Lightweight Implementations of Probabilistic Programming Languages via Transformational Compilation

David Wingate
wingated@mit.edu

Andreas Stuhlmueller
andreas@stuhlmueller.info

Noah Goodman
ngoodman@stanford.edu

Implementing Probabilistic Programming Languages Without the Agonizing Pain

David Wingate
wingated@mit.edu

Andreas Stuhlmueller
andreas@stuhlmueller.info

Noah Goodman
ngoodman@stanford.edu

Summary

Goal: Create **probabilistic programming languages** which help express complex probabilistic models

Observation: We would like to leverage **existing language infrastructure** (compilers, parallelization, profilers, debuggers, etc.)

Contribution: A method to help **transform any language** into a probabilistic programming language

Outline

- **Introduction to Probabilistic Programming**
- **Lightweight Implementations of PPLs**
- **New Inference Options**

Probabilistic Programming

The Big Idea

Use **a programming language** to express your model

1) Write down a function which makes random choices

2) Fix the output of the function (ie, condition the model)

3) Reason about the random choices needed to produce the output **(run the program backwards!)**

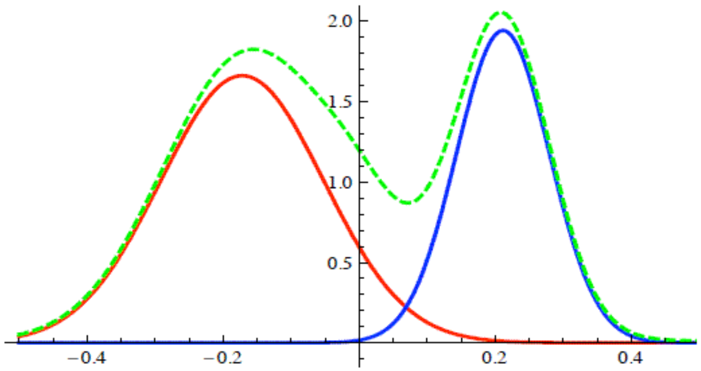
Write a **probability compiler / interpreter** to perform inference

Distributions over Traces

Probabilistic programs define distributions by defining a **distribution over possible execution traces**

The distribution is fully specified by a generative procedure

```
function X = gmm()
  if ( rand > 0.5 )
    X = -0.2 + randn
  else
    X = 0.2 + 0.5*randn
  end;
return;

>> gmm
-0.21
>> gmm
0.3
...
```



Complex distributions are crafted **compositionally**

Example: LDA

```
function X = lda()

  K = 10;
  for k=1:K
    topics(k,:) = dirichlet( 1, vocab_size );
  end;

  for d=1:num_docs
    topic_dist = dirichlet( 1, K );
    for w=1:num_words(d)
      topic = multinomial( topic_dist );
      X{d}(w) = multinomial( topics(topic,:) );
    end;
  end;

return;
```

Nonparametrics

Nonparametric distributions are
implemented with **stochastic
memoization**

Idea:
Given a stochastic function
g
a stochastic memoizer returns a new function
f
which treats **g** as a base measure

Calls to **f** return
a previous or a new value from **g**
according to
the **Dirichlet Process**

```
>> g = @randn;

>> f = dpmem( g, 1.0 );

>> f()
-1.2
>> f()
-0.8
>> f()
-1.2
>> f()
-1.2
```

Example: ((H)DP)MM

```
function X = gaussian_mixture_model()
  K = 3;
  mu = randn( 1, K );

  for i=1:100
    ind = randi( k );
    X(i) = mu( ind ) + randn;
  end;

return;
```

Easy! Fun!

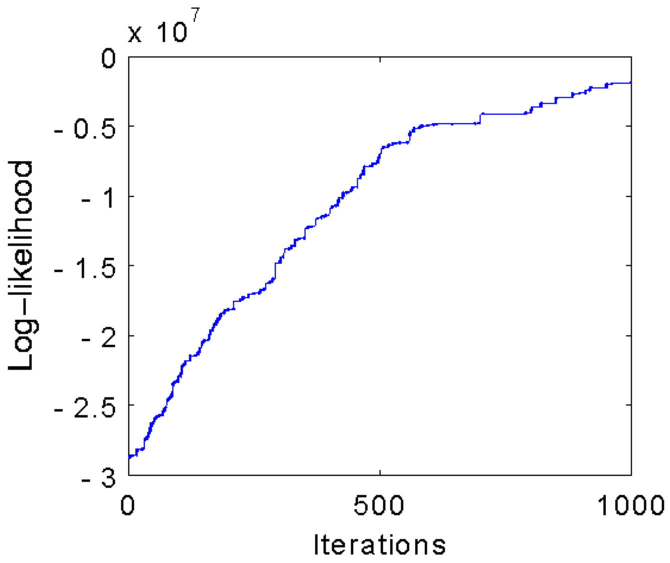
```
function X = dp_mixture_model()

  b = dpmem( 1.0, @randn );

  for i=1:100
    X(i) = b() + randn;
  end;

return;
```

```
function X = hdp_mixture_model()

  a = dpmem( 1.0, @randn );
  b = dpmem( 1.0, @a );

  for i=1:100
    X(i) = b() + randn;
  end;

return;
```

Meta-Modeling

If your language has **eval()** in it, you can reason about **the structure of the model itself**

```
function Y = induce_program( X )

  text_of_code = sample_pcfg();

  f = eval( text_of_code );

  for i=1:100
    Y(i) = f( X(i) ) + randn;
  end;

return;
```



Lightweight PPL Implementations

Inference

Goal: Perform inference in an **arbitrary program**

Appoach: **MCMC**

Need: **Proposals, scoring mechanisms**

Therefore need: The ability to **control program execution**

…without the agonizing pain.

Observation: Execution Trace

```
function X = simple_model()
  m = poissrnd();

  for i=1:m
    X(i) = gammarnd();
  end;

  for i=m+1:2*m
    X(i) = randn();
  end;

return;
```

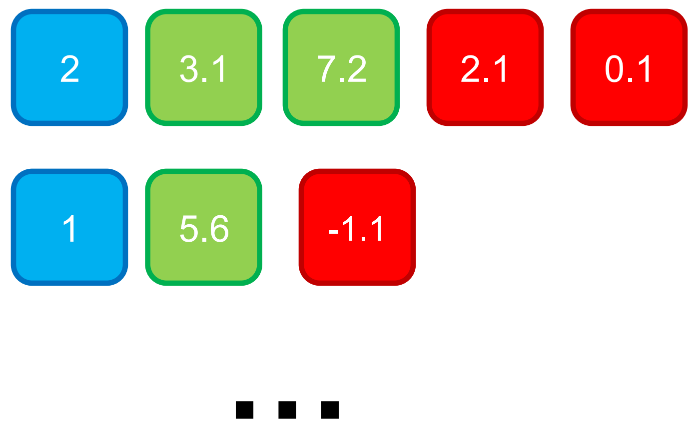
Random Choices Encountered

2 3.1 7.2 2.1 0.1

Observation: Execution Trace

```
function X = simple_model()

  m = poissrnd();

  for i=1:m
    X(i) = gammarnd();
  end;

  for i=m+1:2*m
    X(i) = randn();
  end;

return;
```

Random Choices Encountered



Note: if two traces make all of the same choices, **their execution paths will be the same!**

Transformational Compilation

This suggests the following approach:

1) Give every random choice **a name**
2) Rewrite code to **make it deterministic**
3) When a random choice is encountered, use its name to look up its value in a **database of random values**

We can control execution traces
by manipulating values in the database!

MCMC over Execution Traces

With the database in hand, we can implement inference:

1) Given an execution trace
2) Propose changes to some variables
3) Update the trace, **reusing as much randomness as possible**
4) Score; accept/reject

Initialize: [$ll, \mathbb{D}$] = trace_update($\emptyset$)
Repeat forever:
  Select a random f_k via its name n
  Look up its current value $(t, x, l, \theta_{db}) = \mathbb{D}(n)$.
  Propose a new value $x' \sim \mathcal{K}_t(\cdot|x, \theta_{db})$
  Compute $F = \log \mathcal{K}_t(x'|x, \theta_{db})$
  Compute $R = \log \mathcal{K}_t(x|x', \theta_{db})$
  Compute $l' = \log p_t(x'|\theta_{db})$
  Let $\mathbb{D}' = \mathbb{D}$
  Set $\mathbb{D}'(n) = (t, x', l', \theta_{db})$
  [$ll', \mathbb{D}'$] = trace_update($\mathbb{D}'$);
  if ($\log(\text{rand}) < ll' - ll + R - F$)
    // accept
    $\mathbb{D} = \mathbb{D}'$
    $ll = ll'$
    // clean out unused values from $\mathbb{D}$
  else
    // reject; discard $\mathbb{D}'$
  endif;
end repeat;

But What Name?

How should we name random variables?

Idea: according to their
structural position in the trace

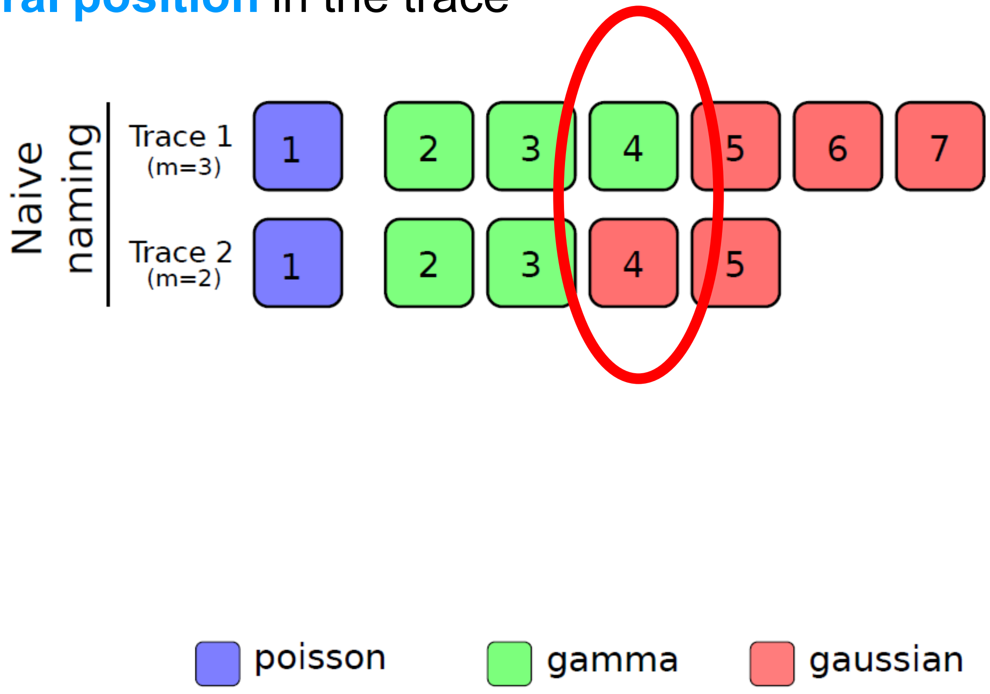
```
function X = simple_model()

  m = poissrnd();

  for i=1:m
    X(i) = gammarnd();
  end;

  for i=m+1:2*m
    X(i) = randn();
  end;

return;
```

But What Name?

How should we name random variables?

Idea: according to their
structural position in the trace

```
function X = simple_model()

  m = poissrnd();

  for i=1:m
    X(i) = gammarnd();
  end;

  for i=m+1:2*m
    X(i) = randn();
  end;

return;
```

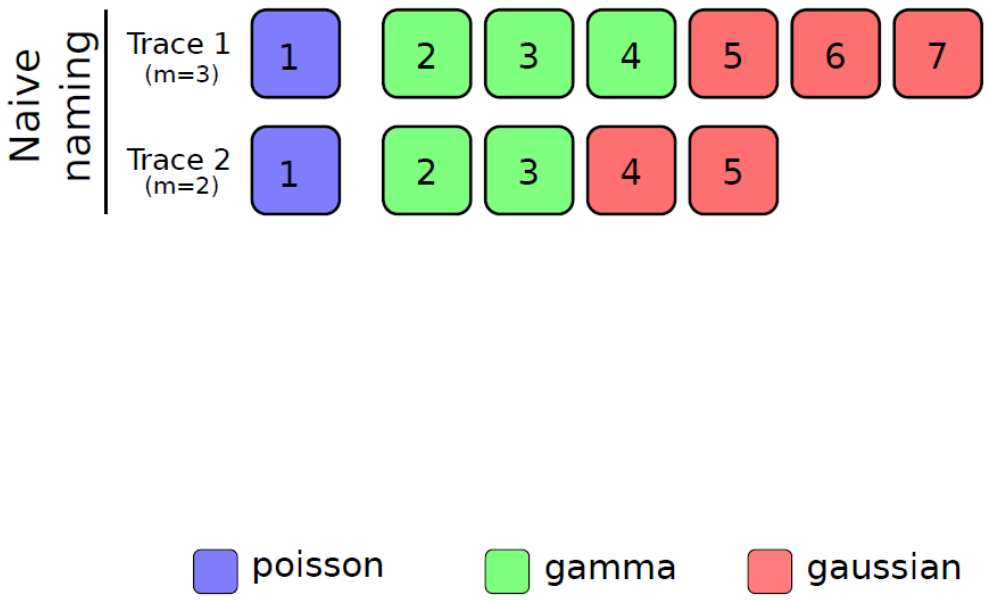


But What Name?

How should we name random variables?

Idea: according to their
structural position in the trace

```
function X = simple_model()

  m = poissrnd();

  for i=1:m
    X(i) = gammarnd();
  end;

  for i=m+1:2*m
    X(i) = randn();
  end;

return;
```



Generating Names

Augment the transformed source with
additional name-generating code

Imperative Naming Specification

- Begin executing *f* with empty function, line, and loop stacks.
- When entering a new function,
  - push a unique function id on the function stack.
  - push a 0 on the line stack.
- When moving to a new line
  - increment the last value on the line stack.
- When starting a loop
  - push a 0 on the loop stack.
- When iterating through a loop
  - increment the last value on the loop stack.
- When exiting a loop
  - pop the loop stack.
- When exiting a function
  - pop the function stack and the line stack.

Stochastic Matlab

Functional Naming Specification

$$\mathcal{A}^{top}[\![E]\!] = \texttt{((lambda (addr) } \mathcal{A}[\![E]\!]\texttt{) '(top))}$$

$$\mathcal{A}[\![\texttt{(lambda (}I_{i=1}^{n}\texttt{) } E_{body}\texttt{)}]\!] = \texttt{(lambda (addr . } I_{i=1}^{n}\texttt{) } \mathcal{A}[\![E_{body}]\!]\texttt{)}$$

where $\mathcal{S}$ is a globally unique symbol.

$$\mathcal{A}[\![\texttt{(mem } E\texttt{)}]\!] = \texttt{((lambda (maddr f) (lambda (addr . args)}$$
$$\texttt{(apply f (cons args maddr) args))) addr } \mathcal{A}[\![E]\!]\texttt{)}$$

$$\mathcal{A}[\![\texttt{(begin } E_{i=1}^{n}\texttt{)}]\!] = \texttt{(begin } \mathcal{A}[\![E_i]\!]_{i=1}^{n}\texttt{)}$$

$$\mathcal{A}[\![\texttt{(letrec ((}I_i\ E_i\texttt{)}_{i=1}^{n}\texttt{) } E_{body}\texttt{)}]\!] = \texttt{(letrec ((}I_i\ \mathcal{A}[\![E_i]\!]\texttt{)}_{i=1}^{n}\texttt{) } \mathcal{A}[\![E_{body}]\!]\texttt{)}$$

$$\mathcal{A}[\![\texttt{(if } E_t\ E_c\ E_a\texttt{)}]\!] = \texttt{(if } \mathcal{A}[\![E_t]\!]\ \mathcal{A}[\![E_c]\!]\ \mathcal{A}[\![E_a]\!]\texttt{)}$$

$$\mathcal{A}[\![\texttt{(define } I\ E\texttt{)}]\!] = \texttt{(define } I\ \mathcal{A}[\![E]\!]\texttt{)}$$

$$\mathcal{A}[\![\texttt{(quote } E\texttt{)}]\!] = \texttt{(quote } E\texttt{)}$$

$$\mathcal{A}[\![\texttt{(}E_{op}\ E_{i=1}^{n}\texttt{)}]\!] = \texttt{(}\mathcal{A}[\![E_{op}]\!]\texttt{ (cons '}\mathcal{S}\texttt{ addr) } \mathcal{A}[\![E_i]\!]_{i=1}^{n}\texttt{)}$$

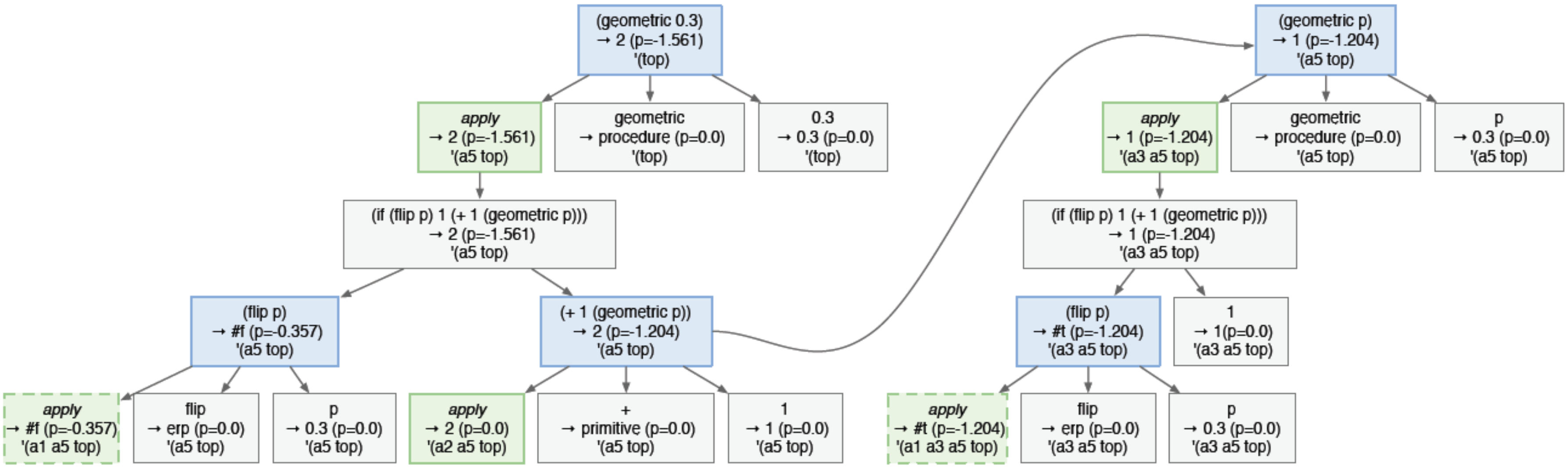
$\mathcal{A}[\![E]\!] = E$, otherwise.

Bher

Example: Geometric

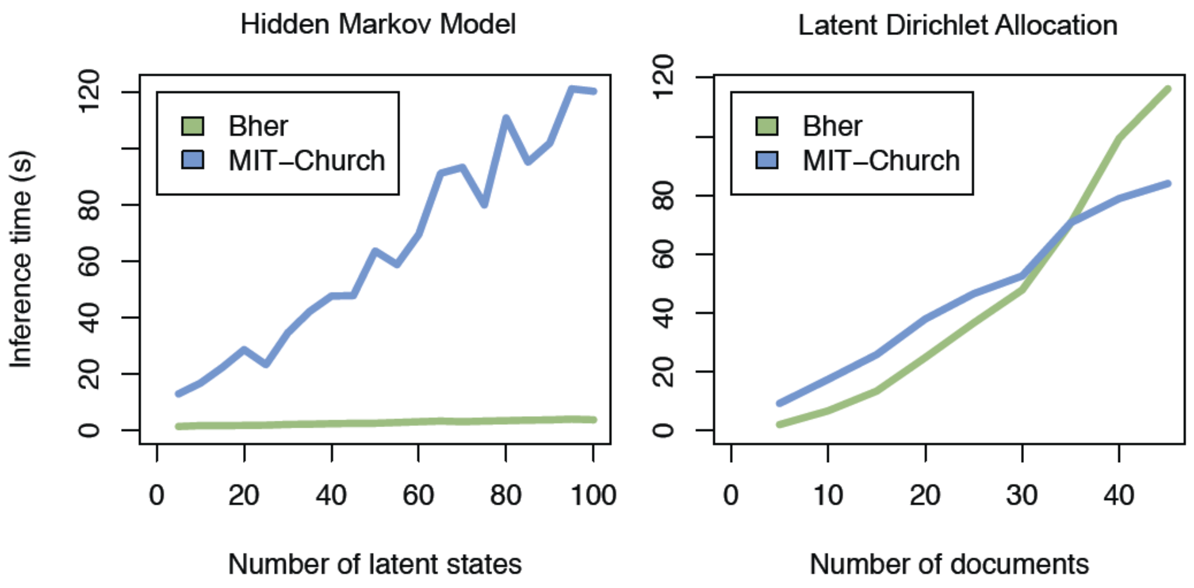
```
(begin
  (define geometric
    (lambda (p)
      (if (flip p)
          1
          (+ 1 (geometric p)))))
  (geometric .7))
```

```
((lambda (addr)
   (begin
     (define geometric
       (lambda (addr p)
         (if (flip (cons 'a1 addr) p)
             1
             (+ (cons 'a2 addr)
                1
                (geometric (cons 'a3 addr) p)))))
     (geometric (cons 'a4 addr) 0.7)))
 '(top))
```



Minimal Interpretative Overhead

Can **improve performance** by leveraging native ecosystem



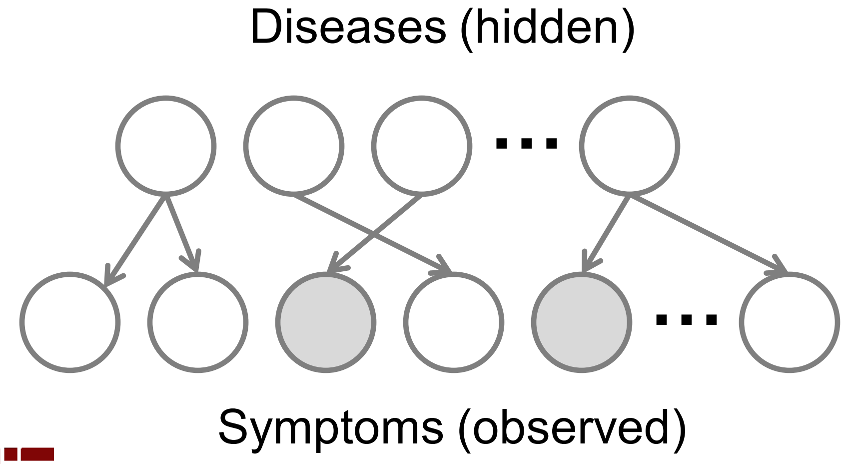
Note: can also simplify implementations
(Bher codebase is **1/10 the size** of MIT-Church)

New Inference Options

Different Inference Options

The model is in **readable/executable format**
Can give the code a **non-standard interpretation**

For example:
automatic differentiation
program analysis to identify known efficient sub-structures
operator overloading

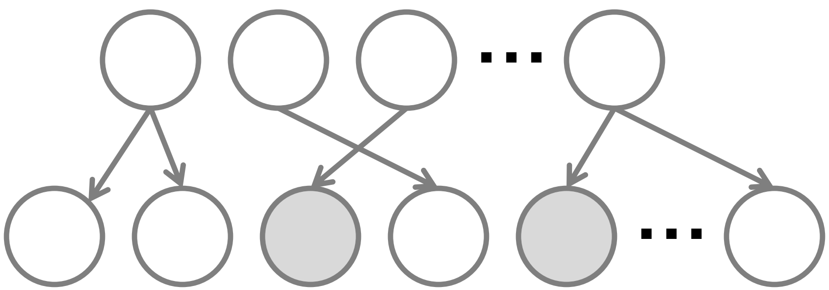


Dynamic Dependency Analysis

Use **operator overloading**
to **dynamically track** fine-grained dependencies

Construct a **good proposal**
by **getting rid of the bad parts** of a big proposal

Diseases (hidden)



Symptoms (observed)

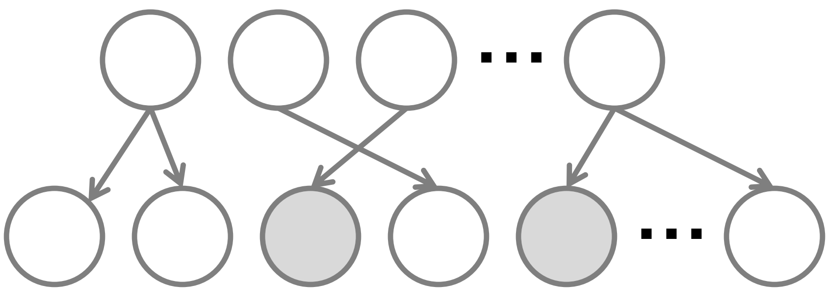
1) Propose lots of changes
2) Track dependency structure
3) Whittle away bad parts!

Dynamic Dependency Analysis

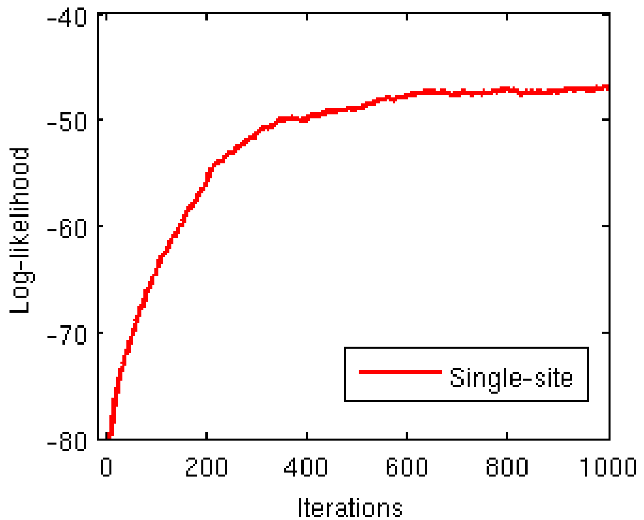
Use **operator overloading**
to **dynamically track** fine-grained dependencies

Construct a **good proposal**
by **getting rid of the bad parts** of a big proposal

Diseases (hidden)



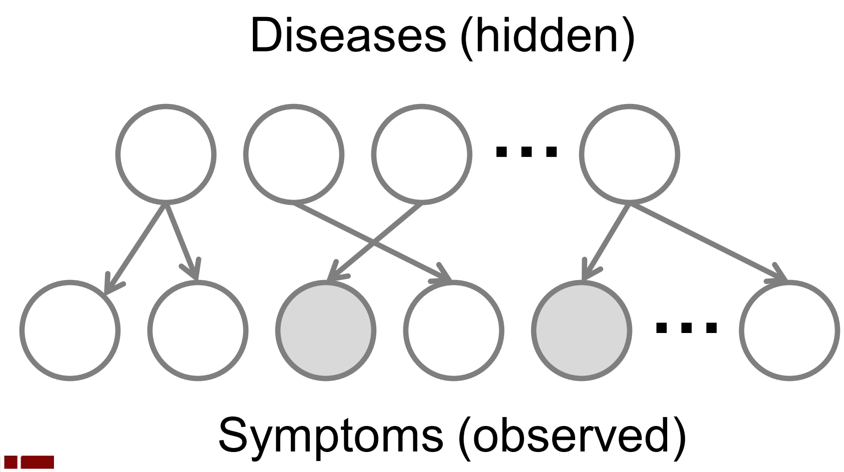
Symptoms (observed)

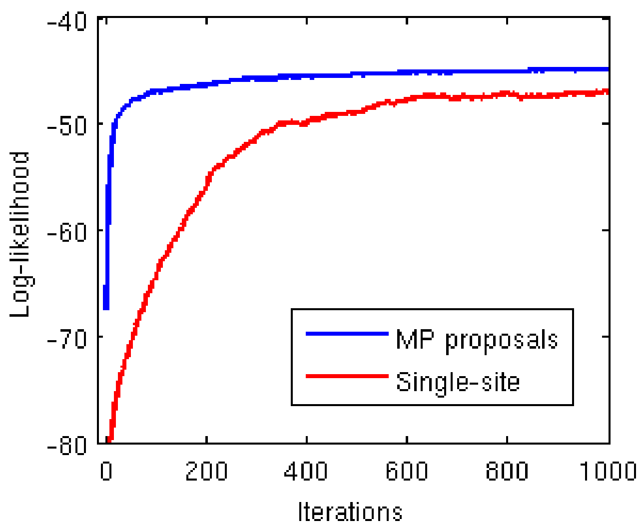


Dynamic Dependency Analysis

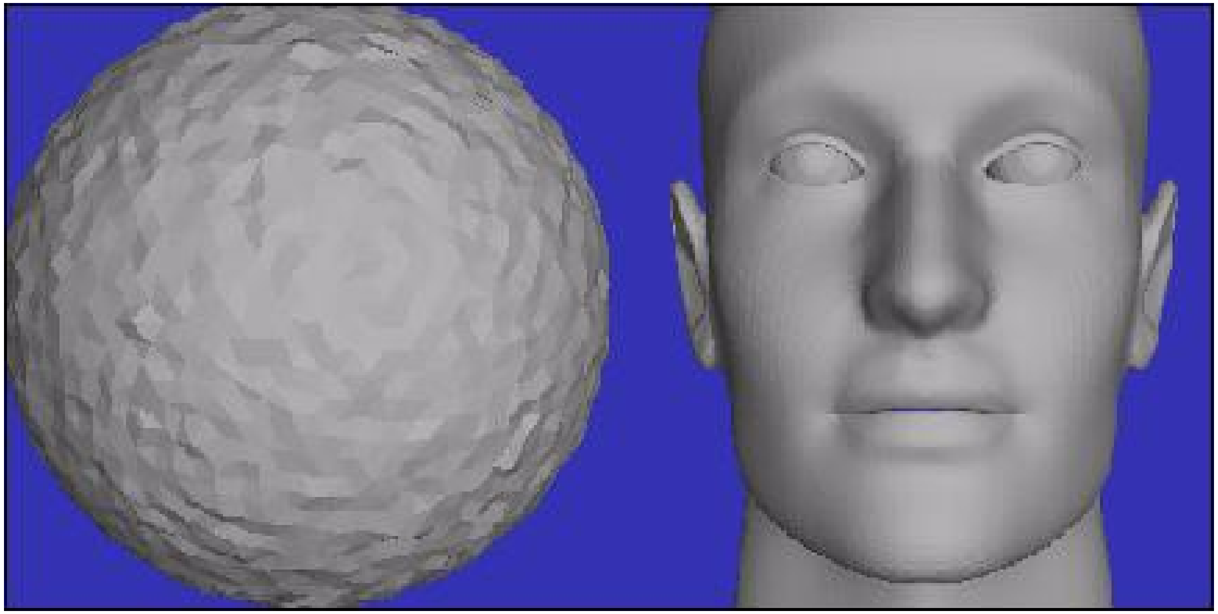
Use **operator overloading**
to **dynamically track** fine-grained dependencies

Construct a **good proposal**
by **getting rid of the bad parts** of a big proposal





Example: mesh inference



```
function X = mesh_inference( base_mesh )
  mesh = base_mesh + randn( size(base_mesh) );
  img = render( mesh );
  X = img + randn(size(img));
return;
```

Summary

- **A generic technique for implementing PPLs**
  - Name random choices
  - Manipulate execution traces via a database of randomness
  - Structural naming conventions
  - Transformational compilation

- **Basis for several language implementations**
  - Functional: Bher
  - Imperative: Stochastic Matlab
  - New: pystoch

- **Machine-readable models**
  - suggests new options for analysis, inference

Thank you!

Stochastic Matlab

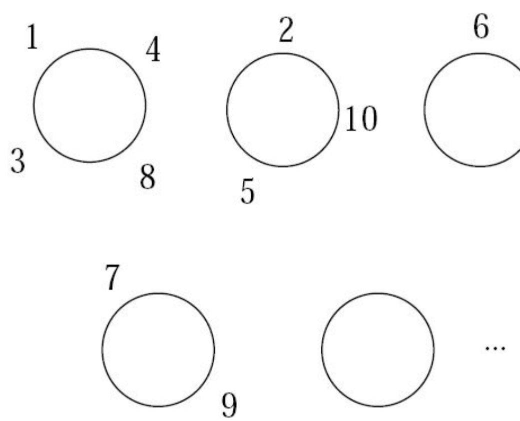
- An open-source project

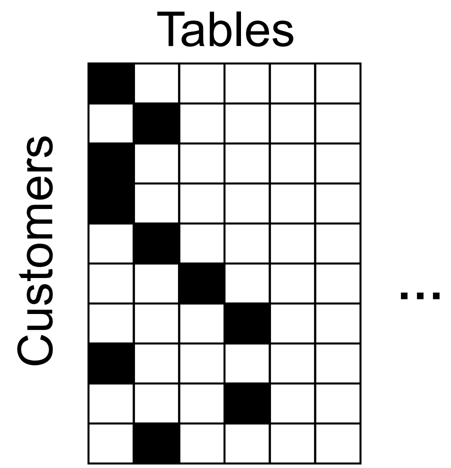
- Sample-based inference
- Freely mix deterministic and stochastic functions
- Can use MEX files
- Some integration with GPU

- Integrated profiling / debugging / visualizations

The Chinese Restaurant Process

A stochastic, generative process which induces a **distribution over partitions**





Properties:

An exchangeable, nonparametric distribution

Can **identify dimensionality of data**

Allows dimensionality to grow with new data (unbounded seating!)

Allows for **parameter sharing**

Classic Model Specification

How do we
specify a probabilistic model?

Currently: a combination of
The English text in the paper
Statistician's notation
Graphical notation

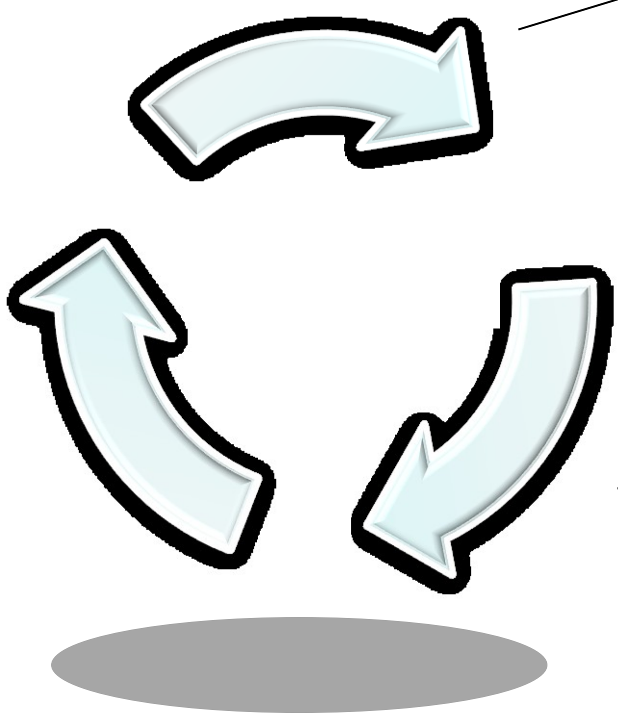
But how do you clearly specify **complex models**?

Typical Design Cycle

Do math

Construct a model

- Come up with equations
- Limited by what you think is tractable



Code up inference

It's different every time

- Does it match your math?
- Want to try 100 different algorithms?
- Hopefully, it's right…

Experiment and analyze

Run Experiments

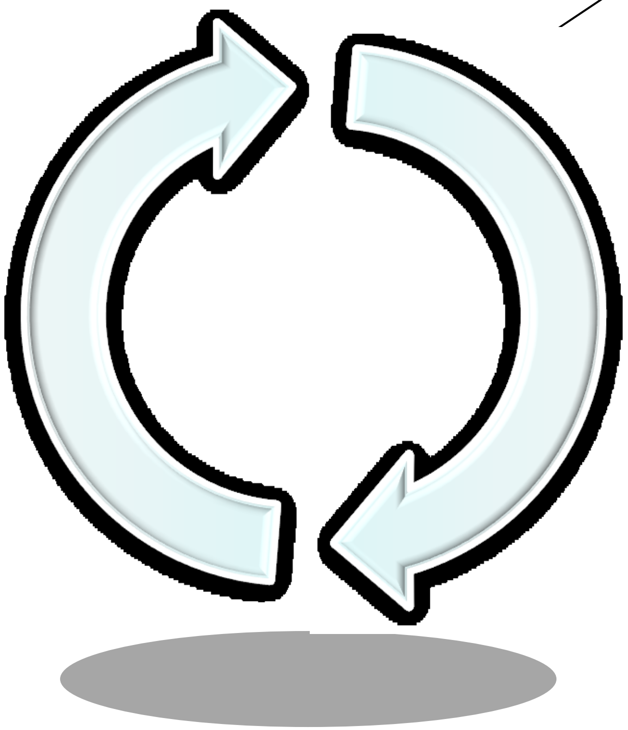
- Is your model right?
- Is your code right?
- Is your data good?

New Design Cycle

Code up your model

A precise specification

- The compiler does the heavy lifting
- Can try multiple inference algorithms (coded by experts!)



Experiment and analyze

Run Experiments

- Is your model right?
- ~~Is your code right?~~
- Is your data good?

PCFGs

```
function X = sample_sentences()
  for i=1:10
    X{i} = [ NP() VP() ];
  end;
return;


function X = NP()
  if ( rand > 0.5 )
    X = [ DET() N() ];
  else
    X = [ DET() ADJ() N() ];
  end;
return;


function X = N()
  nouns = { 'dog', 'cat', 'klein bottle' };
  i = randi( length(nouns) );
  X = nouns{ i };
return;
```

Adaptor Grammars

```
function X = sample_sentences()
  dp_NP = dpmem( 1.0, @NP() );
  for i=1:10
    X{i} = [ dp_NP() VP() ];
  end;
return;

function X = NP()
  if ( rand > 0.5 )
    X = [ DET() N() ];
  else
    X = [ DET() ADJ() N() ];
  end;
return;


function X = N()
  nouns = { 'dog', 'cat', 'klein bottle' };
  i = randi( length(nouns) );
  X = nouns{ i };
return;
```