming

Tabling with sound answer-subsumption

probabilistic programming

$P\lambda\omega$NK: Functional Probabilistic NetKAT

What do I work on?

functional programming

Tabling-monad in Haskell

logic programming

Tabling with sound answer-subsumption

probabilistic programming

P$\lambda\omega$NK: Functional Probabilistic NetKAT

P$\lambda\omega$NK

Network hardware is expensive

Mistakes are expensive

security breaches, downtime, …

And

network protocols are **hard** to get right

P$\lambda\omega$NK

Can we model and predict the behaviour of networks in software?

Including probabilistic behaviour?

P$\lambda\omega$NK

We can, but it's a pain with existing languages.

```
in =
  SW ← 0; PT ← 0;

t =
  (
  (SW = 0; PT = 0); SW ← 0; PT ← 0
  &
  (SW = 0; PT = 1); SW ← 1; PT ← 0
  &
  (SW = 0; PT = 2); SW ← 2; PT ← 0
  &
  (SW = 0; PT = 4); SW ← 4; PT ← 0
  &
  …
```

NetKAT

high-level network

low-level programming

P$\lambda\omega$NK

We can, but it's a pain with existing languages.

Solution: apply programming language techniques

lambda-abstraction

Challenges: theoretical and practical

side-effects probabilities higher-order functions

language-design implementation

Overview

I. **Probabilistic Programming**
II. **NetKAT**
III. **P$\lambda\omega$NK**
IV. **Conclusions**

Part I: Probabilistic Programming

A New Paradigm

Probabilistic Programming

MODEL + INFERENCE

Pose Reconstruction,
Information Retrieval,
Genetics,
Seismographic Data

use real-world data

Probabilistic Programming

Domain

↓ domain expert

Model

↓ statistician

Inference Algorithm

↓ programmer

Inference Program

Probabilistic Programming

PPL = MODEL + REUSABLE INFERENCE

goal: make probabilistic inference easier, more reusable, less error prone, …

.. by cutting out the middle men

domain expert ~~statistician~~ ~~programmer~~

Terminology

Statistical Processes

throwing dice, tossing coins, assigning seat numbers, temperature, …

Events

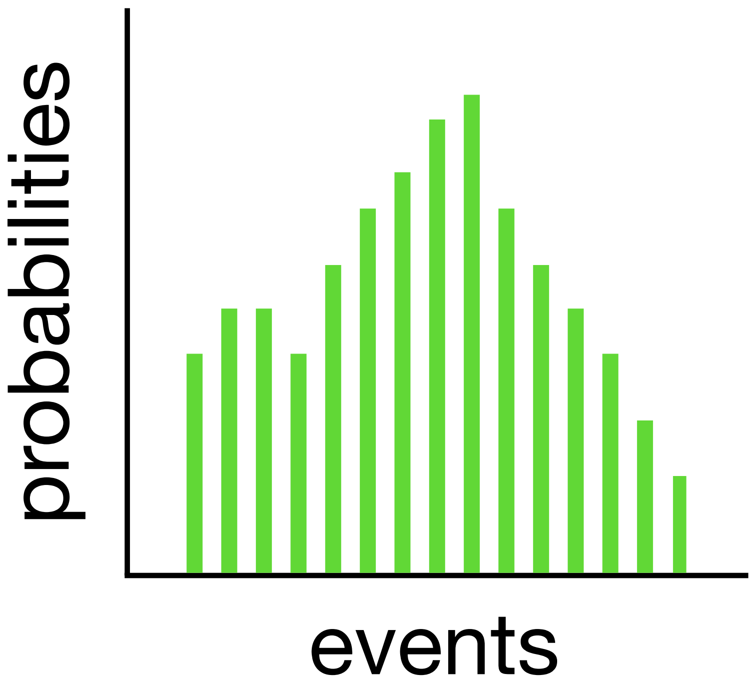
1… 6 eyes; heads or tails, a seat assignment, a temperature, …

Probability Distribution

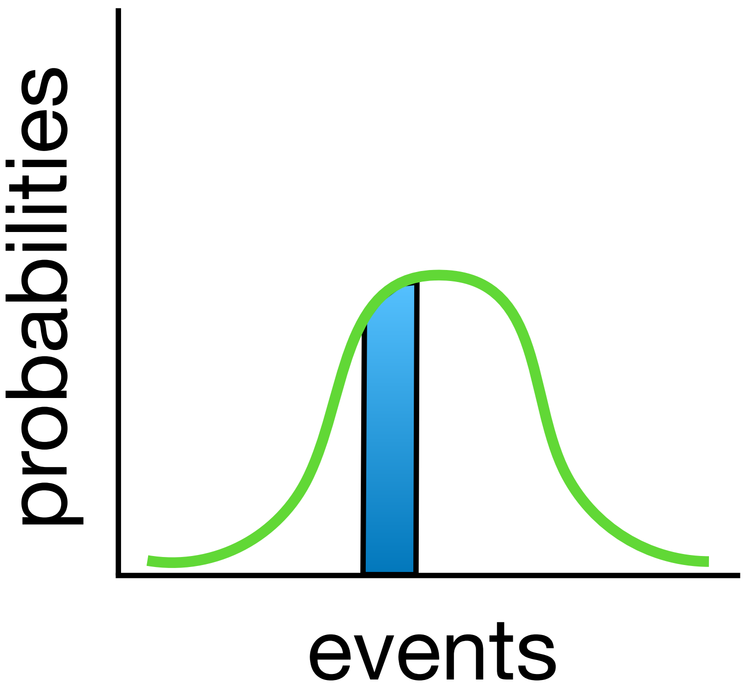
$$\mathbb{P} : (S \subseteq \text{Events}) \rightarrow [0,1]$$

$$\mathbb{P}(heads) = 0.5 \quad \mathbb{P}(tails) = 0.5$$

Discrete vs. Continous



individual events have weight



no individual events have weight, but sets do!

Warm Up

```haskell
data Coin = H | T

coin :: Double → Dist Coin
```

```
> run (coin 0.5)
T ====================.................... 50%
H ====================.................... 50%
```

Warm Up

```
data Coin = H | T
coin :: Double → Dist Coin

twoCoins :: Dist (Coin,Coin)
twoCoins = do
  x ← coin 0.5
  y ← coin 0.4
  return (x,y)
```

has type Double

has type Dist Double

Warm Up

```
data Coin = H | T
coin :: Double → Dist Coin

twoCoins :: Dist (Coin,Coin)
twoCoins = do
  x ← coin 0.5
  y ← coin 0.4
  return (x,y)
```

has type Double

has type Dist (Double,Double)

```
> run twoCoins
(T,T) ============.......................... 30.0%
(T,H) ========.............................. 20.0%
(H,T) ============.......................... 30.0%
(H,H) ========.............................. 20.0%
```

Warm Up

```
data Coin = H | T
coin :: Double → Dist Coin

twoHeads :: Dist Bool
twoHeads = do
  (x,y) ← twoCoins
  return (x == H || y == H)
```

has type Dist Bool

has type Bool

```
> run twoHeads
True  ============================............ 70.0%
False ============............................ 30.0%
```

Discrete Example

```
trail :: Double → Int → Dist Int
trail p n = do
  outcomes ← replicateM n (coin p)
  let count x = length . filter (== x)
  return (count H outcomes)
```

has type [Coin]

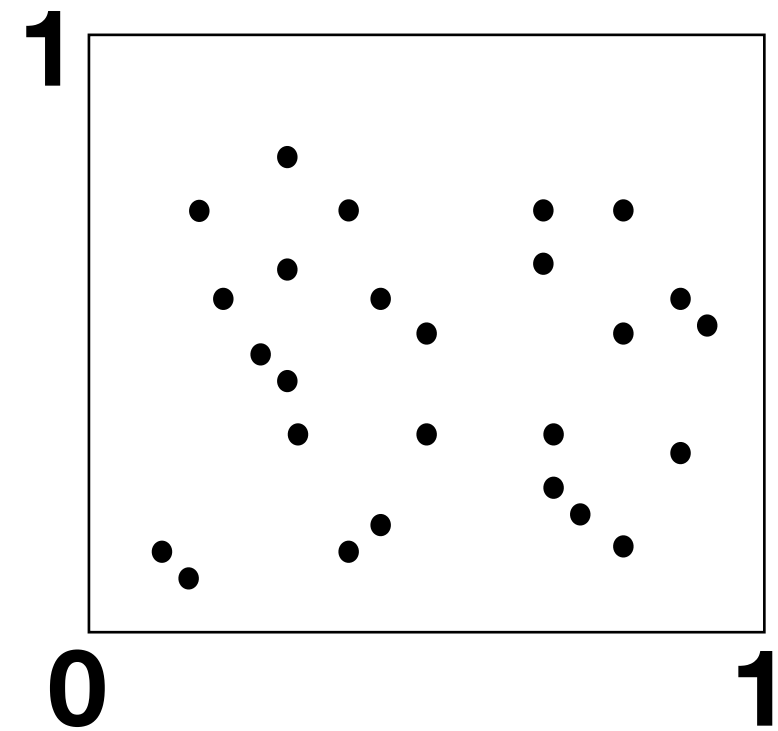
has type Coin → [Coin] → Int

```
> run (trail 0.5 4)
4 ===................................. 6.25%
3 ==========.......................... 25.0%
2 ===============..................... 37.5%
1 ==========.......................... 25.0%
0 ===................................. 6.25%
```

Continuous Example

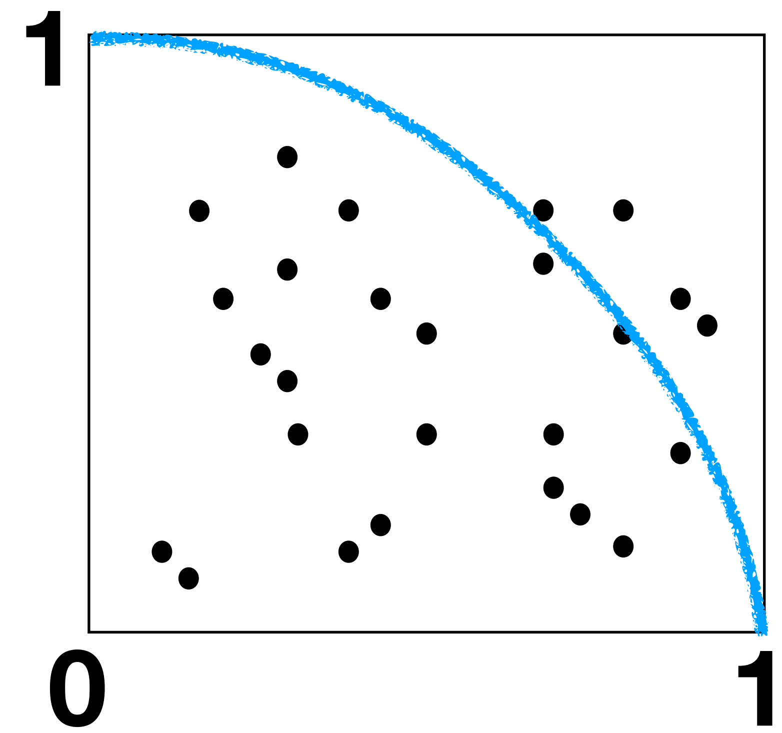
```
x ↜ uniform(0,1)
y ↜ uniform(0,1)
if sqrt(x*x + y*y) ⩽ 1:
  return 1
else:
  return 0
```

Hakaru



Continuous Example

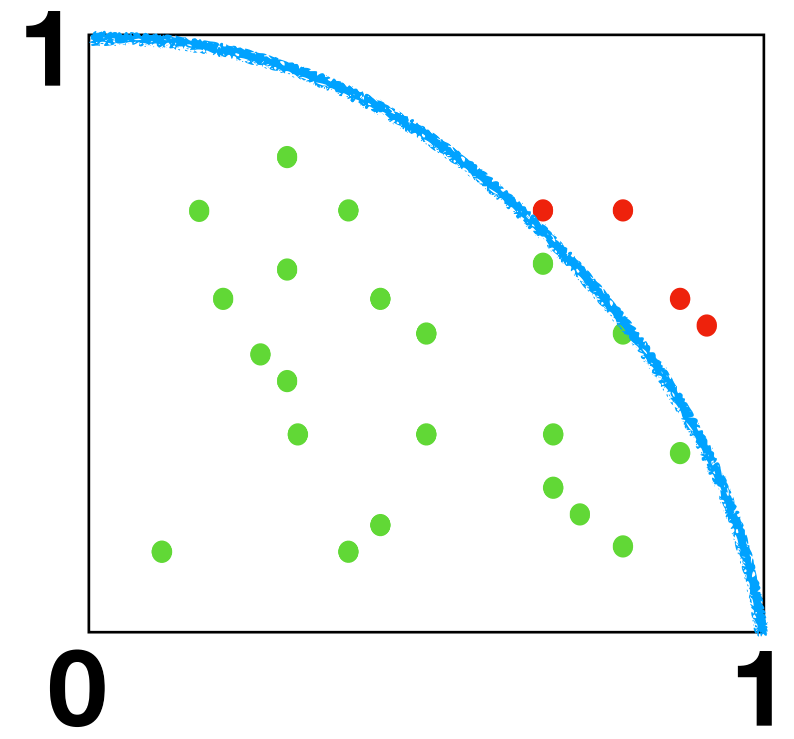
```
x ↜ uniform(0,1)
y ↜ uniform(0,1)
if sqrt(x*x + y*y) ⩽ 1:
  return 1
else:
  return 0
```



Hakaru

Continuous Example

```
x ↜ uniform(0,1)
y ↜ uniform(0,1)
if sqrt(x*x + y*y) ⩽ 1:
  return 1
else:
  return 0
```



Hakaru

$$E = \frac{1}{N}\sum \bullet = \frac{1}{N}\sum {\color{green}\bullet} \approx \frac{\pi}{4}$$

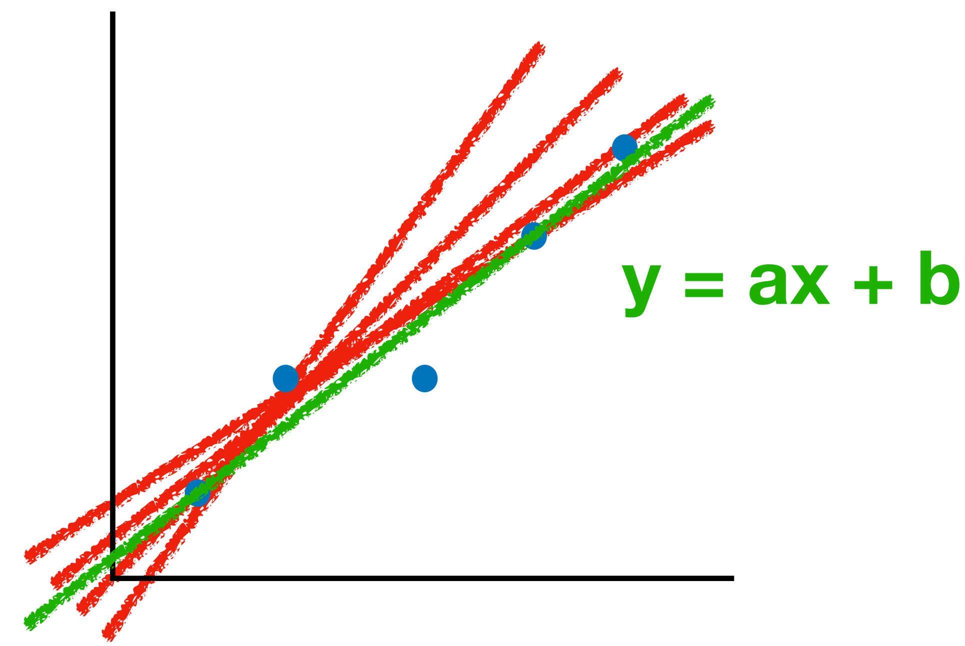
Linear Interpolation

```
a ↜ normal(0,3)
b ↜ normal(0,3)
line = fn x real: a * x + b

xs = [1,2,3,4,5]

fuzzy_ys ↜ plate i of 5:
  normal(line(xs[i]),0.5)

return(fuzzy_ys,(a,b))
```



X	1	2	3	4	5
Y	2.5	3.8	4.5	6.2	8.0

What is **(a,b)** given the **data**?

Applications

Languages

Stan, **MonadBayes**, **Pyro**, Anglican,
Hakaru, Edward, ProbLog, Turing, …

Applications

Pose Reconstruction, Information Retrieval, Genetics,
Seismographic Data (for the military)

Algorithms

MH, **SMC**, **HMC,….** ⟶ these are **hard**

let's take a step back

Part II:
NetKAT

The Network Strikes Back

Overview

I. Probabilistic Programming
II. NetKAT
III. P$\lambda\omega$NK
IV. Conclusions

Network Modelling

Network hardware is expensive

Mistakes are expensive

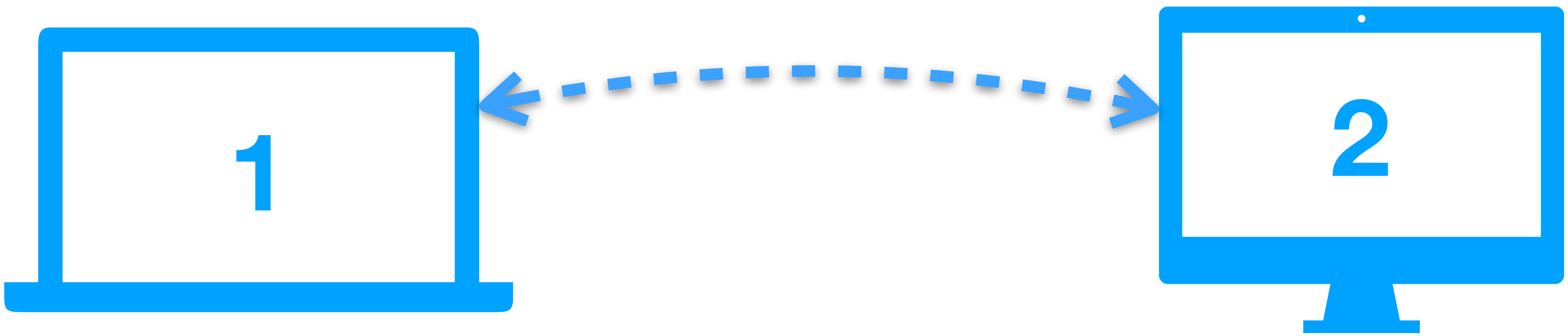
security breaches, downtime, …

And

network protocols are **hard** to get right

predict in software

Example



$$(\mathtt{SW} = 1;\ \mathtt{SW} \leftarrow 2)\ \&\ (\mathtt{SW} = 2;\ \mathtt{SW} \leftarrow 1)$$

if node 1 — send to node 2 — if node 2 — send to node 1

Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, David Walker:
NetKAT: semantic foundations for networks. POPL 2014: 113-126

Packet Trace Transformer

IN

Sets of Packet Traces

NetKAT

OUT

Sets of Packet Traces

Notation

`{[(SW: 1, PT: 2),(SW: 1, PT: 3)]}`

packet packet

trace

set

Guards

```
S          --skip-->      S

S          --drop-->      ∅

{[(SW: 1)],[(SW: 2)]}  --SW = 1-->  {[(SW: 1)]}
```

Modification

PT ← 2

{[(SW: 1, PT: 1)]} → {[(SW: 1,PT: 2)]}

SW ← 2

{[(SW: 1),(SW: 1)]} → {[(SW: 2),(SW: 1)]}

SW ← 2

{[(SW: 1)],[(SW: 2)]} → {[(SW: 2)]}

Duplication

```
                  dup
{[(SW: 1)]}    ------->    {[(SW: 1),(SW: 1)]}
```

Sequence

```
                  SW = 1                       SW ← 2
{[(SW: 1)],
 [(SW: 3)]}        →        {[(SW: 1)}         →        {[(SW: 2)}
```

```
                              SW = 1;
                              SW ← 2
{[(SW: 1)],[(SW: 3)]}        →        {[(SW: 2)}
```

Parallel

```
  SW = 1
& SW = 2
```

```
{[(SW: 1)],          {[(SW: 1)],
 [(SW: 2)]}    →      [(SW: 2)]}
```

```
            SW = 1     {[(SW: 1)]}
{[(SW: 1)],
 [(SW: 2)]}                          ∪ = {[(SW: 1)],
                                          [(SW: 2)]}
            SW = 2     {[(SW: 1)]}
```

Parallel (cont.)

(SW = 1; SW ← 2) & (SW = 2; SW ← 1)

{[(SW: 1)]} → {[(SW: 2)]}

Iteration

```
( SW = 1; SW ← 2
& SW = 2; SW ← 3)*
```

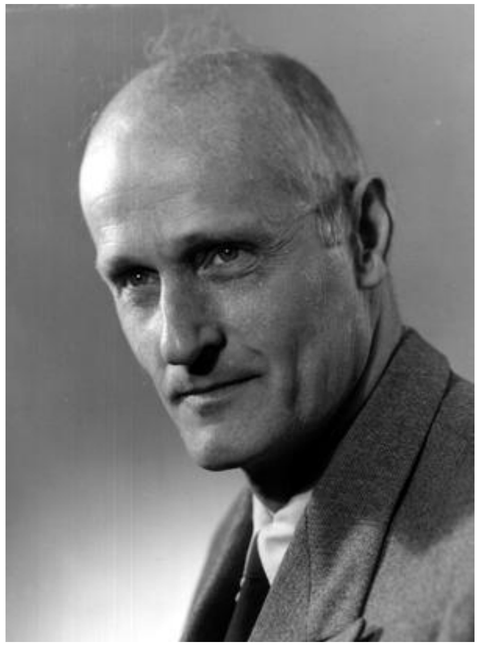
```
{[(SW: 1)]}  ──────▶  {[(SW: 1)],
                       [(SW: 2)],
                       [(SW: 3)]}
```

```
e* = skip & (e*;e)
```

NetKAT = Net + KAT

KAT = Kleene Algebra + Test

same **Kleene** as regular expressions



NetKAT = Net + KAT

KAT = Kleene Algebra + Test

logic theory ($\supseteq$ Hoare logic)	make proofs
Kleene theorem: automata	verification by simulation e.g. termination = no routing loops
	compilation to routing tables SDN

Probabilistic NetKAT

IN

sets of packet traces

Probabilistic NetKAT →

OUT

distributions of sets of packet traces

Choice

```
SW = 1 <0.4> SW = 2
```

```
{[(SW: 1)],                 0.4:{[(SW: 1)]}
 [(SW: 2)]}      ──────▶    0.6:{[(SW: 2)]}
```

& is not idempotent

```
SW = 1 <0.5> SW = 2
```

```
{[(SW: 1)],
 [(SW: 2)]}
```

→

```
0.5:{[(SW: 1)]}
0.5:{[(SW: 2)]}
```

```
 (SW = 1 <0.5> SW = 2)
&(SW = 1 <0.5> SW = 2)
```

```
{[(SW: 1)],
 [(SW: 2)]}
```

→

```
0.25:{[(SW: 1)]}
0.25:{[(SW: 2)]}
0.5 :{[(SW: 1)],
      [(SW: 2)]}
```

Prob. NetKAT $\neq$ Net + KAT

~~logic theory $\supseteq$ Hoare logic~~

~~Kleene theorem: automata~~

~~make proofs~~

~~verification by simulation~~

~~compilation to routing tables~~

What *can* we do?

approximation by iteration

approximate probabilities

---

verification by exact probabilistic inference (without dup)

discrete distribution

decidable equivalence

Why?

faults and failures

e.g. probability of delivery

traffic approximation

e.g. expected latency

probabilistic protocols

e.g. correct routing

Example

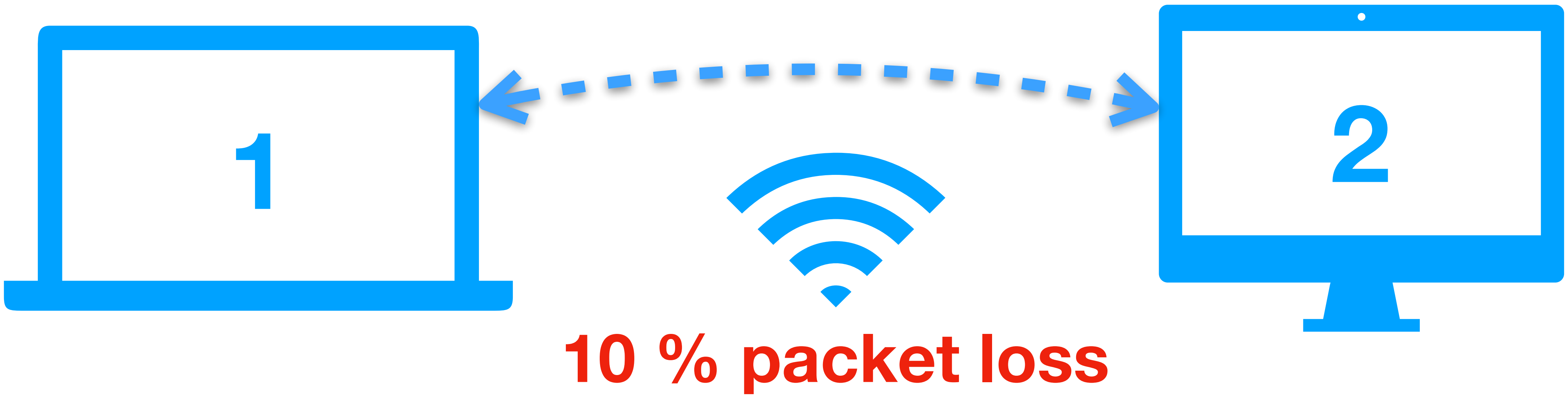
10 % packet loss

```
(SW = 1; SW ← 2 <0.9> drop) & (SW = 2; SW ← 1 <0.9> drop)
```

if node 1	send to node 2	or drop packet	if node 2	send to node 1	or drop packet
	90%	10%		90%	10%

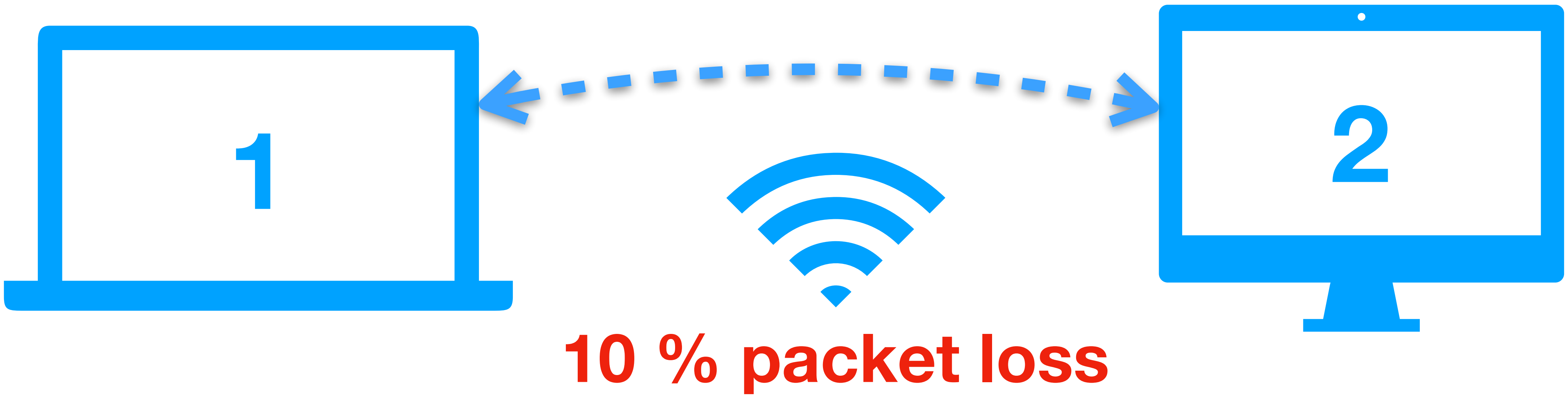
Nate Foster, Dexter Kozen, Konstantinos Mamouras, Mark Reitblatt, Alexandra Silva: Probabilistic NetKAT. ESOP 2016: 282-309

Functions



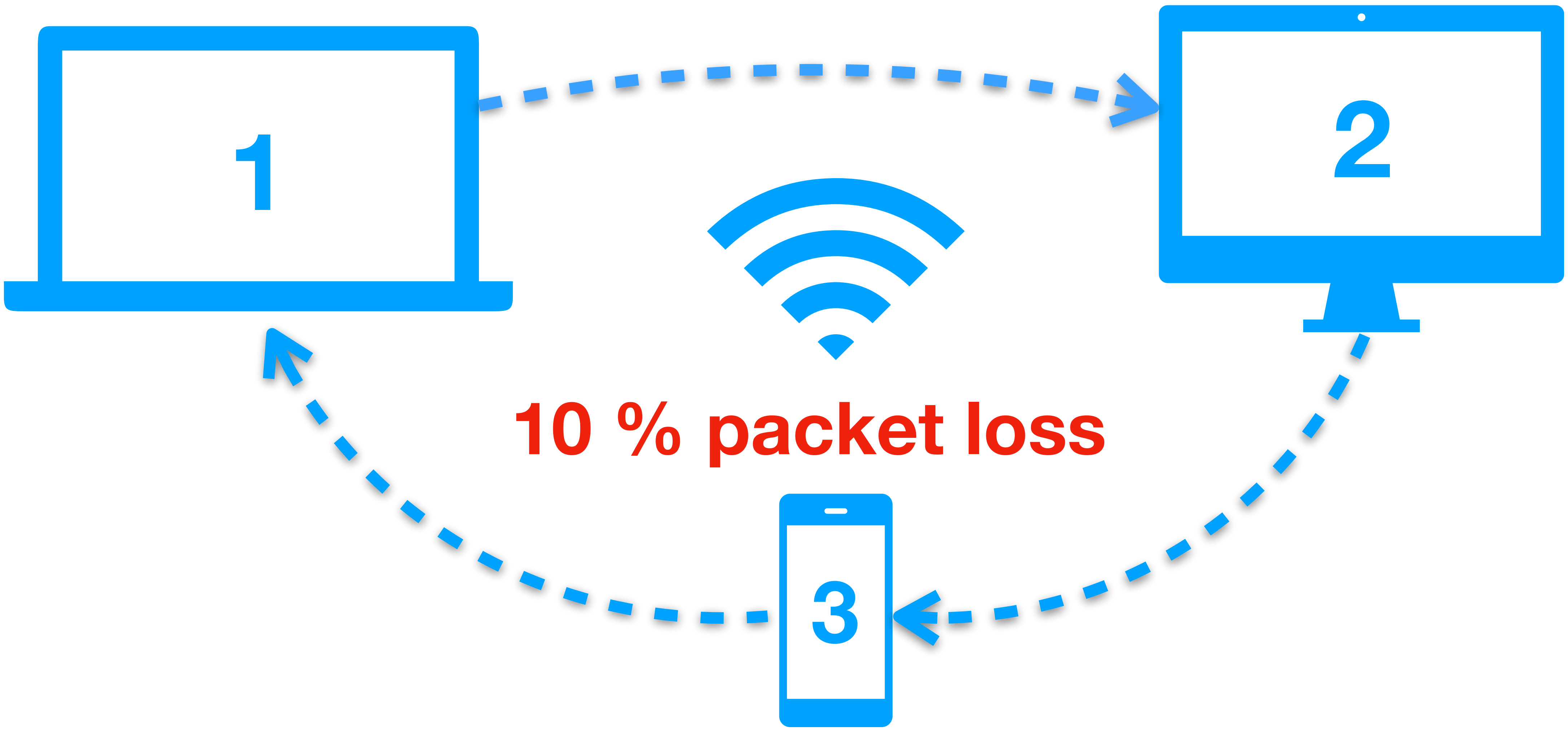

```
forward = λsrc.λdst.SW = src; SW ← dst <0.9> drop
(SW = 1; SW ← 2 <0.9> drop) & (SW = 2; SW ← 1 <0.9> drop)
```

Functions



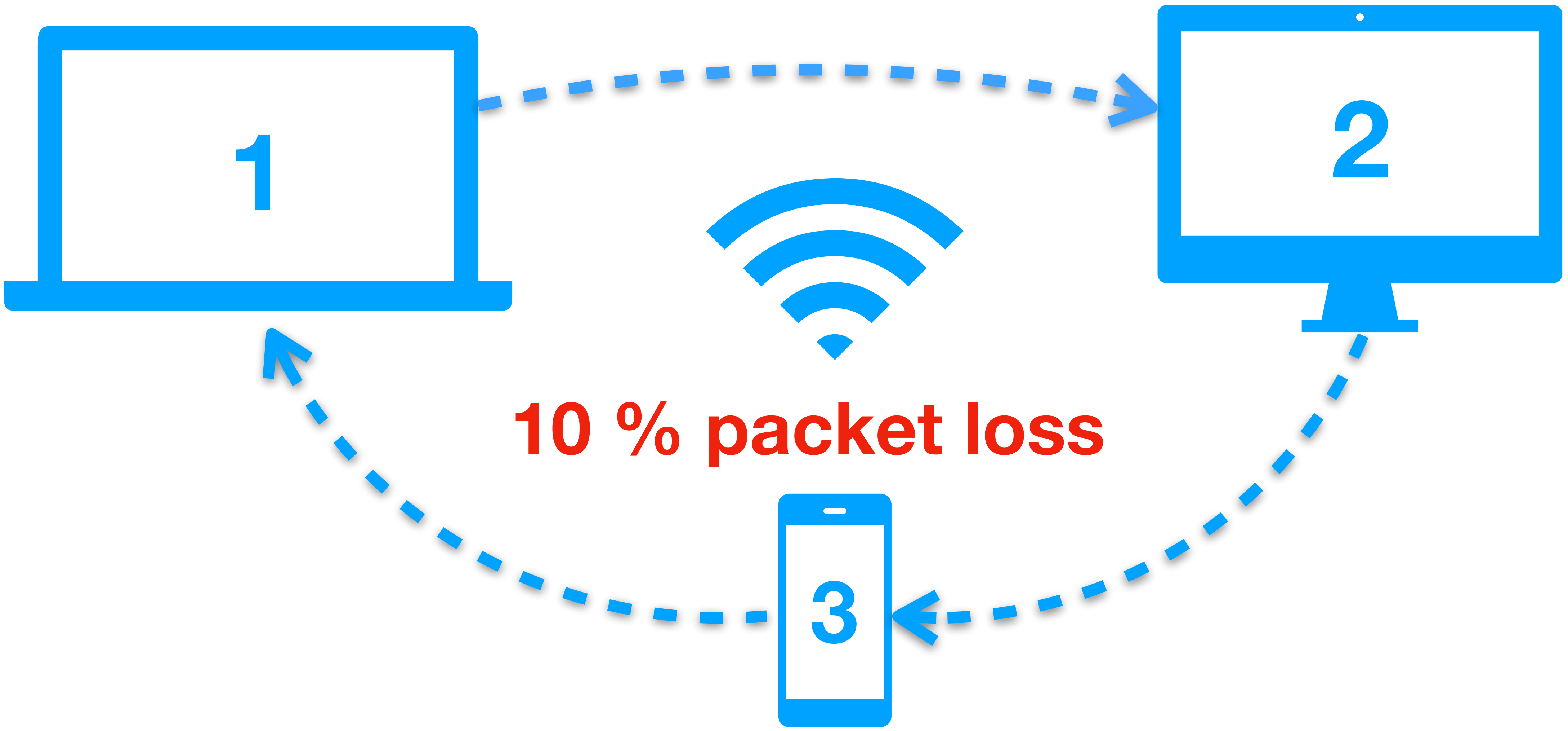

```
forward = λsrc.λdst.SW = src; SW ← dst <0.9> drop
forward 1 2 & forward 2 1
```

Functions




```
  (SW = 1; SW ← 2 <0.9> drop)
& (SW = 2; SW ← 3 <0.9> drop)
& (SW = 3; SW ← 1 <0.9> drop)
```

Functions



`forward 1 2 & forward 2 3 & forward 3 1`

Part III:

$P\lambda\omega NK$

Return of the Lambda

Overview

I. Probabilistic Programming
II. NetKAT
III. P$\lambda\omega$NK
IV. Conclusions

Challenge I:
Functions & Side-effects

```
(λx.SW = 1) (SW ← 1)
```

test if SW is 1

set SW to 1

Challenge I:
Functions & Side-effects

Call-By-Name

`(λx.SW = 1) (SW ← 1)`

`{[(SW: 0)]}` ⟶ ∅

Call-By-Value

`(λx.SW = 1) (SW ← 1)`

`{[(SW: 0)]}` ⟶ `{[(SW:1)]}`

Challenge I:
Functions & Side-effects

Call-By-Name

`(λx.SW = 1) (SW ← 1)`

`{[(SW: 0)]}` ⟶ ∅

Call-By-Value

`(λx.SW = 1) (SW ← 1)`

`{[(SW: 0)]}` ⟶ `{[(SW:1)]}`

Solution I: Fine-Grained Call-By-Value

Call-By-Name

`(λx.SW = 1) (λx.SW ← 1)`

`{[(SW: 0)]}` ⟶ ∅

Call-By-Value

`SW ← 1 to y.(λx.SW = 1) y`

`{[(SW: 0)]}` ⟶ `{[(SW:1)]}`

Paul Blain Levy. 2001. Call-by-push-value. Ph.D. Dissertation. Queen Mary University of London, UK.

Challenge II:
Higher-order Functions

```
(λf.f 1) <0.5> (λf.f 2)
```

```
{[(SW:0)]}  →  0.5:{
                 (g,[(SW:0)])
               }
               0.5:{
                 (h,[(SW:0)])
               }
```

where f and g are **higher-order** functions

Challenge II: Higher-order Functions

```
0.5:{
   (f,[(SW:0)])
}
0.5:{
   (g,[(SW:0)])
}
```

a **probability distribution** over **higher-order** functions

… a **continuous** distribution

Measure Theory

Solution II:
QBS

Measure Theory

not **cartesian-closed**

Quasi-Borel Spaces

are **cartesian-closed**

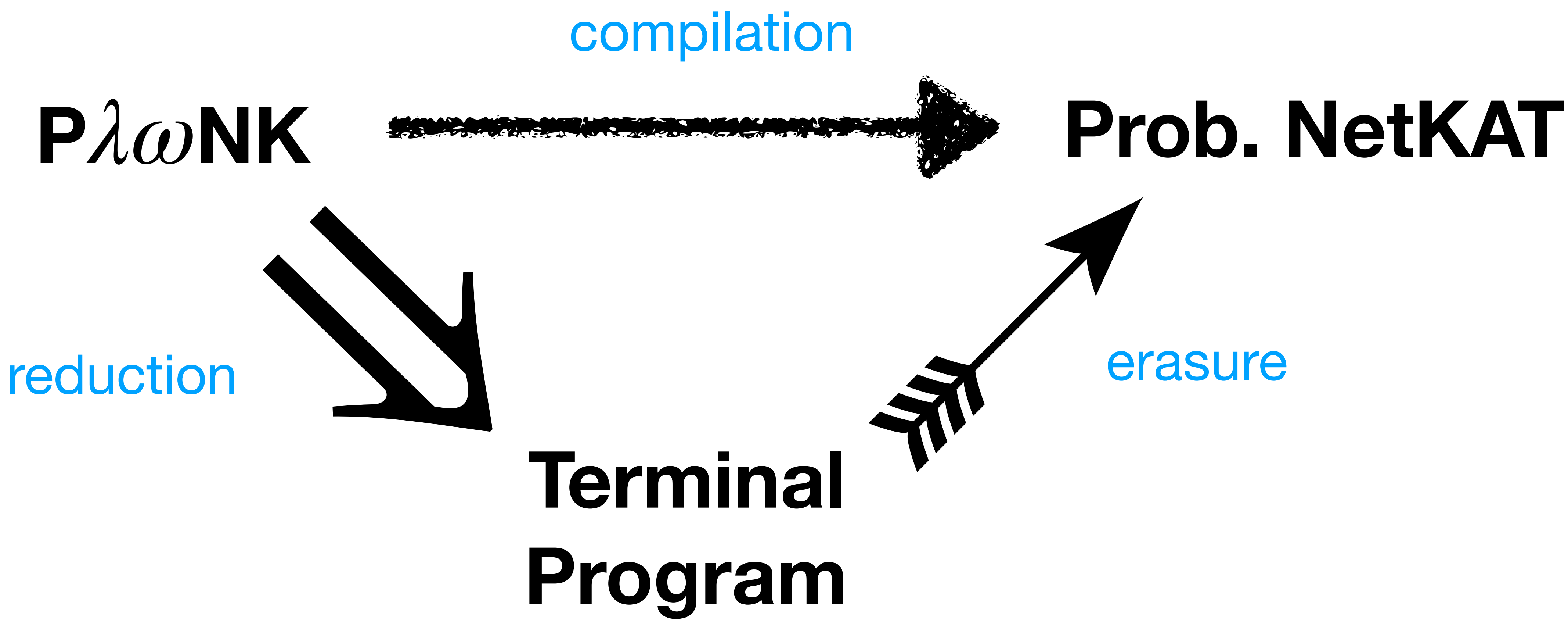
… but the maths are considerably more complicated

brand new!

Chris Heunen, Ohad Kammar, Sam Staton, and Hangseok Yang. 2017. A convenient category for higher-order probability theory. In LICS. IEE Computer Society, 1-12

Mathijs Vákár, Ohad Kammar, and Sam Staton. 2019. A domain theory for statistical probabilistic programming PACMPL 3, POPL (2019), 36:1-36:29

Challenge III: Compilation



Challenge III: Compilation

non-termination

`λf.(λx.f (x x) (λx.f (x x)))` ⇛ ?

unsoundness

`(λx.SW = 1) & (λx.SW = 2)` ⇛ ?

Solution III: Typing

$$G \vdash E : T$$

context expression type

Solution III: Typing

```
G ⊢ E : T
```

simple types

`λf.(λx.f (x x) (λx.f (x x)))` is ill-typed

strong normalisation = termination

not Turing-complete … but this is a **modelling language**

Solution III: Typing

$$G \vdash E : T$$

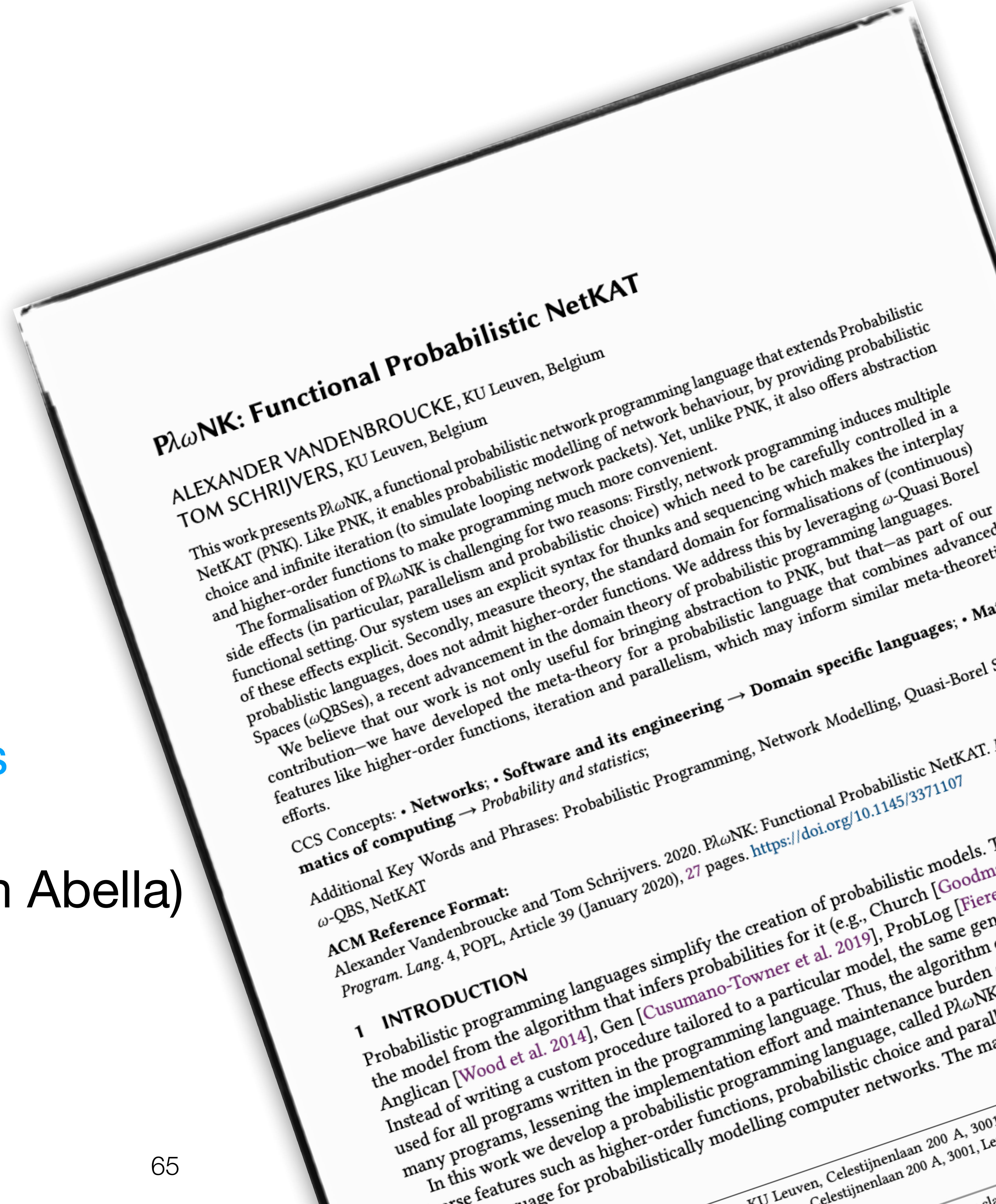
parallel type

if G ⊢ A : unit and G ⊢ B : unit
then G ⊢ A & B : unit

(λx.SW = 1) & (λx.SW = 2) is ill-typed

A & B can only produce **unit** values … like NetKAT

> In the paper:
> > background,
> > denotational semantics
> > compilation procedure (partially *mechanised* in Abella)


PλωNK: Functional Probabilistic NetKAT

ALEXANDER VANDENBROUCKE, KU Leuven, Belgium
TOM SCHRIJVERS, KU Leuven, Belgium

This work presents PλωNK, a functional probabilistic network programming language that extends Probabilistic NetKAT (PNK). Like PNK, it enables probabilistic modelling of network behaviour, by providing probabilistic choice and infinite iteration (to simulate looping network packets). Yet, unlike PNK, it also offers abstraction and higher-order functions to make programming much more convenient.

The formalisation of PλωNK is challenging for two reasons: Firstly, network programming induces multiple side effects (in particular, parallelism and probabilistic choice) which need to be carefully controlled in a functional setting. Our system uses an explicit syntax for thunks and sequencing which makes the interplay of these effects explicit. Secondly, measure theory, the standard domain for formalisations of (continuous) probabilistic languages, does not admit higher-order functions. We address this by leveraging ω-Quasi Borel Spaces (ωQBSes), a recent advancement in the domain theory of probabilistic programming languages.

We believe that our work is not only useful for bringing abstraction to PNK, but that—as part of our contribution—we have developed the meta-theory for a probabilistic language that combines advanced features like higher-order functions, iteration and parallelism, which may inform similar meta-theoreti efforts.

CCS Concepts: • Networks; • Software and its engineering → Domain specific languages; • Ma matics of computing → Probability and statistics;

Additional Key Words and Phrases: Probabilistic Programming, Network Modelling, Quasi-Borel S ω-QBS, NetKAT

ACM Reference Format:
Alexander Vandenbroucke and Tom Schrijvers. 2020. PλωNK: Functional Probabilistic NetKAT. *Program. Lang.* 4, POPL, Article 39 (January 2020), 27 pages. https://doi.org/10.1145/3371107

1 INTRODUCTION

Probabilistic programming languages simplify the creation of probabilistic models. T the model from the algorithm that infers probabilities for it (e.g., Church [Goodm Anglican [Wood et al. 2014], Gen [Cusumano-Towner et al. 2019], ProbLog [Fiere Instead of writing a custom procedure tailored to a particular model, the same gen used for all programs written in the programming language. Thus, the algorithm many programs, lessening the implementation effort and maintenance burden In this work we develop a probabilistic programming language, called PλωNK rse features such as higher-order functions, probabilistic choice and paral uage for probabilistically modelling computer networks. The ma

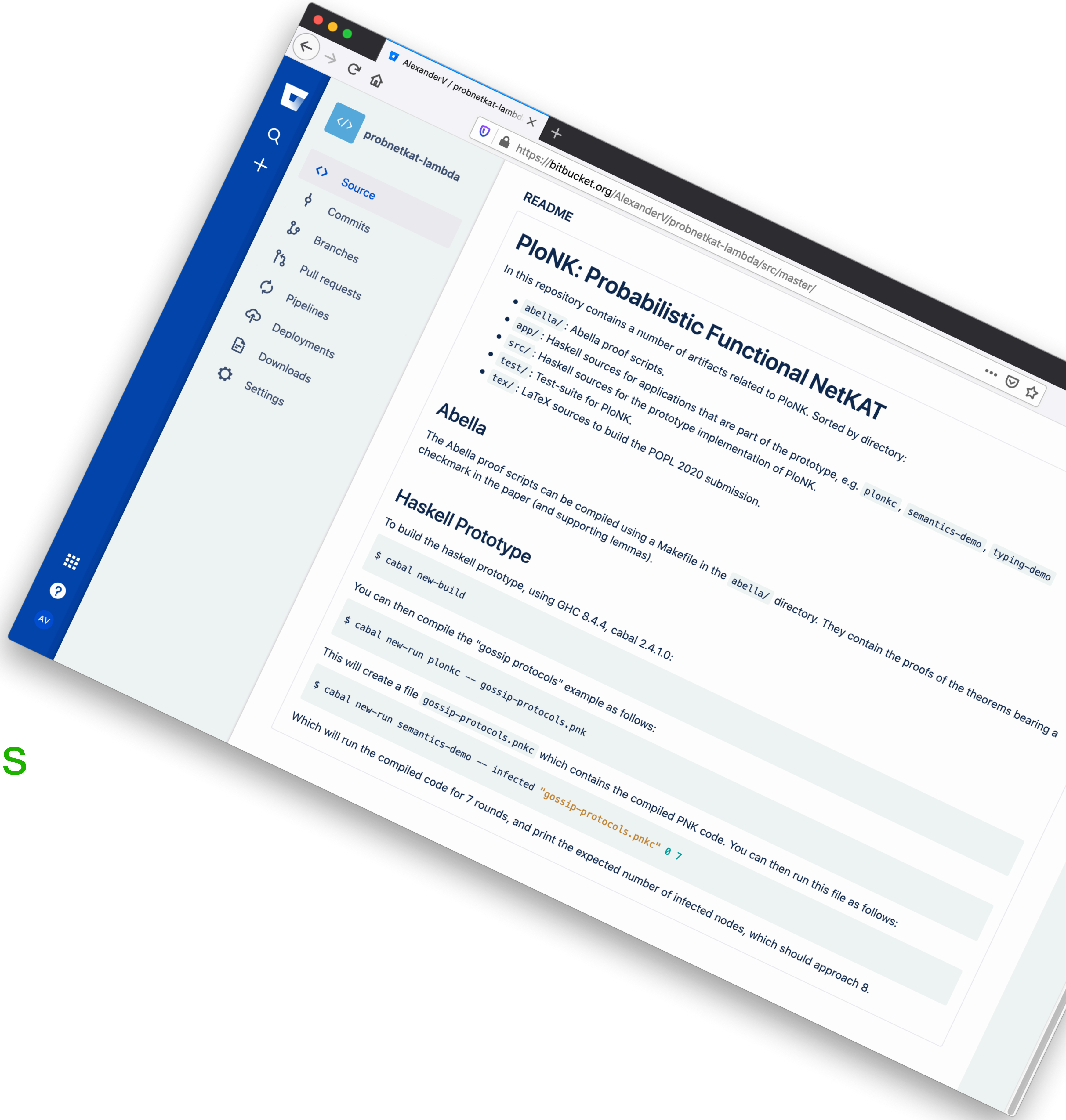
KU Leuven, Celestijnenlaan 200 A, 3001
Celestijnenlaan 200 A, 3001, Le

> On bitbucket

> prototype implementation,

> examples

> Abella proof scripts



Conclusions

Overview

I. **Probabilistic Programming**
II. **NetKAT**
III. **P$\lambda\omega$NK**
IV. **Conclusions**

Conclusions

- **Networks are difficult to predict**
- **model them in a software language**
- **modelling-language design is challenging**
- **language and software engineering require a good grasp of theoretical and practical concepts**

Thank You for Listening!

```
λquestion.
  dst ← answer question <0.1> panic
panic = drop*
```