Probabilistic Programming from the Ground Up

Lecture 1: Why Probabilistic Programming

June 10, 2024

Steven Holtzen

s.holtzen@northeastern.edu

Oregon Programming Languages Summer School 2024

https://www.khoury.northeastern.edu/home/sholtzen/oplss24-ppl/

Goals of this course:

To be able to design, implement, and use your own probabilistic programming language

Secondary objective: get a "bird's eye view" of the field of probabilistic programming

Course overview

- Lecture 1: Syntax, semantics, implementation of TinyPPL and TinyCond
- Lecture 2: Approximate reasoning via sampling
  - Direct sampling and TinySamp
  - Rejection sampling
  - Markov-Chain Monte-Carlo sampling (?)
- Lecture 3: Tractability and expressivity
  - Binary decision diagrams (BDD)
  - Inference via weighted model counting
- Lecture 4: Markov-Chain Monte Carlo, perspectives

Course logistics

- All course content available on course webpage
- All implementations provided in Ocaml, available here: https://github.com/SHoltzen/oplss24-ppl
- You are encouraged to follow along with the implementation and build on these interpreters to make your own languages
- This course is a condensed version of a few other courses I have taught:
  - https://neuppl.github.io/CS7470-Fall23/
  - https://www.khoury.northeastern.edu/home/sholtzen/CS7480Fall21/

Course philosophy

- Implementation-oriented: you should play with the provided implementations (maybe find bugs?)
  - Improve them!
  - Add features to them!

- All the implementations are in pure OCaml, single-file. You can run them in your browser without installing OCaml if you want: https://try.ocamlpro.com/

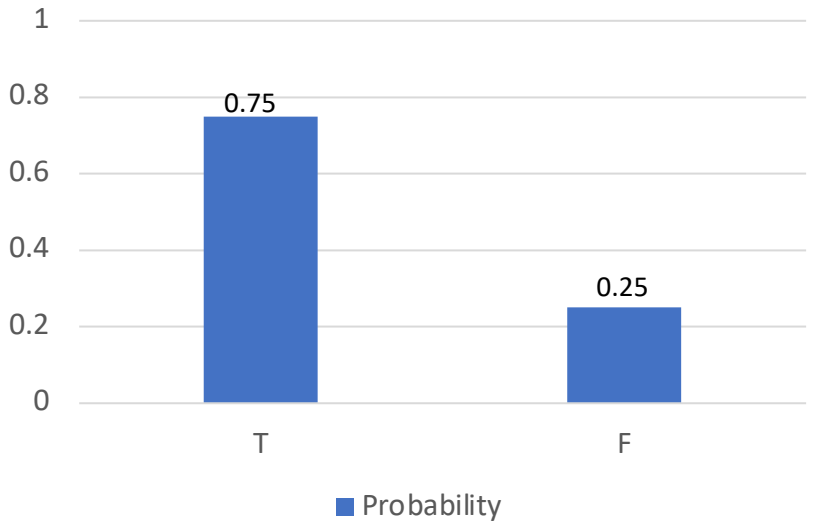
- Think of these slides like instructor notes; they supplement the course, but they are not actually used to present

Q: What are probabilistic programs?

A: Programs that denote probability distributions

Syntax for introducing uncertainty:
True with probability 1/2

$$[\![\; x \leftarrow \texttt{flip}\ 1/2; \quad y \leftarrow \texttt{flip}\ 1/2; \quad \texttt{return}\ x \vee y \;]\!] =$$



Why probabilistic programming languages?

The PL Argument

for probabilistic programming

1. **It is important to verify programs.**

"Program testing can be used to show the presence of bugs, but never to show their absence."

E. Dijkstra

1. It is important to verify programs.
2. **Many kinds of programs have inherent probabilistic uncertainty.**

The Power and Weakness of Randomness in Computation

Avi Wigderson

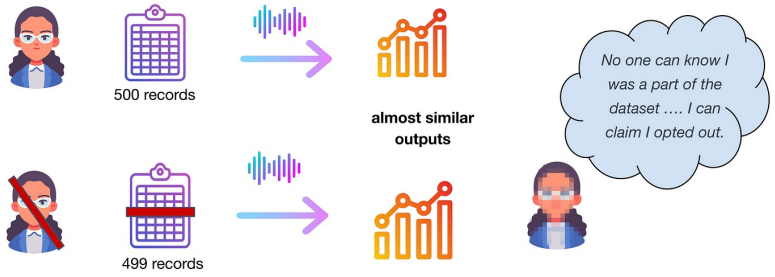
Institute for Advanced Study, Princeton

Randomness is Paramount to Computational Efficiency. The use of randomness can dramatically enhance computation (and do other wonders) for a variety of problems and settings. In particular, examples will be given of probabilistic algorithms (with tiny error) which are exponentially faster than their (best known) deterministic counterparts, and probabilistic algorithms which achieve significant space savings over deterministic ones. Other settings include distributed algorithms where randomness (provably) achieves exponentially smaller congestion than deterministic ones. Finally we'll show that using randomness, proof systems can be enhanced to allow properties unattainable without it. Letting the verifier and prover toss coins, proof systems can allow spot checking of proofs (PCPs - a central tool in the theory of approximation), as well as zero-knowledge proofs (proofs revealing nothing except their validity - a central tool in cryptography).

2023 ACM A.M. Turing Award Laureate

1. It is important to verify programs.
2. **Many kinds of programs have inherent probabilistic uncertainty.**

Differential Privacy



https://medium.com/dsaid-govtech/protecting-your-data-privacy-with-differential-privacy-an-introduction-abee1d7fcb63

Networking & distributed systems

Randomized algorithms

Success stories and systems

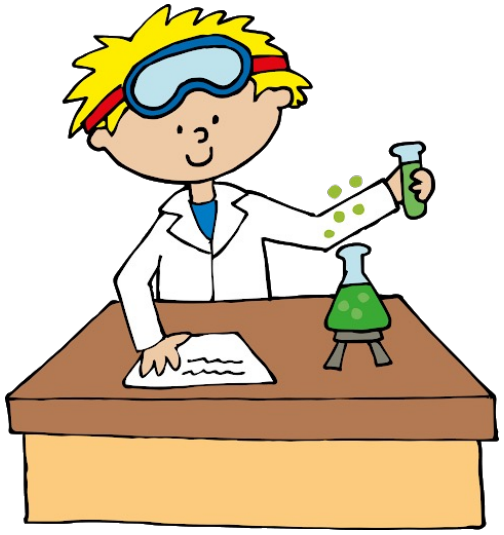
- Verified differential privacy
  - Barthe, Gilles, et al. "Proving differential privacy via probabilistic couplings." *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science*. 2016.
- Verified cryptography
  - Barthe, Gilles, et al. "Probabilistic relational verification for cryptographic implementations." *ACM SIGPLAN Notices* 49.1 (2014): 193-205.
- Verified networking
  - Foster, Nate, et al. "Probabilistic netkat." *Programming Languages and Systems: 25th European Symposium on Programming, ESOP 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2–8, 2016, Proceedings 25*. Springer Berlin Heidelberg, 2016.
- Verified runtime behavior of randomized algorithms
  - Kaminski, Benjamin Lucien, et al. "Weakest precondition reasoning for expected runtimes of randomized algorithms." Journal of the ACM (JACM) 65.5 (2018): 1-68.

1. It is important to verify programs.
2. Many kinds of programs have inherent probabilistic uncertainty.
3. **We need programming languages with probabilistic semantics.**

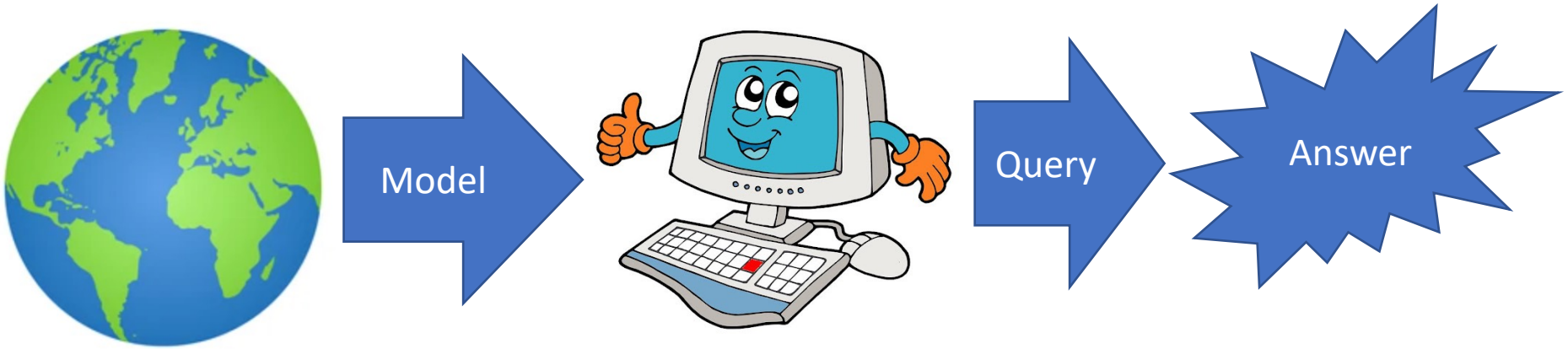
The AI Argument

For probabilistic programming

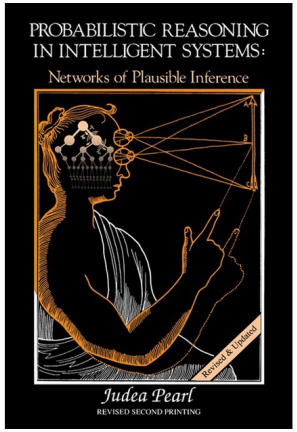
1. We want programs to help us reason rationally about the world.



1. We want programs to help us reason rationally about the world.
2. We need a language for describing the world to a computer.



1. We want programs to help us reason rationally about the world.
2. We need a language for describing the world to a computer.
3. The world is too complicated to describe without probabilities.



J. Pearl
Turing Award Winner (2011)

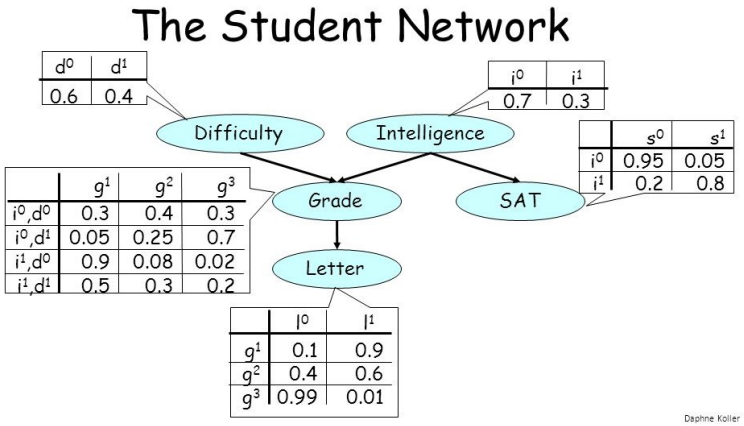
1.1 INTRODUCTION

1.1.1 Why Bother with Uncertainty?

Reasoning about any realistic domain always requires that some simplifications be made. The very act of preparing knowledge to support reasoning requires that we leave many facts unknown, unsaid, or crudely summarized. For example, if we choose to encode knowledge and behavior in rules such as "Birds fly" or "Smoke suggests fire," the rules will have many exceptions which we cannot afford to enumerate, and the conditions under which the rules apply (e.g., seeing a bird or smelling smoke) are usually ambiguously defined or difficult to satisfy precisely in real life. Reasoning with exceptions is like navigating a minefield: Most steps are safe, but some can be devastating. If we know their location, we can avoid or defuse each mine, but suppose we start our journey with a map the size of a postcard, with no room to mark down the exact location of every mine or the way they are wired together. An alternative to the extremes of ignoring or enumerating exceptions is to *summarize* them, i.e., provide some warning signs to indicate which areas of the minefield are more dangerous than others. Summarization is

1. We want programs to help us reason rationally about the world.
2. We need a language for describing the world to a computer.
3. The world is too complicated to describe without probabilities.
4. We need programming languages with probabilistic semantics.

Encode!



Success stories and systems

- There have been over 50 PPLs proposed within the last 15 years
- Some widely-used examples:
  - Stan https://mc-stan.org/
  - Pyro https://pyro.ai/
  - PyMC3 https://www.pymc.io/projects/docs/en/stable/learn.html
- Many applications of PPLs:
  - Stan case studies https://mc-stan.org/users/documentation/case-studies
- Many users today within the scientific community, machine learning community
- Several corporations have developed their own PPLs (Google, Meta, Microsoft)

Syntax and Semantics

of a Tiny Probabilistic Programming Language (TinyPPL)

TinyPPL Syntax

Example programs

```
x <- flip 0.5;
y <- flip 0.5;
return x || y;
```

Flip 2 coins, return disjunction

```
x <- flip 0.5;
y <- return (if x then flip 0.2
             else flip 0.5)
return x || y;
```

If-expressions allow branching

Syntactic features:

- Monadic do-style syntax separates effectful from pure programs
- `flip` for allocating fresh randomness
- Conditionals and basic logic connectives

Notably absent: loops, functions, modules, integers… we will slowly add in a few more features.

TinyPPL Grammar

Pure computations

```
<p> ::=
  | <ident>
  | true
  | false
  | if <p> then <p> else <p>
  | <p> && <p>
  | <p> || <p>
  | !<p>
```

Probabilistic computations

```
<e> ::=
  | flip <float>
  | <id> <- <e>; <e>
  | return <p>
```


"Monadic style"; not too important what that means

TinyPPL Pure Semantics

- Semantics of pure terms are standard

$$[\![\cdot]\!]_p : \texttt{Env} \to \texttt{Bool}$$

- Some examples:

$$[\![\texttt{true}]\!]_p \rho = \top$$

$$[\![x]\!]_p \rho = \rho(x)$$

ρ is an environment, $\top$ is semantic true, $\bot$ is semantic false

TinyPPL Probabilistic Semantics

- Denote probability distributions on values

$$[\![\cdot]\!]_e : \texttt{Env} \to \texttt{Bool} \to [0, 1]$$

- Definition:

$$[\![\texttt{flip}\ \theta]\!]\rho = v \mapsto \begin{cases} \theta & \text{if } v = \top \\ 1 - \theta & \text{otherwise} \end{cases}$$

$$[\![\texttt{return}\ p]\!]\rho = v \mapsto \begin{cases} 1 & \text{if } v = [\![p]\!]_p \rho \\ 0 & \text{otherwise} \end{cases}$$

TinyPPL Probabilistic Semantics

Semantics of bind

$$[\![\begin{array}{l} x \leftarrow \texttt{flip}\ 1/2; \\ y \leftarrow \texttt{flip}\ 1/2; \\ \texttt{return}\ x \vee y \end{array}]\!] \rho(v) \quad = ?$$

TinyPPL Probabilistic Semantics

Semantics of bind

$$[\![x \leftarrow \texttt{flip } 1/2; \; y \leftarrow \texttt{flip } 1/2; \; \texttt{return } x \vee y]\!]\rho(v) = \frac{1}{2}[\![x \leftarrow \texttt{return true}; \; y \leftarrow \texttt{flip } 1/2; \; \texttt{return } x \vee y]\!]\rho(v) + \frac{1}{2}[\![x \leftarrow \texttt{return false}; \; y \leftarrow \texttt{flip } 1/2; \; \texttt{return } x \vee y]\!]\rho(v)$$

TinyPPL Probabilistic Semantics

Semantics of bind

$$[\![x \leftarrow \texttt{flip}\ 1/2; \; y \leftarrow \texttt{flip}\ 1/2; \; \texttt{return}\ x \vee y]\!]\rho(v) =$$

$$\frac{1}{4}[\![x \leftarrow \texttt{return true}; \; y \leftarrow \texttt{return true}; \; \texttt{return}\ x \vee y]\!]\rho(v) +$$

$$\frac{1}{4}[\![x \leftarrow \texttt{return true}; \; y \leftarrow \texttt{return false}; \; \texttt{return}\ x \vee y]\!]\rho(v) +$$

…

TinyPPL Probabilistic Semantics

Semantics of bind

$$[\![x \leftarrow e_1; e_2]\!]\rho(v) = \sum_{v'} [\![e_1]\!](v') \times [\![e_2]\!]\rho[x \mapsto v'](v)$$

Sum over all intermediate states

Run e_2 in new environment with x mapping to v'

Q: How hard do you think it is to evaluate a TinyPPL Program?

Hardness of Evaluating TinyPPL Programs

- **The #SAT problem**: given a Boolean formula φ, count the number of solutions to that formula

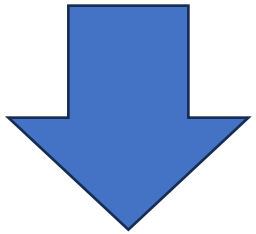
$$\#SAT(a \vee \neg b) = 3$$

- Widely regarded to be computationally intractable (belongs to complexity class #P [Val79], as hard as polytime hierarchy [Toda91])

Hardness of Evaluating TinyPPL Programs

Reducing #SAT to evaluating TinyPPL

$$\#SAT(a \vee b) \wedge (a \vee c)$$



$$2^3 \times [\![$$

```
a <- flip 1/2;
b <- flip 1/2;
c <- flip 1/2;
return (a || b) && (a || c)
```

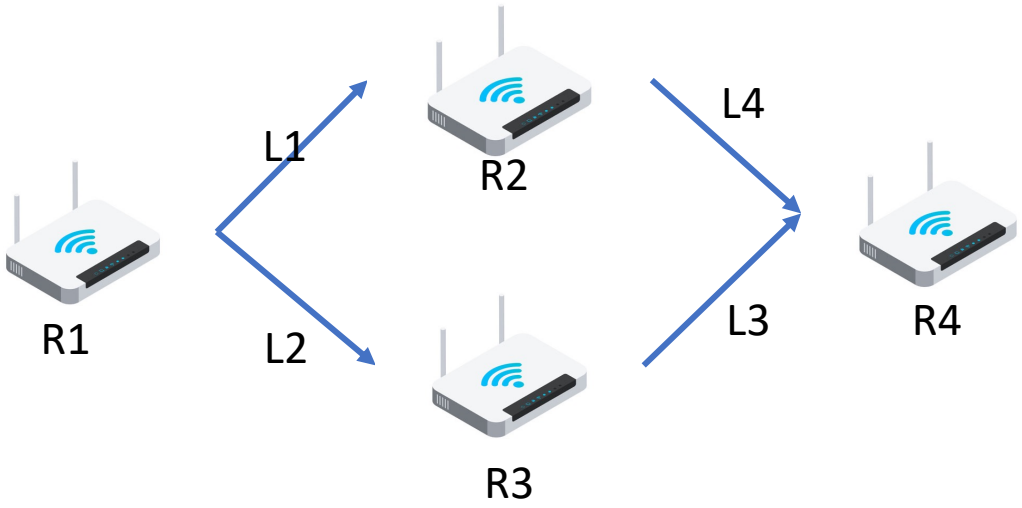
$$]\!] \rho(\mathsf{T})$$

Evaluating TinyPPL programs is *at least* as hard as #SAT!

TinyPPL Demo

```
utop[59]> let p2 = tinyppl_e_of_string "(bind x (flip 0.5)
                                (bind y (flip 0.4)
                                (bind z (flip 0.6)
                                (return (if x y z)))))";;
val p2 : expr =
  Bind ("x", Flip 0.5,
   Bind ("y", Flip 0.4,
    Bind ("z", Flip 0.6, Return (Ite (Ident "x", Ident "y", Ident "z")))))
utop[60]> (prob p1 StringMap.empty true);;
- : float = 0.5
```

TinyPPL Exercise: Network Reliability



Suppose:

- each link fails independently with probably 1/50
- each router chooses which router to forward an incoming packet to with uniform probability.

What is the probability that an incoming packet reaches R4?

Network reliability

```
let network = tinyppl_e_of_string
  "(bind r2forward (flip 0.5)
   (bind l1fail (flip 0.02)
   (bind l2fail (flip 0.02)
   (bind l3fail (flip 0.02)
   (bind l4fail (flip 0.02)
   (return (if r2forward (and (not l1fail) (not l4fail))
                         (and (not l2fail) (not l3fail)))))))))”;;

> prob network StringMap.empty true ;;
- : float = 0.96039999999999809
```

TinyCond

Bayesian conditioning and observation

What is conditioning

- Conditioning *updates your beliefs about the world* given *observations*

- Classic example: medical diagnosis

- Your COVID-19 test comes back positive. Does that mean you have COVID?
  - Not necessarily!
  - The probability that you have COVID should *go up*, but by how much? This depends on the test, prevalence of COVID, etc.

COVID Test Motivating Example

- Query: What is the probability that I have COVID given the test came back positive?
- Required data:
  - **True positive probability**: the probability that the test will be positive if you do have COVID

$$\Pr(\text{Test} = T \mid \text{Covid} = T) = 0.99$$

  - **False positive probability**: the probability that the test will be positive if you do not have COVID

$$\Pr(\text{Test} = T \mid \text{Covid} = F) = 0.05$$

  - **Latent rate**: the probability that an average person has COVID

$$\Pr(\text{Covid} = T) = 0.01$$

Possible worlds

Check that probability of all worlds sums to 1

COVID	Test	Pr(COVID, Test)
T	T	0.01*0.99=0.0099
T	F	0.01*0.01=0.0001
F	T	0.99*0.05=0.0495
F	F	0.99*0.95=0.9405

Possible worlds after conditioning on Test = T

COVID	Test	
T	T	0.0099
T	F	0
F	T	0.0495
F	F	0

Called ***unnormalized probability distribution***, since it does not sum to 1.

Possible worlds after renormalizing (Bayes's Rule)

COVID	Test	Pr(Test | Covid = True)
T	T	0.0099/(0.0099+0.0495)=0.1666666
T	F	0.0001
F	T	0.0495/(0.0099+0.0495)=0.8333333
F	F	0.9405

Note: Even though the test came back positive, it is still more likely that we do not have COVID!

Probability can be surprisingly unintuitive.

Modeling the COVID Diagnosis Scenario in a PPL

```
has_covid <- flip 0.01;
test_pos_with_covid <- flip 0.99;
test_pos_no_covid <- flip 0.05;
test <- return if covid then test_pos_with_covid
               else test_pos_no_covid;
observe test;
return has_covid
```

TinyCond Grammar

Pure computations

```
<p> ::=
  | <ident>
  | true
  | false
  | if <p> then <p> else <p>
  | <p> && <p>
  | <p> || <p>
  | !<p>
```

Probabilistic computations

```
<e> ::=
  | flip <float>
  | <id> <- <e>; <e>
  | observe <e>; <e>
  | return <p>
```

Semantics of TinyCond

- **Unnormalized semantics**: denoted $[\![\mathsf{e}]\!]_U$ essentially the same as TinyPPL, but with added rule for observe:

$$[\![\texttt{observe}\ e_1; e_2]\!]_U(\rho)(v) = \begin{cases} [\![e_2]\!](\rho)(v) & \text{if } [\![e_1]\!](\rho) = \texttt{true} \\ 0 & \text{otherwise.} \end{cases}$$

- Simply assigns probability 0 to all executions that do not satisfy the observation.

Normalized semantics

- Then, we can compute the normalized semantics from the unnormalized semantics:

$$[\![\mathtt{e}]\!](\rho)(v) = \frac{[\![\mathtt{e}]\!](\rho)(v)}{[\![\mathtt{e}]\!](\rho)(\mathtt{tt}) + [\![\mathtt{e}]\!](\rho)(\mathtt{ff})}$$

Non-locality of Conditioning

```
x <- flip 0.5;
y <- flip 0.5;
observe x || y;
return x
```

≡

```
x <- flip 0.6666;
y <- flip 0.6666;
return x
```

Semantically, observation “reaches back in time” to affect previous probabilistic operations in the program!

Some more perspective on semantics

Program Semantics as a Discipline

6. Formal Language Definition

A high level programming language, such as ALGOL, FORTRAN, or COBOL, is usually intended to be implemented on a variety of computers of differing size, configuration, and design. It has been found a serious problem to define these languages with sufficient rigour to ensure compatibility among all implementors. Since the purpose of compatibility is to facilitate interchange of programs expressed in the language, one way to achieve this would be to insist that all implementations of the language shall "satisfy" the axioms and rules of inference which underlie proofs of the properties of programs expressed in the language, so that all predictions based on these proofs will be fulfilled, except in the event of hardware failure. In effect, this is equivalent to accepting the axioms and rules of inference as the ultimately definitive specification of the meaning of the language.

- Hoare, Charles Antony Richard. "An axiomatic basis for computer programming." *Communications of the ACM* 12.10 (1969): 576-580.

- Semantics are a tool to abstract away details of the implementation
- Formally prove programs have certain behaviors
- Given as *logical relations* between input and output states

In many cases, the validity of the results of a program (or part of a program) will depend on the values taken by the variables before that program is initiated. These initial preconditions of successful use can be specified by the same type of general assertion as is used to describe the results obtained on termination. To state the required connection between a precondition (P), a program (Q) and a description of the result of its execution (R), we introduce a new notation:

$$P\,\{Q\}\,R.$$

Tiny Timeline

E. Dijkstra

"An axiomatic basis for computer programming."

1969

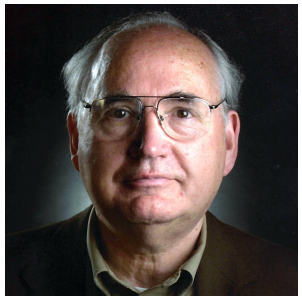
Probabilistic Program Semantics

- Languages are too informal: meaning given by just what the program compiler does!
- We can't implement things consistently or prove things correct
- Need formal system for reasoning
- Introduced logical relations

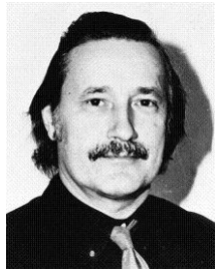
Denotational Semantics

TOWARD A MATHEMATICAL SEMANTICS
FOR
COMPUTER LANGUAGES

0. INTRODUCTION. The idea of a *mathematical* semantics for a language is perfectly well illustrated by the contrast between *numerals* on the one hand and *numbers* on the other. The numerals are *expressions* in a certain familiar language; while the numbers are mathematical *objects* (abstract objects) which provide the intended *interpretations* of the expressions. We need the expressions to be able to communicate the results of our theorizings about the numbers, but the symbols themselves should not be confused with the concepts they denote. For one thing, there are many *different* languages adequate for conveying the *same* concepts (e.g. binary, octal, or decimal numerals). For another, even in the *same* language many different expressions can denote the same concepts (e.g. 2+2, 4, 1+(1+(1+1)), etc.). The problem of explaining these *equivalences* of expressions (whether in the same or different languages) is one of the tasks of semantics and is much too important to be left to syntax alone. Besides, the mathematical concepts are required for the *proof* that the various equivalences have been *correctly* described.



D. Scott



C. Strachey

Scott, Dana S., and Christopher Strachey. *Toward a mathematical semantics for computer languages*. Vol. 1. Oxford: Oxford University Computing Laboratory, Programming Research Group, 1971.

Denotational Semantics

Semantically speaking each of the numerals is meant to denote a unique number. Let N be the set of numbers. (The elements of Nml are expressions; while the elements of N are mathematical objects conceived in abstraction independently of notation.) The obvious principle of interpretation provides a function, the *evaluation mapping*, which we might call $\mathcal{V}$, and which has the functional character:

$$\mathcal{V} : \mathrm{Nml} \to \mathrm{N}.$$

Thus for each $\nu \in \mathrm{Nml}$, the function value

$$\mathcal{V}[\![\nu]\!]$$

is the number denoted by ν.

How is the evaluation function $\mathcal{V}$ determined? Inasmuch as it is to be defined on a recursively defined set Nml, it is reasonable that $\mathcal{V}$ should itself be given a recursive definition. Indeed by following exactly the four clauses of the recursive definition Nml, we are motivated by our understanding of numerals to write:

$$\begin{aligned}
\mathcal{V}[\![0]\!] &= 0 \\
\mathcal{V}[\![1]\!] &= 1 \\
\mathcal{V}[\![\nu 0]\!] &= 2 \cdot \mathcal{V}[\![\nu]\!] \\
\mathcal{V}[\![\nu 1]\!] &= 2 \cdot \mathcal{V}[\![\nu]\!] + 1
\end{aligned}$$

- Introduced semantic bracket: distinction between syntax and semantic domain
- Semantics should be *inductive* on the syntax
- Key problem: notion of *program equivalence*

Denotational Semantics

- Why the fuss with functions, semantic domains, etc.? Dijkstra had a problem: Loops

"**while** *B* **do** *S*"

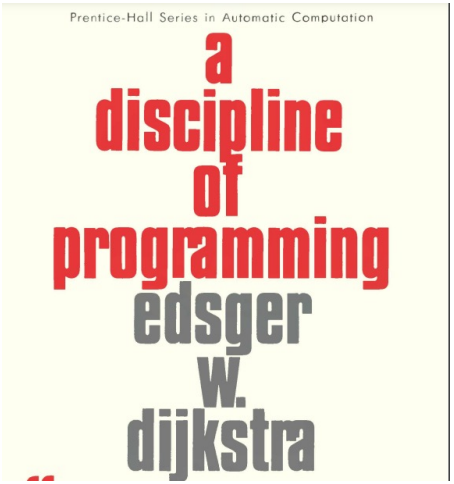
defined as that of the call

"*whiledo*(*B*, *S*)"

of the recursive procedure (described in ALGOL 60 syntax):

procedure *whiledo* (*condition*, *statement*);
begin if *condition* **then begin** *statement*;
whiledo (*condition*, *statement*) **end**
end

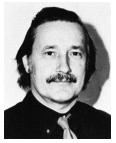
Although correct, it hurts me, for I don't like to crack an egg with a sledgehammer, no matter how effective the sledgehammer is for doing so. For the generation of theoretical computing scientists that became involved in the subject during the sixties, the above recursive definition is often not only "the natural one", but even "the true one". In view of the fact that we cannot even define what a Turing machine is supposed to do without appealing to the notion of repetition, some redressing of the balance seemed indicated.



Tiny Timeline


"An axiomatic basis for computer programming."

1969

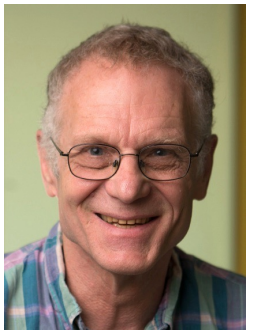
D. Scott C. Strachey

1971

Probabilistic Program Semantics

- Denotational semantics and recursive decompositions of programs
- Syntactic and semantic domains
- Associate each program term with a mathematical function
- Gave a way to formalize loops by working in the semantic domain!
- Entire research program of giving denotational semantics to different kinds of programs…

Kozen 1979



- Kozen, Dexter. "Semantics of probabilistic programs." *20th Annual Symposium on Foundations of Computer Science (sfcs 1979)*. IEEE, 1979.

JOURNAL OF COMPUTER AND SYSTEM SCIENCES 22, 328–350 (1981)

Semantics of Probabilistic Programs

DEXTER KOZEN

IBM Thomas J. Watson Research Center, Yorktown Heights, New York 10598

Revised January 5, 1981

- Gave 4 reasons for studying probabilistic program semantics

Kozen's Motivation

1. Clearing up the difference between endogenous vs. exogenous randomness

2. Match existing program's behavior: languages like ALGOL60 (!) have rand and loops!
   - Formalized an incredibly powerful language (but no conditioning or functions)

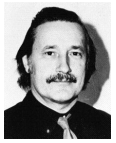
3. A formal system for verifying randomized algorithms

4. Connect existing denotational semantics theory with probability (Loops!)

Tiny Timeline


1969


D. Scott


C. Strachey

1971


D. Kozen

1979

Probabilistic Program Semantics

- Connected randomized algorithms and denotational semantics
- Gave a formal basis for reasoning about both
- Formalized a special language
  - Continuous distributions
  - Loops
  - No observations
  - No functions

Modern Challenges

- Semantics of PPLs is still a vibrant modern research topic

A Domain Theory for Statistical Probabilistic Programming

MATTHIJS VÁKÁR, Columbia University, USA
OHAD KAMMAR, University of Oxford, UK
SAM STATON, University of Oxford, UK

Distinguished Paper POPL'19

Contextual Equivalence for a Probabilistic Language with Continuous Random Variables and Recursion

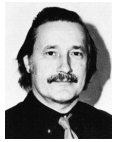
MITCHELL WAND, Northeastern University
RYAN CULPEPPER, Northeastern University
THEOPHILOS GIANNAKOPOULOS, BAE Systems, Burlington MA
ANDREW COBB, Northeastern University

2018, some work done here!

- Peruse POPL and PLDI for many dozens more!
- Questions:
  - How rich can we make the language and still find an interesting/useful semantic domain?
  - What can we prove about probabilistic programs?

Tiny Timeline




D. Scott C. Strachey

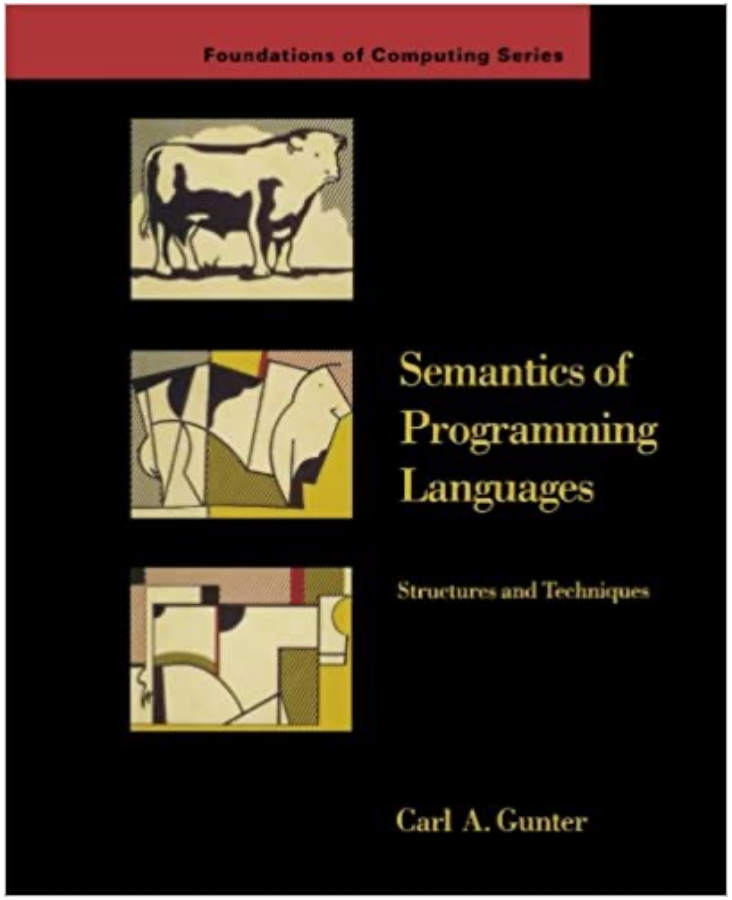
D. Kozen

1969 1971 1979

Probabilistic Program Semantics

- Giry, Michele. "A categorical approach to probability theory." Categorical Aspects of Topology and Analysis: Proceedings of an International Conference Held at Carleton University, Ottawa, August 11–15, 1981. Berlin, Heidelberg: Springer Berlin Heidelberg, 1982.
- Jones, Claire, and Gordon D. Plotkin. "A probabilistic powerdomain of evaluations." Proceedings. Fourth Annual Symposium on Logic in Computer Science. IEEE Computer Society, 1989.
- Ramsey, Norman, and Avi Pfeffer. "Stochastic lambda calculus and monads of probability distributions." Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. 2002.
- Shan, Chung-chieh, and Norman Ramsey. "Exact Bayesian inference by symbolic disintegration." Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages. 2017.
- Heunen, Chris, et al. "A convenient category for higher-order probability theory." *2017 32nd Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. IEEE, 2017.
- Zhang, Yizhou, and Nada Amin. "Reasoning about “reasoning about reasoning”: semantics and contextual equivalence for probabilistic programs with nested queries and recursion." Proceedings of the ACM on Programming Languages 6.POPL (2022): 1-28.

More on Semantics Basics



- Gunter, Carl A. *Semantics of programming languages: structures and techniques*. MIT press, 1992.
- Chapter 1 has a fantastic overview and whirlwind tour of semantics
- Endnotes contain a nice history of the topic
- Unfortunately no book on probabilistic program semantics (yet)

References

- [Val79] Valiant, L.G. (1979). "The complexity of computing the permanent". Theoretical Computer Science. 8 (2): 189–201. doi:10.1016/0304-3975(79)90044-6.
- [Toda91] Toda, Seinosuke (October 1991). "PP is as Hard as the Polynomial-Time Hierarchy". SIAM Journal on Computing. 20 (5): 865–877. CiteSeerX 10.1.1.121.1246. doi:10.1137/0220053. ISSN 0097-5397.