Monolingual probabilistic programming using generalized coroutines

Oleg Kiselyov  Chung-chieh Shan
FNMOCS Rutgers University
oleg@pobox.com ccshan@cs.rutgers.edu

19 June 2009
This session . . .

programming

formalism
This talk . . .

Modular programming
Expressive formalism
Efficient implementation
This talk ... is about knowledge representation

Modular programming       – Factored representation
Expressive formalism     – Informative prior
Efficient implementation  – Custom inference
This talk . . . is about knowledge representation

Modular programming – Factored representation
Expressive formalism – Informative prior
Efficient implementation – Custom inference
Declarative probabilistic inference

Model (what)  Inference (how)
Declarative probabilistic inference

Model (what)                      Inference (how)

Toolkit  invoke → distributions, (BNT, PFP)        conditionalization, ...

Language random choice, (BLOG, IBAL, observation, ...) ← interpret
    Church

Today: Best of both
Express models and inference as interacting programs
in the same general-purpose language.
Declarative probabilistic inference

<table>
<thead>
<tr>
<th>Model (what)</th>
<th>Inference (how)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Toolkit (BNT, PFP)</td>
<td>+ use existing libraries, types, debugger</td>
</tr>
<tr>
<td>Language (BLOG, IBAL, Church)</td>
<td>+ random variables are ordinary variables</td>
</tr>
</tbody>
</table>
Declarative probabilistic inference

<table>
<thead>
<tr>
<th>Model (what)</th>
<th>Inference (how)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Toolkit (BNT, PFP)</td>
<td>+ use existing libraries, types, debugger</td>
</tr>
<tr>
<td></td>
<td>+ easy to add custom inference</td>
</tr>
<tr>
<td>Language (BLOG, IBAL, Church)</td>
<td>+ random variables are ordinary variables</td>
</tr>
<tr>
<td></td>
<td>+ compile models for faster inference</td>
</tr>
</tbody>
</table>

Today: invoke → interpret
Best of both

Express models and inference as interacting programs in the same general-purpose language.
## Declarative probabilistic inference

<table>
<thead>
<tr>
<th>Model (what)</th>
<th>Inference (how)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Toolkit</strong></td>
<td>use existing libraries, types, debugger</td>
</tr>
<tr>
<td>(BNT, PFP)</td>
<td></td>
</tr>
<tr>
<td><strong>Language</strong></td>
<td>random variables are ordinary variables</td>
</tr>
<tr>
<td>(BLOG, IBAL, Church)</td>
<td></td>
</tr>
</tbody>
</table>

**Today:**

**Best of both**

- Payoff: expressive model
  + models of inference: bounded-rational theory of mind
- Payoff: fast inference
  + deterministic parts of models run *at full speed*
  + importance sampling

Express models and inference as interacting programs in the same general-purpose language.
Outline

▶ Expressivity
  - Memoization
  - Nested inference

Implementation
  - Reifying a model into a search tree
  - Importance sampling with look-ahead

Performance
Grass model

```ocaml
let flip = fun p ->
  dist [(p, true); (1.-.p, false)]
```

Models are ordinary code (in OCaml) using a library function `dist`. 
Let \( \text{flip} = \text{fun } p \rightarrow \text{dist } [(p, \text{true}); (1.-p, \text{false})] \)

Models are ordinary code (in OCaml) using a library function \( \text{dist} \).
Grass model

let flip = fun p ->
  dist [(p, true); (1.-.p, false)]

let cloudy = flip 0.5 in
let rain = flip (if cloudy then 0.8 else 0.2) in
let sprinkler = flip (if cloudy then 0.1 else 0.5) in
let wet_roof = flip 0.7 && rain in
let wet_grass = flip 0.9 && rain ||
  flip 0.9 && sprinkler in
if wet_grass then rain else fail ()

Models are ordinary code (in OCaml) using a library function \texttt{dist}. Random variables are ordinary variables.
Grass model

let flip = fun p ->
    dist [(p, true); (1.-.p, false)]

    let cloudy = flip 0.5 in
    let rain = flip (if cloudy then 0.8 else 0.2) in
    let sprinkler = flip (if cloudy then 0.1 else 0.5) in
    let wet_roof = flip 0.7 && rain in
    let wet_grass = flip 0.9 && rain ||
        flip 0.9 && sprinkler in
    if wet_grass then rain else fail ()

Models are ordinary code (in OCaml) using a library function dist. Random variables are ordinary variables.
Grass model

let flip = fun p ->
    dist [(p, true); (1.-.p, false)]

let grass_model = fun () ->
    let cloudy = flip 0.5 in
    let rain = flip (if cloudy then 0.8 else 0.2) in
    let sprinkler = flip (if cloudy then 0.1 else 0.5) in
    let wet_roof = flip 0.7 && rain in
    let wet_grass = flip 0.9 && rain ||
                    flip 0.9 && sprinkler in
    if wet_grass then rain else fail ()

normalize (exact_reify grass_model)

Models are ordinary code (in OCaml) using a library function dist.
Random variables are ordinary variables.
Inference applies to *thunks* and returns a distribution.
Grass model

```ocaml
let flip = fun p ->
  dist [(p, true); (1.-.p, false)]

let grass_model = fun () ->
  let cloudy    = flip 0.5 in
  let rain      = flip (if cloudy then 0.8 else 0.2) in
  let sprinkler = flip (if cloudy then 0.1 else 0.5) in
  let wet_roof  = flip 0.7 && rain in
  let wet_grass = flip 0.9 && rain ||
                  flip 0.9 && sprinkler in
  if wet_grass then rain else fail ()

normalize (exact_reify grass_model)
```

Models are ordinary code (in OCaml) using a library function \texttt{dist}. Random variables are ordinary variables. Inference applies to \textit{thunks} and returns a distribution. Deterministic parts of models run at full speed.
Models as programs in a general-purpose language

Reuse existing infrastructure!

▶ Rich libraries: lists, arrays, database access, I/O, ...
▶ Type inference
▶ Functions as first-class values
▶ Compiler
▶ Debugger
▶ Memoization

Express Dirichlet processes, etc. (Goodman et al. 2008)

Speed up inference using lazy evaluation, bucket elimination, sampling w/memoization (Pfeffer 2007)
Models as programs in a general-purpose language

Reuse existing infrastructure!

- Rich libraries: lists, arrays, database access, I/O, ...
- Type inference
- Functions as first-class values
- Compiler
- Debugger
- Memoization
  Express Dirichlet processes, etc. (Goodman et al. 2008)
  Speed up inference using lazy evaluation
Models as programs in a general-purpose language

Reuse existing infrastructure!

- Rich libraries: lists, arrays, database access, I/O, ...
- Type inference
- Functions as first-class values
- Compiler
- Debugger
- Memoization
  Express Dirichlet processes, etc. (Goodman et al. 2008)
  Speed up inference using lazy evaluation
    bucket elimination
    sampling w/memoization (Pfeffer 2007)
Choose a coin that is either fair or completely biased for true.

```ocaml
let biased = flip 0.5 in
let coin = fun () -> flip 0.5 || biased in
```

Choose a coin that is either fair or completely biased for true.

\[
\text{let biased} = \text{flip 0.5 in} \\
\text{let coin} = \text{fun () -> flip 0.5 || biased in}
\]

Let \( p \) be the probability that flipping the coin yields true.

What is the probability that \( p \) is at least 0.3?
Choose a coin that is either fair or completely biased for true.

```
let biased = flip 0.5 in
let coin = fun () -> flip 0.5 || biased in
```

Let $p$ be the probability that flipping the coin yields true.

What is the probability that $p$ is at least 0.3?

Answer: 1.

```
at_least 0.3 true (exact_reify coin)
```
Nested *inference*

```
exact_reify (fun () ->
    let biased = flip 0.5 in
    let coin = fun () -> flip 0.5 || biased in

    Let \( p \) be the probability that flipping the coin yields \texttt{true}.

    What is the probability that \( p \) is at least 0.3?
    Answer: 1.

    \atleast 0.3 true (exact_reify coin)
```
Nested inference

```ocaml
exact_reify (fun () ->

Choose a coin that is either fair or completely biased for true.

  let biased = flip 0.5 in
  let coin = fun () -> flip 0.5 || biased in

Let $p$ be the probability that flipping the coin yields true.
Estimate $p$ by flipping the coin twice.
What is the probability that our estimate of $p$ is at least 0.3?
Answer: 7/8.

  at_least 0.3 true (sample 2 coin)

)
Nested inference

exact_reify (fun () ->

Choose a coin that is either fair or completely biased for true.

let biased = flip 0.5 in
let coin = fun () -> flip 0.5 || biased in

Let $p$ be the probability that flipping the coin yields true. Estimate $p$ by flipping the coin twice. What is the probability that our estimate of $p$ is at least 0.3? Answer: 7/8.

at_least 0.3 true (sample 2 coin)

Outline

Expressivity
  Memoization
  Nested inference

► Implementation
  Reifying a model into a search tree
  Importance sampling with look-ahead

Performance
Reifying a model into a search tree

Exact inference by depth-first brute-force enumeration.
Rejection sampling by top-down random traversal.
Reifying a model into a search tree

open

Exact inference by depth-first brute-force enumeration. Rejection sampling by top-down random traversal.
Reifying a model into a search tree

Exact inference by depth-first brute-force enumeration.
Rejection sampling by top-down random traversal.
Reifying a model into a search tree

Exact inference by depth-first brute-force enumeration.
Rejection sampling by top-down random traversal.
Reifying a model into a search tree

Exact inference by depth-first brute-force enumeration. Rejection sampling by top-down random traversal.
Reifying a model into a search tree

Inference procedures cannot access models’ source code. Reify then reflect:

- Brute-force enumeration becomes bucket elimination
- Sampling becomes particle filtering
Reifying a model into a search tree

![Diagram showing reify, open, reflect, and unit -> bool]

**Implementation:** represent a probability and state monad
(Giry 1982, Moggi 1990, Filinski 1994)
using first-class delimited continuations
(Strachey & Wadsworth 1974,
Felleisen et al. 1987,
Danvy & Filinski 1989)

**Implementation:** using clonable user-level threads
- Model runs inside a thread.
- `dist` clones the thread.
- `fail` kills the thread.
- Memoization mutates thread-local storage.
Importance sampling with look-ahead

open

Probability mass $p_c = 1$
Importance sampling with look-ahead

1. Expand one level.

Probability mass $p_c = 1$

```
  closed
     /   \
   .3   .2   .5
 open  false  open
```
Importance sampling with look-ahead

1. Expand one level.
2. Report shallow successes.

Probability mass $p_c = 1$

(.2, false)
Importance sampling with look-ahead

1. Expand one level.
2. Report shallow successes.
3. Expand one more level and tally open probability.
Importance sampling with look-ahead

Probability mass $p_c = .75$

(.2, false)

1. Expand one level.
2. Report shallow successes.
3. Expand one more level and tally open probability.
4. Randomly choose a branch and go back to 2.
Importance sampling with look-ahead

Probability mass \( p_c = 0.75 \)

\((0.2, \text{false}) (0.6, \text{true})\)

1. Expand one level.
2. Report shallow successes.
3. Expand one more level and tally open probability.
4. Randomly choose a branch and go back to 2.
Importance sampling with look-ahead

1. Expand one level.
2. Report shallow successes.
3. Expand one more level and tally open probability.
4. Randomly choose a branch and go back to 2.

Probability mass $p_c = 0$
($0.2$, false) ($0.6$, true)
Importance sampling with look-ahead

Probability mass $p_c = 0$

(.2, false) (.6, true)

1. Expand one level.
2. Report shallow successes.
3. Expand one more level and tally open probability.
4. Randomly choose a branch and go back to 2.
Outline

Expressivity
  Memoization
  Nested inference

Implementation
  Reifying a model into a search tree
  Importance sampling with look-ahead

► Performance
Motivic development in Beethoven sonatas

Source motif

\[
\begin{array}{c}
\text{\scalebox{0.75}{Source motif}} \\
\end{array}
\]
Motivic development in Beethoven sonatas (Pfeffer 2007)

Source motif

\[ \text{\includegraphics[width=\textwidth]{motivic-development}} \]
Motivic development in Beethoven sonatas

(Pfeffer 2007)
Motivic development in Beethoven sonatas

(Pfeffer 2007)
Motivic development in Beethoven sonatas

(Pfeffer 2007)

Implemented using lazy stochastic lists.

Motif pair 1 2 3 4 5 6 7
% correct using importance sampling

Pfeffer 2007 (30 sec) 93 100 28 80 98 100 63

This paper (90 sec) 98 100 29 87 94 100 77

This paper (30 sec) 92 99 25 46 72 95 61
Motivic development in Beethoven sonatas

(Pfeffer 2007)

Implemented using lazy stochastic lists.

<table>
<thead>
<tr>
<th>Motif pair</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>% correct using importance sampling</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pfeffer 2007 (30 sec)</td>
<td>93</td>
<td>100</td>
<td>28</td>
<td>80</td>
<td>98</td>
<td>100</td>
<td>63</td>
</tr>
<tr>
<td>This paper (90 sec)</td>
<td>98</td>
<td>100</td>
<td>29</td>
<td>87</td>
<td>94</td>
<td>100</td>
<td>77</td>
</tr>
<tr>
<td>This paper (30 sec)</td>
<td>92</td>
<td>99</td>
<td>25</td>
<td>46</td>
<td>72</td>
<td>95</td>
<td>61</td>
</tr>
</tbody>
</table>
Motivic development in Beethoven sonatas (Pfeffer 2007)

![Graph showing frequency in 100 trials for IBAL 90 seconds and 30 seconds.](image)
Noisy radar blips for aircraft tracking

(Milch et al. 2007)

Blips present and absent

Probability

Number of planes

Particle filter. Implemented using lazy stochastic coordinates.
Noisy radar blips for aircraft tracking

(Milch et al. 2007)

Blips present and absent

$t = 1$

Probability

Number of planes

Particle filter. Implemented using lazy stochastic coordinates.
Noisy radar blips for aircraft tracking

(number of planes)

Particle filter. Implemented using lazy stochastic coordinates.
Noisy radar blips for aircraft tracking

(Milch et al. 2007)

Blips present and absent

\[ t = 1, \ t = 2, \ t = 3 \]

Number of planes

3 4

Particle filter. Implemented using lazy stochastic coordinates.
## Summary

<table>
<thead>
<tr>
<th>Toolkit</th>
<th>Model (what)</th>
<th>Inference (how)</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ use existing libraries, types, debugger</td>
<td>+ easy to add custom inference</td>
<td></td>
</tr>
</tbody>
</table>

| Language | + random variables are ordinary variables | + compile models for faster inference |

### Today:
Best of both

Payoff: expressive model
+ models of inference: bounded-rational theory of mind

Payoff: fast inference
+ deterministic parts of models run at full speed
+ importance sampling

Express models and inference as interacting programs in the same general-purpose language.