Two ways to construct computer systems

**Programming**
Given a specification over input and output, construct a program that satisfies the specification

Sort a sequence of numbers in ascending order

QuickSort(A, p, q):
if p < q then
  r := Partition(A, p, q)
  QuickSort(A, p, r-1)
  QuickSort(A, r+1, q)

Initial call:
QuickSort(A, 0, n-1)

**(Supervised) Learning**
Given a set of training examples (input-output pairs) learn a model that generalizes and learns the transformation from input to output
Two ways to construct computer systems

**Programming**

Given a specification $\varphi(x, y)$ over input and output, construct a program $P$ such that $\forall x. \varphi(x, P(x))$

- QuickSort(A,p,q):
  - if $p < q$ then
    - $r := \text{Partition}(A, p, q)$
    - QuickSort(A, p, r-1)
    - QuickSort(A, r+1, q)

Initial call:
QuickSort(A, 0, n-1)

**Supervised) Learning**

Given a set of training examples $T = \{(x_i, y_i)|i = 1 \ldots N\}$
learn a model $M$ that minimizes the loss $\sum_{1 \leq i \leq N}(L(M(x_i), y_i))$

Sort a sequence of numbers in ascending order
When programming, and when learning?

Programming makes sense when there exists
• precise requirements
• a provably correct program to satisfy the requirements
Even if we don’t write these down formally!
E.g. Database, operating system, device driver, payroll processing, tax calculations

Learning makes sense when it is hard to write
• precise requirements
• or provably correct implementation
Even if we were to spend time and energy to write these down formally!
E.g. Image classification, NLP, sentiment understanding, language translation, search
Is there value in combining Programs and Models?

Why bother?
Programs and Models: Serving Each Other

• We can instrument the software development process (coding, code reviews, testing, deployment, debugging, etc) collect data, and use ML models to make the process more efficient.

• We can use programming tools to make learning more efficient.
Large scale Programming (Software Engineering) can benefit from using ML to provide recommendations during software life cycle.
• Large scale Programming (Software Engineering) can benefit from using ML to provide recommendations during software life cycle.

[OSDI 18 (best paper), ICSE 19, FSE19, NSDI 20]
Widely deployed and used inside Microsoft
More information:
https://www.microsoft.com/en-us/research/project/sankie/
Large scale Programming (Software Engineering) can benefit from using ML to provide recommendations during software life cycle.
Large scale Programming (Software Engineering) can benefit from using ML to provide recommendations during software life cycle.
Large scale Programming (Software Engineering) can benefit from using ML to provide recommendations during software life cycle.

Programming language and compiler techniques play a key role in making ML systems flexible and efficient.
• Large scale Programming (Software Engineering) can benefit from using ML to provide recommendations during software life cycle

• Programming language and compiler techniques play a key role in making ML systems flexible and efficient
Is there value in combining programs and models more deeply?
When programming, and when learning?

Programming makes sense when there exists
• precise requirements
• a provably correct program to satisfy the requirements
Even if we don’t write these down formally!
E.g. Database, operating system, device driver, payroll processing, tax calculations

Learning makes sense when it is hard to write
• precise requirements
• or provably correct implementation
Even if we were to spend time and energy to write these down formally!
E.g. Image classification, NLP, sentiment understanding, language translation, search
Characteristics of programs and models

Programs are intended to work for all inputs satisfying a precondition.

If the specification or environment changes, programs typically fail!

Programs are succinct ways to specify domain knowledge.

Learning works well “on average” when the test distribution is similar to training distribution.

ML models can generalize and work on unforeseen inputs.

ML models can be opaque sets of floating-point numbers, and hard to interpret.
What if we want programs to be adaptive?

• What if a mathematical specification exists, but it keeps changing and evolving over time? Can we have the system evolve and “adapt” without programmer intervention?

• What if the environment of the program changes, and we want the program to “self-tune” itself in response to the environment changes?
Examples of changing specifications

Changing data formats: shopping web pages, machine generated formats

Customization by each entity in an industry: health data formats, financial data formats, ....
Extract from machine-to-human (M2H) emails

Millions of emails/day
Heterogeneity: 100s of ever evolving formats
Some rare formats with very few emails
10s of data annotators write 100s of hard-crafted templates
Every breakage fixed manually

Goals:
1. Self repair when formats change
2. When new airlines and travel aggregators come online, handle them as automatically as possible
3. Predictability
Two approaches to entity extraction from emails

• Train ML models using labeled data
• Write or automatically synthesize programs from labeled data (using systems such as PROSE)

DNN training

Programs in a Domain Specific Language
What if the input format changes?

• Programs work well when formats are stable, but just fall flat when format’s change

• ML Models generalize somewhat, but don’t get to 100%

• Combining both produces better results than either one in isolation!
Models for Generalization, Programs for Predictability

$\approx 10^3$ each
Models for Generalization, Programs for Predictability

Program Synthesis (PROSE)

New airline or changed format
~70% precision

Programs are regular
(usually, all right or all wrong)
Heterogeneous Data Extraction Framework

Different strategies for feedback
High ranked program outputs can be directly fed in

LSTM-CRF
Noisy labels
NDSYn Program Synthesizer

Labelled Inputs
Mails + Rule-based Extractions
Unlabelled Inputs
Additional annotations
Semi-automated annotator

ML Model
Email HTML + Noisy label output pairs

PROSE + Web DSL + Field constraints
Disjunctive program (Covering Sequence of programs)

Runtime
• Design of “Domain Specific Language” (DSL) is key for useful functioning of the combined system
• With a well-designed DSL, program synthesis can act as a “regularizer” and make the system predictable, whereas ML models enable the system to be generalizable and robust to format changes
Deployment results:
"saves us nearly 100-120 Hrs of flight model maintenance time from data annotation per week ... 50% of data annotator bandwidth”

Programs are cheaper to execute, so they are used at runtime. ML models are used offline for self-healing and robustness when formats change.

Synthesis and Machine Learning for Heterogeneous Extraction, Arun Iyer, Manohar Jonnalagedda, Suresh Parthasarathy, Arjun Radhakrishna, Sriram Rajamani, PLDI 2019
Related work

“Programmatically Interpretable Reinforcement Learning”, Abhinav Verma, Vijayaraghavan Murali, Rishabh Singh, Pushmeet Kohli, and Swarat Chaudhuri, In ICML 2018

“Verifiable reinforcement learning via policy extraction”, Osbert Bastani, Yewen Pu, Armando Solar-Lezama, NIPS 2019

What if we want programs to be adaptive?

- What if a mathematical specification exists, but it keeps changing and evolving over time? Can we have the system evolve and “adapt” without programmer intervention?

- What if the environment of the program changes, and we want the program to “self-tune” itself in response to the environment changes?
Configuration settings are ubiquitous in software!

```
[Resource1]
RefreshInterval=00:01:00
PriorityHighUnderloaded=97
PriorityHighOverloaded=98
PriorityLowUnderloaded=88
PriorityLowOverloaded=90
...
[Resource2]
RefreshInterval=00:01:00
PriorityHighUnderloaded=80
PriorityHighOverloaded=90
...
```

double ScoreLinesMap(double sel, double lines) {
    double minScore = 100.0;
    double alpha = 1.0; double beta = 1.0;
    return alpha * sel + beta * lines + minScore * 20;
}

bool IsLikelyDataRatio(int dataCount, int totalcount) {
    if (totalCount < 10) return dataCount >= 6;
    if (totalCount < 20) return dataCount >= 15;
    if (totalCount < 50) return dataCount >= 30;
    return dataCount / (double) totalCount >= 0.6;
}

Can we learn such “structured” decision functions automatically?
Programming By Rewards (PBR)

- Decision function structure
- Rewards from executions

PBR engine

Learned Decision Function
Programming By Rewards (PBR)

```csharp
int[2] counts = getCounts(contents);
...
if (PBR.DecisionFunction(PBRID_IsLikelyDataRatio, count[0], count[1]))
    preProcess(fileName);
...

double reward =
    (success/Config.Benchmarks.Length) −
    (sw.ElapsedMilliseconds/1000);
PBR.AssignReward(reward);
```
```csharp
int[2] counts = getCounts(contents);
...
if (PBR.DecisionFunction(PBRID_IsLikelyDataRatio, count[0], count[1]))
    preProcess(fileName);
...

bool IsLikelyDataRatio(int dataCount, int totalCount) {
    if (totalCount < 10) return dataCount >= 6;
    if (totalCount < 20) return dataCount >= 15;
    if (totalCount < 50) return dataCount >= 30;
    return (dataCount / totalCount >= 0.6);
}

double reward =
    (success/Config.Benchmarks.Length) –
    (sw.ElapsedMilliseconds/1000);
PBR.AssignReward(reward);
```
For a given **unknown (black-box)** reward function $r$, and a **known** code template for the decision (e.g., linear), the goal is to solve:

$$\max_{w \in \mathbb{R}^d} \sum_i r_i(w^T x_i)$$

RL, online learning, black-box optimizers are expensive in terms of reward calls needed (prop. to $d$)
Learning with black-box rewards

For a given **unknown** (black-box) reward function \( r \), and a **known** code template for the decision (e.g., linear), the goal is to solve:

\[
\max_{w \in \mathbb{R}^d} \sum_i r_i(w^T x_i)
\]

\[
\begin{align*}
& w_1 \cdot \text{latency} + \\
& w_2 \cdot \text{load} + \\
& w_3 \cdot \text{min} + w_4
\end{align*}
\]

Gradient-descent style algorithms, with #rewards needed proportional to #decisions \( m \) (typically, \( m = 1 \))
Case study: PROSE codebase

- We applied Self-Tune to simultaneously learn~70 ranking heuristics in PROSE
- Reward: # tasks where PROSE synthesizes a correct program
- Each reward query is expensive (~20 minutes)
- In ~100 hours of training, PBR improves over state-of-the-art ML-ranker by ~8% in terms of accuracy
- Competitive with the manually-tuned heuristics that took 2+ years of effort

<table>
<thead>
<tr>
<th>Ranker</th>
<th>Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>ML-PROSE [2019**]</td>
<td>606/740</td>
</tr>
<tr>
<td>PROSE + SelfTune [2020*]</td>
<td>668/740</td>
</tr>
</tbody>
</table>


** Learning natural programs from a few examples in real-time. N., Dany Simmons, Naren Datha, Prateek Jain, Sumit Gulwani. AISTATS, 2019.
Learning algorithms can exploit the structure of the decision function to get better sample complexity.
Story so far: Programs + Models

Using ML to provide recommendations during the software life cycle

Using compilers and runtimes to make ML systems flexible and efficient

Having programs adapt when specifications (e.g., formats) change

Having programs “self-tune” when environments change

Unified framework to express logic + probability, domain knowledge + examples, rules + statistics

Probabilistic Programming
Human-level concept learning through probabilistic program induction

Brenden M. Lake, Ruslan Salakhutdinov, Joshua B. Tenenbaum

[Science 2015]

Bayesian programming language framework (BPL)

- Capable of learning visual concepts from a single example
- Programmer specifies primitives, parts and subparts as domain knowledge
- System infers knowledge representation as probabilistic programs using Bayesian inference
bool c1, c2;
c1 = Bernoulli(0.5);
c2 = Bernoulli(0.5);
return(c1,c2);

<table>
<thead>
<tr>
<th>c1</th>
<th>c2</th>
<th>(P(c_1, c_2))</th>
</tr>
</thead>
<tbody>
<tr>
<td>false</td>
<td>false</td>
<td>1/4</td>
</tr>
<tr>
<td>false</td>
<td>true</td>
<td>1/4</td>
</tr>
<tr>
<td>true</td>
<td>false</td>
<td>1/4</td>
</tr>
<tr>
<td>true</td>
<td>true</td>
<td>1/4</td>
</tr>
</tbody>
</table>
bool c1, c2;
c1 = Bernoulli(0.5);
c2 = Bernoulli(0.5);
observe(c1 || c2);
return(c1,c2);

<table>
<thead>
<tr>
<th>c1</th>
<th>c2</th>
<th>P(c1, c2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>false</td>
<td>false</td>
<td>0</td>
</tr>
<tr>
<td>false</td>
<td>true</td>
<td>1/3</td>
</tr>
<tr>
<td>true</td>
<td>false</td>
<td>1/3</td>
</tr>
<tr>
<td>true</td>
<td>true</td>
<td>1/3</td>
</tr>
</tbody>
</table>
float skillA, skillB, skillC;
float perfA1, perfB1, perfB2,
    perfC2, perfA3, perfC3;
skillA = Gaussian(100, 10);
skillB = Gaussian(100, 10);
skillC = Gaussian(100, 10);

// first game: A vs B, A won
perfA1 = Gaussian(skillA, 15);
perfB1 = Gaussian(skillB, 15);
observe(perfA1 > perfB1);

// second game: B vs C, B won
perfB2 = Gaussian(skillA, 15);
perfC2 = Gaussian(skillB, 15);
observe(perfB2 > perfC2);

// third game: A vs C, A won
perfA3 = Gaussian(skillA, 15);
perfC3 = Gaussian(skillB, 15);
observe(perfA3 > perfC3);
return(skillA, skillB, skillC);

Sample perfA from a noisy skillA distribution
Sample perfB from a noisy skillB distribution
if perfA > perfB then A wins else B wins

“TrueSkill” from Infer.Net @MSR Cambridge
• Player A beats Player B, if A performs better than B during the game
• Performance is a stochastic function of skill

skillA = Gaussian(102.1,7.8)
skillB = Gaussian(100.0,7.6)
skillC = Gaussian(97.9,7.8)
double logScr, age;
bool isFemale, isAA;

double f1 =
    estimateLogEGFR(logScr, age,
        isFemale, isAA);
double nLogScr, nAge;
bool nIsFemale, nisAA;

logScr = logScr +
    Uniform(-0.1, 0.1);

logScr = logScr +
    Uniform(-1, 1);

nIsFemale = isFemale;
if(Bernoulli(0.01))
    nIsFemale = !isFemale;

nIsAA = isAA;
if(Bernoulli(0.01))
    nIsAA = !isAA;

c = estimateLogEGFR
    (logScr, age,
        isFemale, isAA)
{
    double k, alpha;
    double f = 4.94;
    if(isFemale){
        k = -0.357;
        alpha = -0.328;
    }else{
        k = -0.105;
        alpha = -0.411;
    }

    if(logScr < k)
    f = alpha * (logscr-k);
else
    f = -1.209 * (logscr-k);

f = f - 0.007 * age;

if(isFemale) f = f + 0.017;
if(isAA) f = f + 0.148;

return f;
}

return(bigChange);
Lotka-Volterra Population Model

```c
int goats, tigers;
double c1, c2, c3, curTime;
// initialize populations
goats = 100; tigers = 4;
// initialize reaction rates
c1 = 1; c2 = 5; c3 = 1;
//initialize time
curTime = 0;

while (curTime < TIMELIMIT)
{
    if (goats > 0 && tigers > 0)
    {
        double rate1, rate2, rate3, rate;
        rate1 = c1 * goats;
        rate2 = c2 * goats * tigers;
        rate3 = c3 * tigers;
        rate = rate1 + rate2 + rate3;
        double dwellTime = Exponential(rate);
        int discrete = Disc5(rate1/rate, rate2/rate);
        curTime += dwellTime;
        switch (discrete)
        {
            case 0: goats++; break;
            case 1: goats--; tigers++; break;
            case 2: tigers--; break;
        }
    }
    else if (goats > 0)
    {
        double rate;
        rate = c1 * goats;
        double dwellTime = Exponential(rate);
        curTime += dwellTime;
        goats++;
    }
    else if (tigers > 0)
    {
        double rate;
        rate = c3 * tigers;
        double dwellTime = Exponential(rate);
        curTime += dwellTime;
        tigers--;
    }
}
//end while loop
return(goats,tigers);
}
```

Lotka, Elements of physical biology. Williams & Wilkins company, Baltimore, 1925.

Several more applications that can be modeled as probabilistic programs

- Hidden Markov Models (eg. for speech recognition)
- Kalman Filters (eg. In computer vision)
- Markov Random Fields (eg. In image processing)
- Markov Chains
- Bayesian Networks
- And more applications:
  - Ecology & Biology (Carbon modeling, Evolutionary Genetics,...)
  - Security (quantitative information flow, inference attacks)
Probabilistic Inference

• Infer the distribution specified by a probabilistic program.
  • Generate samples to test a machine learning algorithm
  • Calculate the expected value of a function wrt the distribution specified by the program
  • Calculate the mode of the distribution specified by the program

• Punchline:
  • Inference is program analysis of probabilistic programs
Pearl’s Burglar alarm example

```c
int alarm() {
    char earthquake = Bernoulli(0.001);
    char burglary = Bernoulli(0.01);
    char alarm = earthquake || burglary;
    char phoneWorking =
        (earthquake)? Bernoulli(0.6) : Bernoulli(0.99);
    char maryWakes;
    if (alarm && earthquake)
        maryWakes = Bernoulli(0.8);
    else if (alarm)
        maryWakes = Bernoulli(0.6);
    else maryWakes = Bernoulli(0.2);
    char called = maryWakes && phoneWorking;
    observe(called);
    return burglary;
}
```

“called” is a low probability event, and causes large number of rejections during sampling
Pre transformation

- Let $P$ be any program
- Let $\text{Pre}(P)$ denote the program obtained by propagating observe statements immediately after sample statements

Theorem: $P = \text{Pre}(P)$

```cpp
int alarm() {
  bool earthquake = Bernoulli(0.001);
  bool burglary = Bernoulli(0.01);
  bool alarm = earthquake || burglary;
  bool phoneWorking = earthquake ? Bernoulli(0.6) : Bernoulli(0.99);
  observe(phoneWorking);
  if (alarm && earthquake) {
    bool maryWakes = Bernoulli(0.8);
    observe(maryWakes && phoneWorking);
  } else if (alarm) {
    bool maryWakes = Bernoulli(0.6);
    observe(maryWakes && phoneWorking);
  } else {
    bool maryWakes = Bernoulli(0.2);
    observe(maryWakes && phoneWorking);
  }
  bool called = maryWakes && phoneWorking;
  observe(called);
  return burglary;
}
```
Problem. Estimate expectation of $\phi(x)$ wrt to the distribution $P^*(x)$: $\int x \cdot P^*(x) \times \phi(x) \, dx$

If we can sample from $P^*(x)$ we can estimate the expectation as:

$$\frac{1}{N} \times (\phi(x_1) + \phi(x_2) \ldots + \phi(x_N))$$

Figure from D J Mackay, Introduction to Monte Carlo Methods
1. Draw samples for $x'$ from a proposal $Q(x; x')$
2. Compute $a = \frac{P^*(x') \times Q(x; x')}{P^*(x) \times Q(x'; x)}$
3. If $a \geq 1$, accept $x'$ else accept with probability $a$

Figure from DJ Mackay, Introduction to Monte Carlo Methods
MH without rejections

For each statement of the form:
\[ x_i = \text{Dist}(E); \text{observe}(\phi) \]

Calculate
\[ \beta_i = \frac{\text{Density(Dist}(E))(x') \times Q|\phi(x^{(t)}; x')}{\text{Density(Dist}(E))(x^{(t)}) \times Q|\phi(x'; x^{(t)})} \]

During each run of \( \pi_i \), for each sample statement:
- Sample from proposal sub-distribution \( Q \) conditioned by \( \phi \)
- \( \beta = \beta_1 \times \beta_2 \times \cdots \times \beta_n \)

If \( \beta \geq 1 \), accept \( x' \) else accept with probability \( \beta \)

R2: An Efficient MCMC Sampler for Probabilistic Programs, 
In AAAI '14: AAAI Conference on Artificial Intelligence, July 2014
Program Slicing
[Mark Wiser, 1981]

Reduce a program to a smaller program “slice” when interested in only some values of interest at a program point

Many applications:
• Debugging
• Optimization
• Maintenance

Abstract
Program slicing is a method used by experienced computer programmers for abstracting from programs. Starting from a subset of a program’s behavior, slicing reduces that program to a minimal form which still produces that behavior. The reduced program, called a “slice”, is an independent program guaranteed to faithfully represent the original program within the domain of the specified subset of behavior.

Finding a slice is in general unsolvable. A dataflow algorithm is presented for approximating slices when the behavior subset is specified as the values of a set of variables at a statement. Experimental evidence is presented that these slices are used by programmers during debugging. Experience with two automatic slicing tools is summarized. New measures of program complexity are suggested based on the organization of a program’s slices.

KEYWORDS: debugging, program maintenance, software tools, program metrics, human factors, dataflow analysis

Introduction
A large computer program is more easily constructed, understood, and maintained when broken into smaller pieces. Several different methods decompose programs during program design, such as information hiding (Coxman 1972), data abstraction, behavior is of interest. For instance, during debugging a subset of behavior is being corrected, and in program modification or maintenance a subset of behavior is being improved or replaced. In these cases, a programmer starts from the program behavior and proceeds to find and modify the corresponding portions of program code. Code not having to do with behavior of interest is ignored. Gould and Drnkowski (1974) report programmers behaving this way during debugging, and a further confirming experiment is presented below.

A programmer maintaining a large, unfamiliar program would almost have to use this behavior-first approach to the code. Understanding an entire system to change only a small piece would take too much time. Since most program maintenance is done by persons other than the program designers, and since 67 percent of programming effort goes into maintenance (Zelkowitz, Shaw, and Bannon 1979), decomposing programs by behavior must be a common occurrence.

Automatic slicing requires that behavior be specified in a certain form. If the behavior of interest can be expressed as the values of some sets of variables at some set of statements, then this specification is said to be a slicing criterion. Dataflow analysis (Hecht 1977) can find all the program code which might have influenced the specified behavior, and this code is called a slice of the program. A slice is itself an executable program, whose behavior must be identi-
Dependences used by Slicing

S1: \[ A := B \times C \]
S2: \[ C := A \times E + 1 \]

S2 is “Data Dependent” on S

S1: \[ \text{if} (A) \text{ then} \]
S2: \[ B = C + D \]

S2 is “Control Dependent” on S1
Probabilistic Programs have new dependences

• Figure represents

\[ p(x, y, z) = p(z| x, y) \cdot p(x) \cdot p(y) \]

• There is no dependence between \( x \) and \( y \)

• On the other hand, if \( z \) (or some descendant of \( z \)) is observed, then \( x \) depends on \( y \) and vice versa

• This is called “observe dependence”

Slicing Probabilistic Programs

Slicing Probabilistic Programs, In PLDI ’14: Programming Language Design and Implementation, Jue 2014

Abstract
Probabilistic programs use familiar notation of programming languages to specify probabilistic models. Suppose we are interested in estimating the distribution of the return expression r of a probabilistic program P. We are interested in slicing the probabilistic program P and obtaining a simpler program S1(P) which returns only those parts of P that are relevant to estimating r, and elides those parts of P that are not relevant to estimating r. We desire that the S1 transformation be both correct and efficient. By correct, we mean that S1(P) and S11(P) have identical semantics on r. By efficient, we mean that estimation over S1(P) be as fast as possible.

1. Introduction
Probabilistic programs are “usual” programs (written in languages like C or Java or LISP or ML) with two added constructs: (1) the ability to draw values at random from distributions, and (2) the ability to condition values of variables in a program through observable statements (which allow data from real world observations to be incorporated into a probabilistic program). A variety of probabilistic programming languages and systems have been proposed [2, 10-12, 18, 20, 23, 26]. However, unlike “usual” programs which are written for the purpose of being executed, the purpose of a probabilistic program is to implicitly specify a probability distribution. Probabilistic programs can be used to represent probabilistic graphical models [19], which use graphs to denote conditional dependencies between random variables. Probabilistic graphical models are widely used in statistics and machine learning, with diverse application areas including information extraction, speech recognition, computer vision, coding theory, biology and reliability analysis.

A Theory of Slicing for Imperative Probabilistic Programs

TORBEN AMTOFT, Kansas State University, USA
ANINDYA BANERJEE, IMDEA Software Institute, Spain

Dedicated to the memory of Sebastian Daničić.

We present a theory for slicing imperative probabilistic programs containing random assignments and “observe” statements for conditioning. We represent such programs as probabilistic control-flow graphs (pCFGs) whose nodes modify probability distributions. This allows direct adaptation of standard machinery such as data dependence, postdominators, relevant variables, and so on, to the probabilistic setting. We separate the specification of slicing from its implementation:

(1) first, we develop syntactic conditions that a slice must satisfy (they involve the existence of another disjoint slice such that the variables of the two slices are probabilistically independent of each other);
(2) next, we prove that any such slice is semantically correct;
(3) finally, we give an algorithm to compute the least slice.

To generate smaller slices, we may in addition take advantage of knowledge that certain loops will terminate (almost) always.

Our results carry over to the slicing of structured imperative probabilistic programs, as handled in recent work by Hur et al. For such a program, we can define its slice, which has the same “normalized” semantics as the original program; the proof of this property is based on a result proving the adequacy of the semantics of pCFGs w.r.t. the standard semantics of structured imperative probabilistic programs.

CCS Concepts: • Theory of computation → Probabilistic computation; Program semantics; Software and its engineering → Correctness; Automated static analysis;

Additional Key Words and Phrases: Probabilistic programming, program slicing, probabilistic control-flow graphs

ACM Reference format:
https://doi.org/10.1145/3372895
Probabilistic Programming

Andrew D. Gordon  
Microsoft Research  
adg@microsoft.com

Thomas A. Henzinger  
IST Austria  
tah@ist.ac.at

Aditya V. Nori  
Microsoft Research  
adityan@microsoft.com

Sriram K. Rajamani  
Microsoft Research  
sriram@microsoft.com

Abstract

Probabilistic programs are usual functional or imperative programs with two added constructs: (1) the ability to draw values at random from distributions, and (2) the ability to condition values of variables in a program via observations. Models from diverse application areas such as computer vision, coding theory, cryptographic protocols, biology and reliability analysis can be written as probabilistic programs.

Probabilistic inference is the problem of computing an explicit representation of the probability distribution implicitly specified by a probabilistic program. Depending on the application, the desired output from inference may vary—we may want to estimate the expected value of some function $f$ with respect to the distribution, or the mode of the distribution, or simply a set of samples drawn from the distribution.

In this paper, we describe connections this research area called “Probabilistic Programming” has with programming languages and software engineering, and this includes language design, and the static and dynamic analysis of programs. We survey current state of the art and speculate on promising directions for future research.
Using ML to provide recommendations during the software life cycle

Using compilers and runtimes to make ML systems flexible and efficient

Having programs adapt when specifications (e.g., formats) change

Having programs “self-tune” when environments change

Probabilistic programs: General framework to express rules and examples

Deep combinations of program analysis ideas with ML ideas have the potential to scale probabilistic inference
Challenges and Opportunities (1)

Adaptive specifications and environments

• Need baselines and benchmarks, that capture evolution over time
• Need metrics to measure manual effort in updating annotations as well as precision and recall over time
• Need new user interaction models for involving annotators and users when the system needs human help

Possibility to develop a new field: “model and program engineering”, on how models and programs evolve over time.
Challenges and Opportunities (2)

Assurance of ML/AI systems using verified monitoring

• Can we write partial specifications for safety critical ML/AI systems?
• Can we synthesize monitors to “safeguard” such systems even if the ML/AI algorithms are hard to verify?
• How do we evolve the specifications over time?

Challenges and Opportunities (3)

Practical and usable frameworks to combine domain knowledge (rules) with empirical knowledge (examples and data)

- Technical challenge:
  - Scaling probabilistic inference
  - Usable programming languages and notation

- Industrial challenge: Development life cycle and tools for probabilistic programs

- Educational challenge:
  - Developing educational material and pedagogy for modeling rules and empirical knowledge together