CS7480: Topics in Programming Languages: Probabilistic Programming

Lecture 5: `SimPPL` Inference

Instructor: Steven Holtzen

Place: Northeastern University

Term: Fall 2021

Course webpage:
https://www.khoury.northeastern.edu/home/sholtzen/CS7480Fall21/

Semantics

- You've now seen the semantics for a simple language… you will see these in some papers!

- Let's take a historical tour through semantics as a discipline now that we have an example of our own semantics
  - Place yourself in the broader literature (one of the course goals)
  - See some hard open problems

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

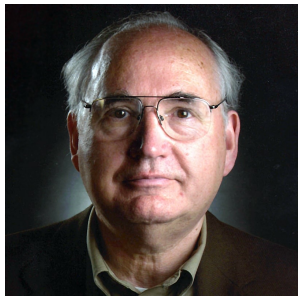
Probabilistic Program Semantics

- 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

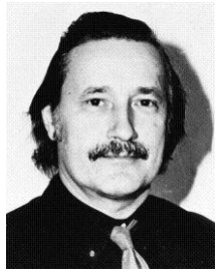
Denotational Semantics

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*"

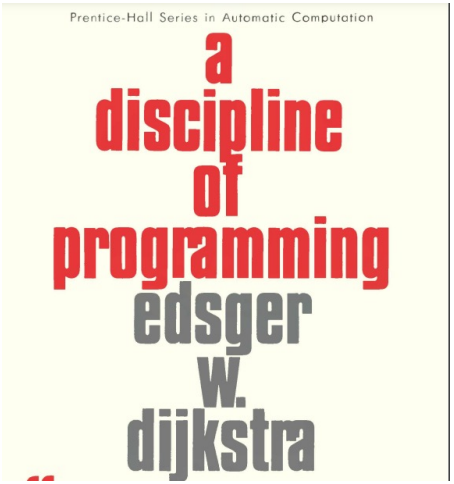
defined as that of the call

"*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

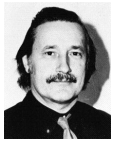
Although correct, it hurts me, for I don't like to crack an egg with a sledgehammer, no matter how effective the sledgehammer is for doing so. For the generation of theoretical computing scientists that became involved in the subject during the sixties, the above recursive definition is often not only "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

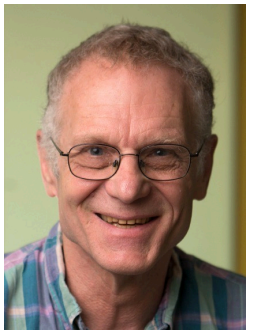
D. Scott C. Strachey

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)

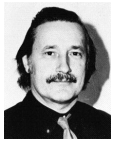
3. A formal system for verifying randomized algorithms

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

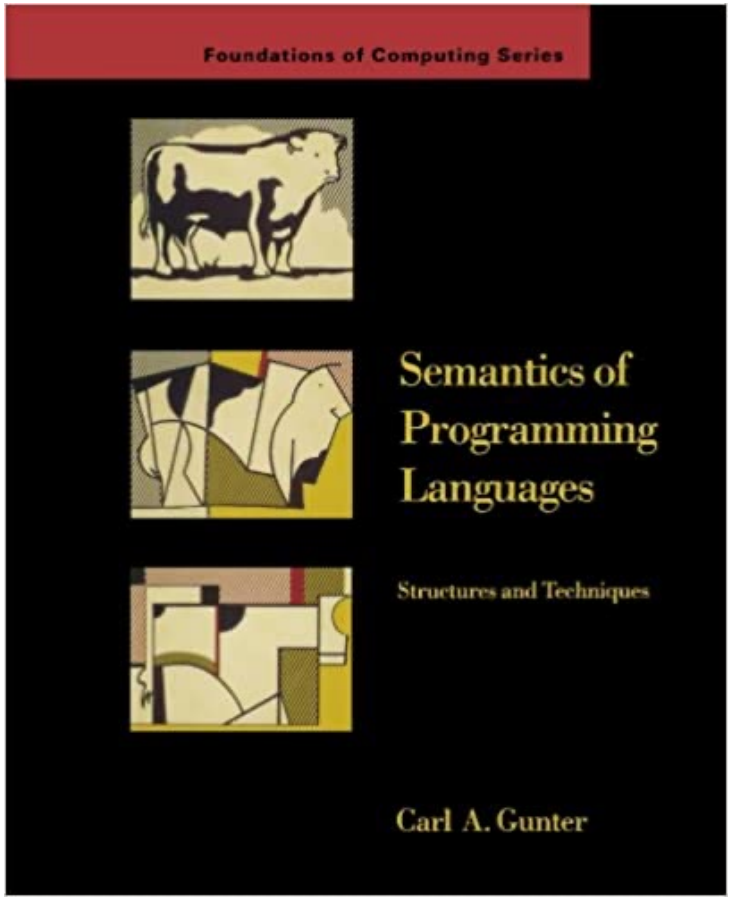
Contextual Equivalence for a Probabilistic Language with Continuous Random Variables and Recursion

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?

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)

Semantics vs. Implementation

- Semantics tell you *what a program does…*

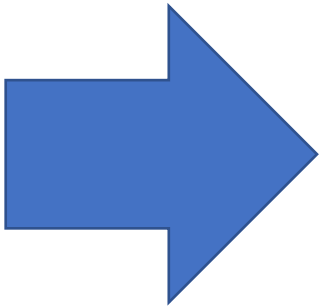
- They don't tell you how to *run the program efficiently*

`SimPPL` Inference

How Hard Is `SimPPL` Inference?

- NP-Hard in the size of the program
- To show this, *give a reduction:* show that we can use `SimPPL` inference to solve a 3SAT problem

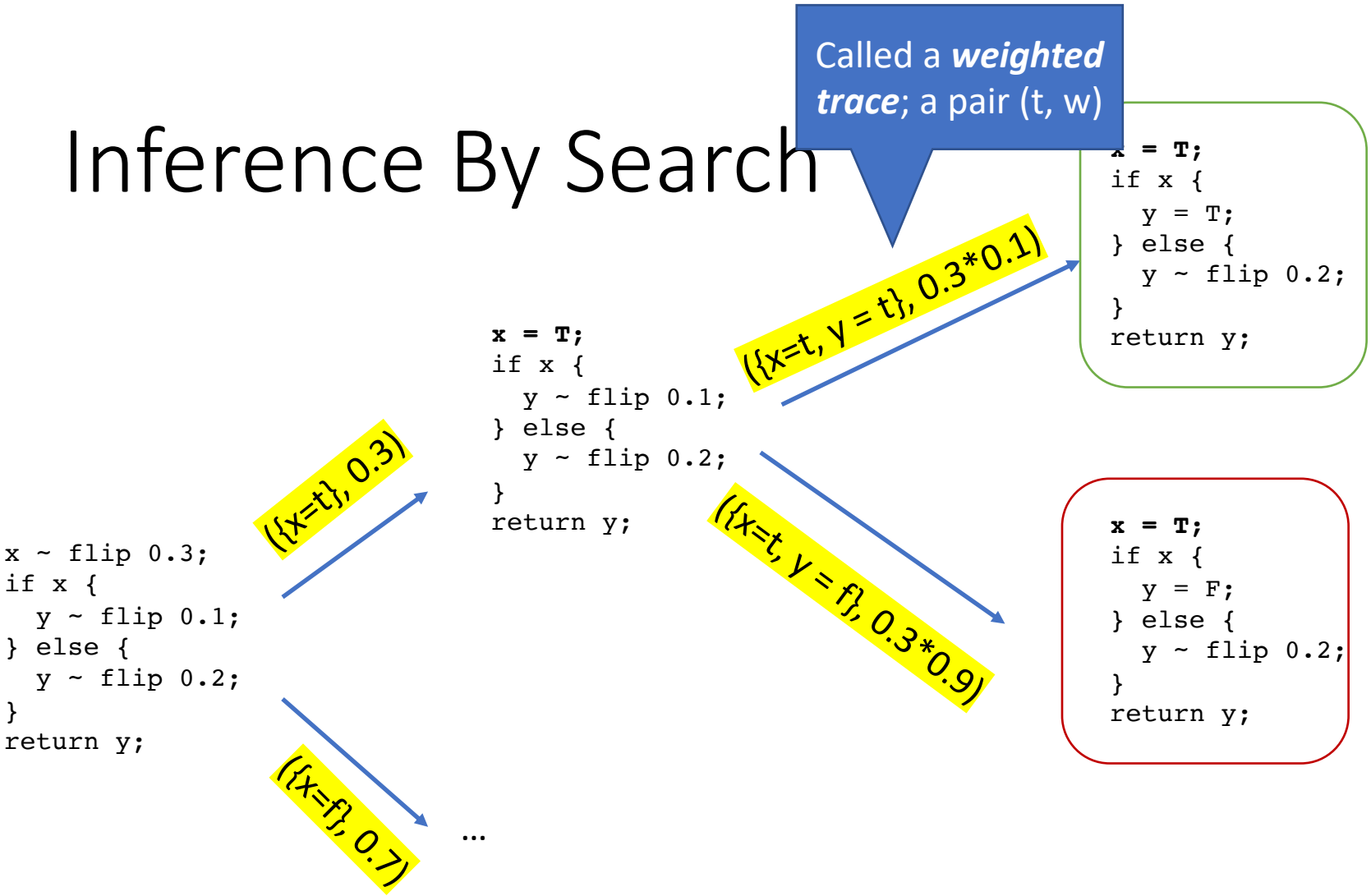
$(A \vee B \vee C) \wedge (A \vee \neg B \vee D)$



```
A ~ flip 0.5;
B ~ flip 0.5;
C ~ flip 0.5;
D ~ flip 0.5;
return (&& (|| (|| A B) C)
           (|| (|| A (! B)) D)
```

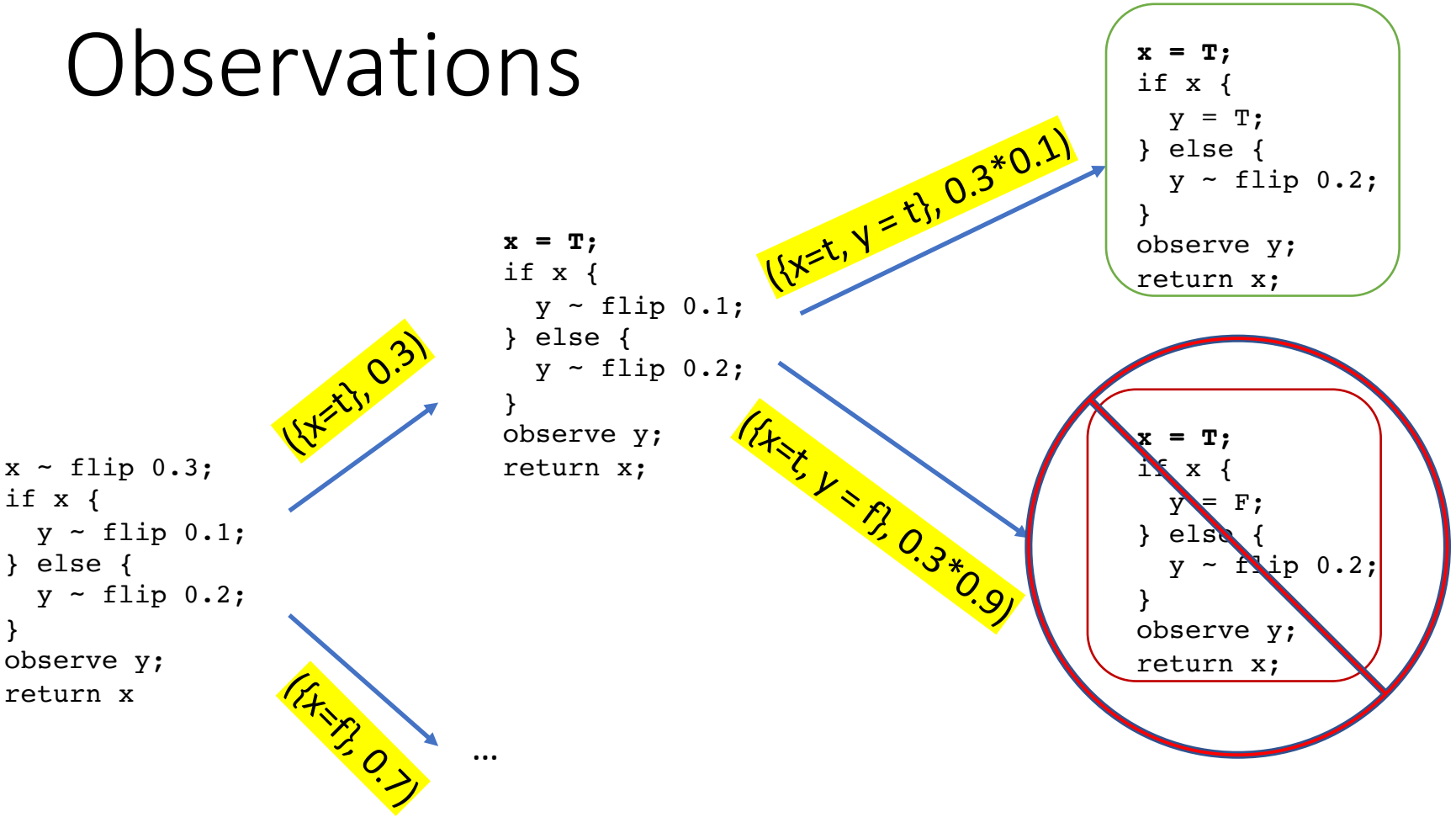
Size does not blow up

Inference By Search



- Recursively branch on all possible values to random variables
- Sum up total probability of outputting true

Observations



- Discard all paths that violate observation
- Compute normalizing constant by summing probability of all non-discarded paths

Formalization of `Search`

- We can give a precise inductive description of how this search procedure computes probabilities
- Define a map `Search: s` $\rightarrow$ `[Trace]` $\rightarrow$ `[Trace]`
  - Given a list of input weighted traces, yields a list of output weighted traces

```
Search([({y = T}, 0.4), ({y = F}, 0.6)])(z~flip θ) =
    [({y=T,z=T}, 0.4*θ),
     ({y=T,z=F}, 0.4*(1-θ)),
     ({y=F,z=T}, 0.6*θ),
     ({y=F,z=F}, 0.6*(1 − θ))]
```

Be careful about overlapping names!


Inference Correctness

- Formally, an inference algorithm is *correct* if it agrees with the semantics, i.e. it satisfies that for any program `s; return e`:

$$\frac{1}{Z} \times \sum_{\{(t,w)|(t,w)\in \texttt{Search}(\emptyset)(s),t\models[\![e]\!]\}} w = [\![\texttt{s ; return e}]\!]$$

Slightly informal

- Z is the normalizing constant (sum over all traces)
- Read: the sum of the weights of all traces that satisfy the return expression is equal to the semantics of the program

Inference Correctness Tiny Example

- First build the set of traces:

$$\texttt{Search}(\emptyset)(\texttt{x} \sim \texttt{flip 0.1; return x}) = [(\{x = T\}, 0.1), (\{x = F\}, 0.9)]$$

- Now we can compare the semantics and search-based inference:

$$[\![\texttt{x} \sim \texttt{flip 0.1; return x}]\!] = 0.1 = \sum_{\{(t,w) \in [(\{x=T\},0.1),(\{x=F\},0.9)]\} \wedge t \models [\![x]\!])} w$$

When Inference by Search Fails

- Size of search tree is *exponential* in number of random variables

```
a ~ flip 0.1;
if a {
   b ~ flip 0.3
} else {
   b ~ flip 0.4
}
if b {
   c ~ flip 0.3
} else {
   c ~ flip 0.4
}
...
```

Approximate Inference

- What if we are willing to relax our notion of inference correctness to *approximate correctness*

$$[\![\texttt{s; return e}]\!] \approx \texttt{Inference}(\texttt{s; return e})$$

Direct Sampling

- Run the program many times and build a set of (unweighted!) traces

```
x ~ flip 0.3;
if x {
  y ~ flip 0.1;
} else {
  y ~ flip 0.2;
}
return y;
```

x	y	Query?
T	T	T
T	F	F
F	F	F
T	T	T

- Then, the probability of the query is given by the fraction of traces that satisfies the query (here, ½)

Direct Sampling with Observations

- Run the program many times and build a set of (unweighted!) traces

```
x ~ flip 0.3;
if x {
  y ~ flip 0.1;
} else {
  y ~ flip 0.2;
}
observe y;
return y;
```

x	y	Query?
T	T	T
T	F	F
F	F	F
T	T	T

- Then, the probability of the query is given by the fraction of *accepted traces* that satisfies the query (here, 2/2)

Direct Sampling Correctness

- Denote by $\texttt{Sample}_n(\texttt{s; return e})$ the fraction of traces that satisfies the query

- Then, it is possible to show that:

$$\lim_{n\to\infty} \texttt{Sample}_n(\texttt{s; return e}) = [\![\texttt{s; return e}]\!]$$

- In practice, we can't run forever, so we choose some finite n to get an approximation
  - How big should n be?!

When Direct Sampling Works

- ***Hoeffding's inequality*** lets us bound n nicely if there are no observations

As n grows, this probability of error gets smaller exponentially quickly!

- Let $\epsilon > 0$, n be # samples. Then:

$$\Pr\left(\left|[\![\texttt{s; return e}]\!] - \texttt{Sample}_n(\texttt{s; return e})\right| \geq t\right) \leq 2e^{-2nt^2}$$

- For instance, to be within $t = 0.001$ of the true answer with probability $\alpha = 0.99$ requires ~16k samples
  - Logarithmic in α, quadratic in t

When Direct Sampling Fails

- When the probability of accepting a sample is low

```
x ~ flip 0.000001;
y ~ flip 0.000000000001;
z ~ flip 0.0000000000000001;
observe (&& x (&& y z));
return x
```

Conclusion

- You're now ready to do the SimPPL project!

Extra slides

Semantics of IBAL

- Pfeffer, Avi. 2005. The Design and Implementation of IBAL: A General-Purpose Probabilistic Language. Harvard Computer Science Group Technical Report TR-12-05.

1.4 Semantics

In specifying the semantics of the language, it is sufficient to provide semantics for the core expressions, since the syntactic sugar is naturally induced from them. The semantics is distributional: the meaning of a program is specified in terms of a probability distribution over values.

1.4.1 Distributional Semantics

We use the notation $\mathcal{M}[e]$ to denote the meaning of expression e, under the distributional semantics. The meaning function takes as argument a probability distribution over environments. The function returns a probability distribution over values. We write $\mathcal{M}[e]\ \Delta\ v$ to denote the probability of v under the meaning of e when the distribution over environments is Δ. We also use the notation $\mathcal{M}[e]\ \epsilon\ v$ to denote the probability of v under the meaning of e when the probability distribution over environments assigns positive probability only to ϵ.
We now define the meaning function for different types of expressions. The meaning of a constant expression is given by

$$\mathcal{M}[v]\ \Delta\ v' = \begin{cases} 1 & \text{if v' = v} \\ 0 & \text{otherwise} \end{cases}$$

Probability Monad

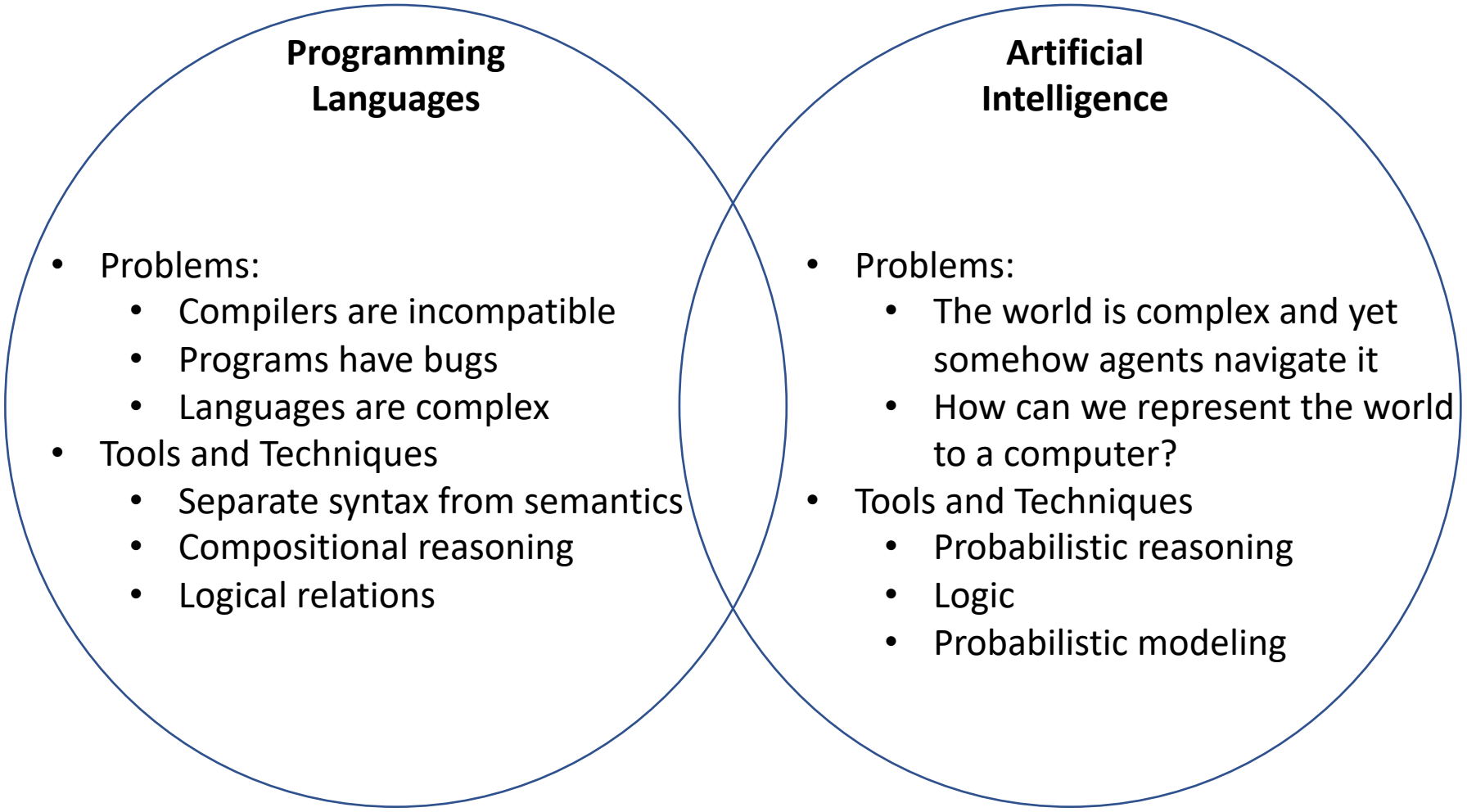
- Ramsey, Norman, and Avi Pfeffer. "Stochastic lambda calculus and monads of probability distributions." Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. 2002.

$$
\begin{aligned}
e ::= {} & x \mid v \mid \lambda x.e \mid e_1\ e_2 \mid \mathbf{let}\ x = e'\ \mathbf{in}\ e \\
\mid {} & \mathbf{choose}\ p\ e_1\ e_2 \\
\mid {} & (e_1, e_2) \mid e.1 \mid e.2 \\
\mid {} & \mathbf{L}\ e_1 \mid \mathbf{R}\ e_2 \mid \mathbf{case}\ e\ e_l\ e_r
\end{aligned}
$$

- Gave a distributional semantics to a stochastic λ-calculus w/ tuples (no observations)

$$
\begin{aligned}
\overline{\mathcal{M}[\![\mathtt{return}\ v]\!](x)} &= \begin{cases} 1, \text{ if } x = v \\ 0, \text{ if } x \neq v \end{cases} \\
\overline{\mathcal{M}[\![d >\!\!>\!= k]\!](x)} &= \textstyle\sum_v \overline{\mathcal{M}[\![d]\!]}(v) \cdot \overline{\mathcal{M}[\![k(v)]\!]}(x) \\
\overline{\mathcal{M}[\![choose\ p\ d_1\ d_2]\!](x)} &= p \cdot \overline{\mathcal{M}[\![d_1]\!]}(x) + (1-p) \cdot \overline{\mathcal{M}[\![d_2]\!]}(x)
\end{aligned}
$$

AI and Programming Languages



Conciseness