

Embedded probabilistic programming

Oleg Kiselyov
FNMOC
oleg@pobox.com

Chung-chieh Shan
Rutgers University
ccshan@cs.rutgers.edu

17 July 2009

Probabilistic inference

Model (what)

Inference (how)

$\Pr(\text{Reality})$

$\Pr(\text{Obs} \mid \text{Reality})$

obs

} $\Pr(\text{Reality} \mid \text{Obs} = \text{obs})$

||

$$\frac{\Pr(\text{Obs} = \text{obs} \mid \text{Reality}) \Pr(\text{Reality})}{\Pr(\text{Obs} = \text{obs})}$$

Declarative probabilistic inference

Model (what)

$\Pr(\text{Reality})$
 $\Pr(\text{Obs} \mid \text{Reality})$
 obs

Inference (how)

$$\left. \begin{array}{l} \Pr(\text{Reality}) \\ \Pr(\text{Obs} \mid \text{Reality}) \\ \text{obs} \end{array} \right\} \Pr(\text{Reality} \mid \text{Obs} = \text{obs})$$
$$\parallel$$
$$\frac{\Pr(\text{Obs} = \text{obs} \mid \text{Reality}) \Pr(\text{Reality})}{\Pr(\text{Obs} = \text{obs})}$$

Declarative probabilistic inference

	Model (what)	Inference (how)
Toolkit (BNT, PFP)	invoke →	distributions, conditionalization, ...
Language (BLOG, IBAL, Church)	random choice, observation, ...	← interpret

Declarative probabilistic inference

	Model (what)	Inference (how)
Toolkit (BNT, PFP)	+ use existing libraries, types, debugger	+ easy to add custom inference
Language (BLOG, IBAL, Church)	+ random variables are ordinary variables	+ compile models for faster inference

Declarative probabilistic inference

	Model (what)	Inference (how)
Toolkit (BNT, PFP)	+ use existing libraries, types, debugger	+ easy to add custom inference
Language (BLOG, IBAL, Church)	+ random variables are ordinary variables	+ compile models for faster inference

Today:
Best of both

← interpret

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

Declarative probabilistic inference

	Model (what)	Inference (how)
Toolkit (BNT, PFP)	+ use existing libraries, types, debugger	+ easy to add custom inference
Language (BLOG, IBAL, Church)	+ random variables are ordinary variables	+ compile models for faster inference
Today: Best of both	Payoff: expressive model + models <i>of inference</i> : bounded-rational theory of mind	Payoff: fast inference + deterministic parts of models run <i>at full speed</i> + importance sampling

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

Outline

► **Expressivity**

Nested inference

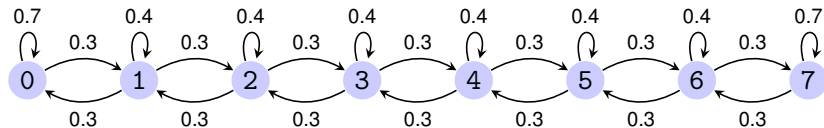
Implementation

Reifying a model into a search tree

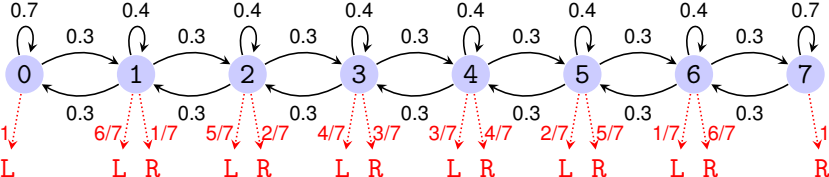
Importance sampling with look-ahead

Performance

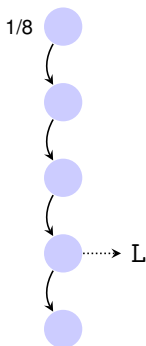
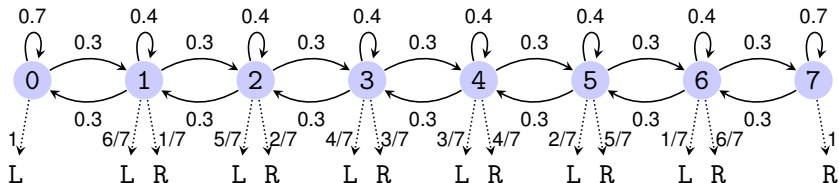
Hidden Markov model



Hidden Markov model

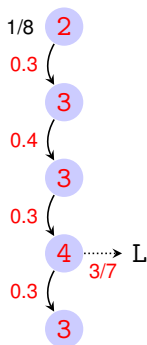
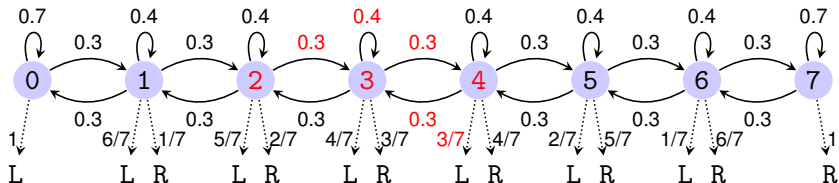


Hidden Markov model



$$\Pr(\text{State}_5 \mid \text{Obs}_4 = \text{L})$$

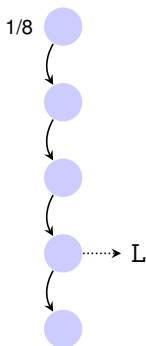
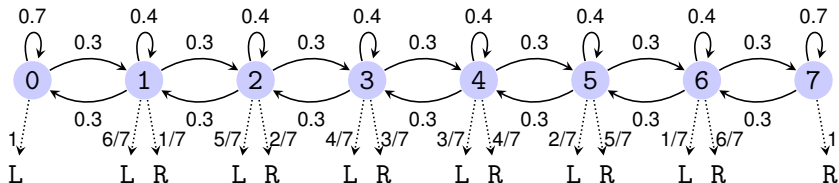
Hidden Markov model



$$\Pr(\text{State}_5 \mid \text{Obs}_4 = L)$$

Hidden Markov model

```
type state = int    type obs = L | R    let nstates = 8
```



$\Pr(\text{State}_5 \mid \text{Obs}_4 = L)$

Hidden Markov model

```
type state = int    type obs = L | R    let nstates = 8
let transition_prob = [| [(0.7,0); (0.3,1)]; ... |]
let evolve : state -> state = fun st ->
    dist (transition_prob.(st))
let observe : state -> obs = fun st ->
    let p = float st /. float (nstates - 1) in
    dist [(1.-.p, L); (p, R)]
let rec run = fun n obs ->
    let st = if n = 1 then uniform nstates else
              evolve (run (n - 1) obs) in
    obs st n; st
run 5
(fun st n -> if n = 4 && observe st <> L then fail ())
```

Models are ordinary code (in OCaml) using a library function `dist`.

Hidden Markov model

```
type state = int    type obs = L | R    let nstates = 8
let transition_prob = [| [(0.7,0); (0.3,1)]; ... |]
let evolve : state -> state = fun st ->
  dist (transition_prob.(st))
let observe : state -> obs = fun st ->
  let p = float st /. float (nstates - 1) in
  dist [(1.-.p, L); (p, R)]
let rec run = fun n obs ->
  let st = if n = 1 then uniform nstates else
           evolve (run (n - 1) obs) in
  obs st n; st
run 5
(fun st n -> if n = 4 && observe st <> L then fail ())
```

Models are ordinary code (in OCaml) using a library function `dist`.

Hidden Markov model

```
type state = int    type obs = L | R    let nstates = 8
let transition_prob = [| [(0.7,0); (0.3,1)]; ... |]
let evolve : state -> state = fun st ->
  dist (transition_prob.(st))
let observe : state -> obs = fun st ->
  let p = float st /. float (nstates - 1) in
  dist [(1.-.p, L); (p, R)]
let rec run = fun n obs ->
  let st = if n = 1 then uniform nstates else
           evolve (run (n - 1) obs) in
  obs st n; st
run 5
(fun st n -> if n = 4 && observe st <> L then fail ())
```

Models are ordinary code (in OCaml) using a library function `dist`.

Hidden Markov model

```
type state = int    type obs = L | R    let nstates = 8
let transition_prob = [| [(0.7,0); (0.3,1)]; ... |]
let evolve : state -> state = fun st ->
  dist (transition_prob.(st))
let observe : state -> obs = fun st ->
  let p = float st /. float (nstates - 1) in
  dist [(1.-p, L); (p, R)]
let rec run = fun n obs ->
  let st = if n = 1 then uniform nstates else
           evolve (run (n - 1) obs) in
  obs st n; st
normalize (exact_reify (fun () -> run 5
  (fun st n -> if n = 4 && observe st <> L then fail ())))
```

Models are ordinary code (in OCaml) using a library function `dist`.
Inference applies to a thunk and returns a distribution.

Hidden Markov model

```
type state = int    type obs = L | R    let nstates = 8
let transition_prob = [| [(0.7,0); (0.3,1)]; ... |]
let evolve : state -> state = fun st ->
  dist (transition_prob.(st))
let observe : state -> obs = fun st ->
  let p = float st /. float (nstates - 1) in
  dist [(1.-.p, L); (p, R)]
let rec run = fun n obs ->
  let st = if n = 1 then uniform nstates else
    evolve (run (n - 1) obs) in
  obs st n; st
normalize (exact_reify (fun () -> run 5
  (fun st n -> if n = 4 && observe st <> L then fail ())))
```

Models are ordinary code (in OCaml) using a library function `dist`.

Inference applies to a thunk 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)

Hidden Markov model

```
type state = int    type obs = L | R    let nstates = 8
let transition_prob = [| [(0.7,0); (0.3,1)]; ... |]
let evolve : state -> state = fun st ->
  dist (transition_prob.(st))
let observe : state -> obs = fun st ->
  let p = float st /. float (nstates - 1) in
  dist [(1.-.p, L); (p, R)]
let rec run = fun n obs ->
  let st = if n = 1 then uniform nstates else
    evolve (run (n - 1) obs) in
  obs st n; st
normalize (exact_reify (fun () -> run 5
  (fun st n -> if n = 4 && observe st <> L then fail ())))
```

Hidden Markov model

```
type state = int    type obs = L | R    let nstates = 8
let transition_prob = [| [(0.7,0); (0.3,1)]; ... |]
let evolve : state -> state = fun st ->
  dist (transition_prob.(st))
let observe : state -> obs = fun st ->
  let p = float st /. float (nstates - 1) in
  dist [(1.-.p, L); (p, R)]
let rec run = fun n obs ->
  let st = if n = 1 then uniform nstates else
    evolve (run (n - 1) obs) in
  obs st n; st
normalize (exact_reify (fun () -> run 5
  (fun st n -> if n = 4 && observe st <> L then fail ())))
```

Hidden Markov model

```
type state = int    type obs = L | R    let nstates = 8
let transition_prob = [| [(0.7,0); (0.3,1)]; ... |]
let evolve : state -> state = fun st ->
  dist (transition_prob.(st))
let observe : state -> obs = fun st ->
  let p = float st /. float (nstates - 1) in
  dist [(1.-.p, L); (p, R)]
let rec run = fun n obs ->
  let st = if n = 1 then uniform nstates else
    evolve (dist (exact_reify (fun () ->
      run (n - 1) obs))) in
  obs st n; st
normalize (exact_reify (fun () -> run 5
  (fun st n -> if n = 4 && observe st <> L then fail ())))
```

Self application: nested *inference*

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
```

Self application: nested *inference*

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?

Self application: nested *inference*

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)
```

Self application: 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.

What is the probability that p is at least 0.3?

Answer: 1.

```
    at_least 0.3 true (exact_reify coin) )
```

Self application: 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) )
```

Self application: 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) )
```

Returns a distribution—not just nested query (Goodman et al. 2008).

Inference procedures are OCaml code using `dist`, like models.

Works with observation, recursion, memoization.

Bounded-rational theory of mind **without interpretive overhead**.

Outline

Expressivity

Nested inference

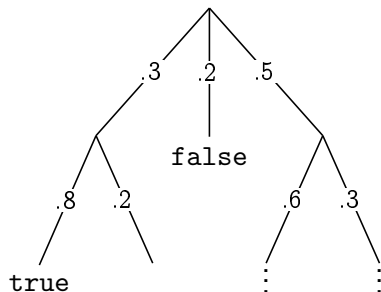
► **Implementation**

Reifying a model into a search tree

Importance sampling with look-ahead

Performance

Reifying a model into a search tree



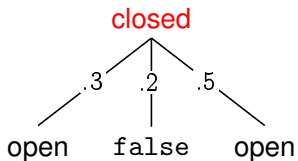
Exact inference by depth-first brute-force enumeration.
Rejection sampling by top-down random traversal.

Reifying a model into a search tree

open

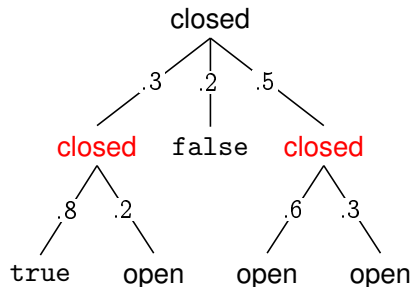
Exact inference by depth-first brute-force enumeration.
Rejection sampling by top-down random traversal.

Reifying a model into a search tree



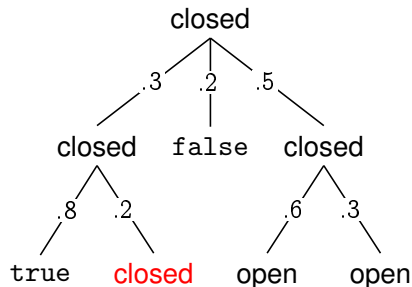
Exact inference by depth-first brute-force enumeration.
Rejection sampling by top-down random traversal.

Reifying a model into a search tree



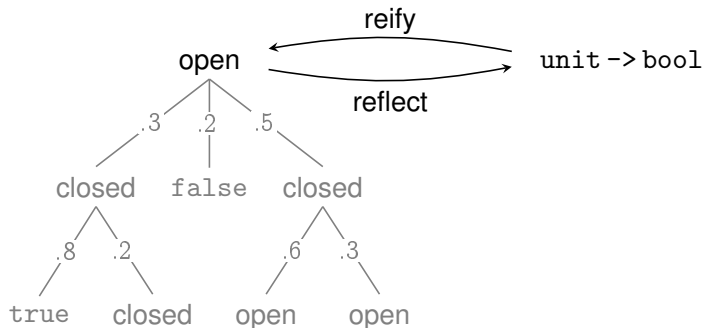
Exact inference by depth-first brute-force enumeration.
Rejection sampling by top-down random traversal.

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

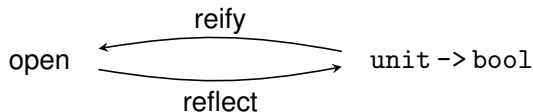


Inference procedures cannot access models' source code.

Reify then reflect (materialized views):

- ▶ Brute-force enumeration becomes bucket elimination
- ▶ Sampling becomes particle filtering

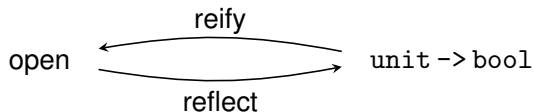
Reifying a model into a search tree



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)

Reifying a model into a search tree



Implementation: shallow DSL embedding

```
let dist ch    = List.map ... ch
let literal x  = unit x
let app e0 e1  = bind e0 (fun f -> bind e1 (fun x -> f x))
```

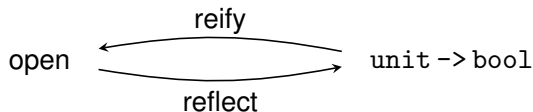
Continuation-passing style

```
let dist ch    = fun k -> List.map ...k... ch
let literal x  = fun k -> k x
let app e0 e1  = fun k -> e0 (fun f -> e1 (fun x -> f x k))
```

First-class delimited continuations

```
let dist ch    = shift (fun k -> List.map ...k... ch)
let literal x  = x
let app e0 e1  = e0 e1
```

Reifying a model into a search tree



Implementation: using clonable user-level threads

- ▶ Model runs inside a thread.
- ▶ `dist` clones the thread.
- ▶ `fail` kills the thread.
- ▶ Memoization mutates thread-local storage.

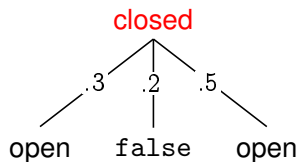
Analogy: Virtualize (not emulate) a CPU. Nesting works.

Importance sampling with look-ahead

open

Probability mass $p_c = 1$

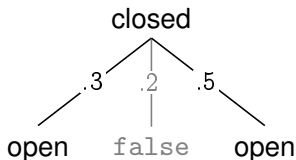
Importance sampling with look-ahead



Probability mass $p_c = 1$

1. Expand one level.

Importance sampling with look-ahead

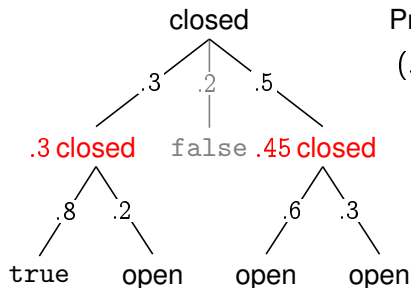


Probability mass $p_c = 1$

(.2, false)

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

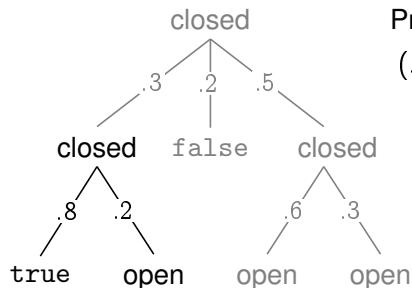
Importance sampling with look-ahead



Probability mass $p_c = .75$
(.2, false)

1. Expand one level.
2. Report shallow successes.
3. Expand one more level and tally open probability.

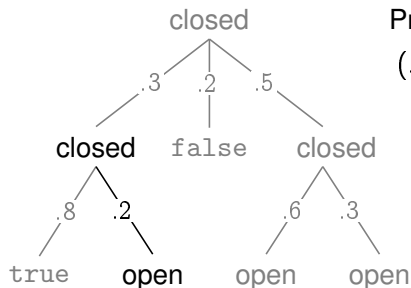
Importance sampling with look-ahead



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

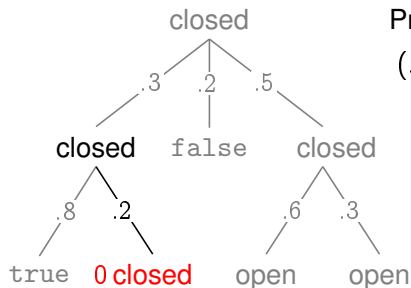


Probability mass $p_c = .75$

(.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.

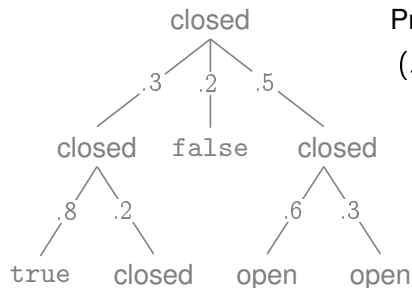
Importance sampling with look-ahead



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.

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

Nested inference

Implementation

Reifying a model into a search tree

Importance sampling with look-ahead

► Performance

Motivic development in Beethoven sonatas

(Pfeffer 2007)



Motivic development in Beethoven sonatas

(Pfeffer 2007)

Source motif

The image shows a musical staff in G major (one sharp) with a treble clef. The notes are G4, A4, B4, G4, A4, B4, C#5, B4, A4, G4. The notes are grouped into two main sections by large brackets below the staff. The first section (G4-A4-B4-G4) is further divided into two sub-sections by smaller brackets: (G4-A4) and (B4-G4). The second section (A4-B4-C#5-B4-A4-G4) is also divided into two sub-sections: (A4-B4-C#5-B4) and (A4-G4). The C#5 note is marked with a sharp sign.

Motivic development in Beethoven sonatas

(Pfeffer 2007)

Source motif

The image displays two staves of musical notation in treble clef. The top staff shows a sequence of notes: a quarter note G4, followed by a quarter note A4, a quarter note B4, a quarter note C5, a quarter note D5, a quarter note E5, a quarter note F#5, a quarter note G5, and a quarter note A5. The notes A4, B4, C5, and D5 are highlighted in red. Red brackets are drawn under the red notes, and black brackets are drawn under the remaining notes. The bottom staff shows the same sequence of notes, but only the notes G4, E5, F#5, and G5 are present, representing a development of the source motif.

Motivic development in Beethoven sonatas

(Pfeffer 2007)

Source motif

The image displays two musical staves in treble clef. The top staff, labeled "Source motif", contains a sequence of notes: G4, A4, B4, G4, A4, B4, C#5, B4, A4. Brackets are used to group these notes: a black bracket under the first four notes (G4, A4, B4, G4), a red bracket under the last four notes (A4, B4, C#5, B4), and a black bracket under the entire nine-note sequence. The bottom staff shows a development of this motif, starting with a single G4 note, followed by a red sequence of notes: B4, C#5, B4, A4, G4.

Motivic development in Beethoven sonatas

(Pfeffer 2007)

Source motif

↑ infer

Destination motif

The image displays two musical staves in treble clef. The top staff, labeled 'Source motif', contains a sequence of notes: G4, A4, B4, C5, B4, A4, G4, F#4, G4, A4. Brackets below the staff group these notes into three segments: the first two notes (G4, A4), the next two (B4, C5), and the final four (B4, A4, G4, F#4, G4, A4). The bottom staff, labeled 'Destination motif', contains a sequence of notes: G4, A4, B4, C5, B4, A4, G4. An upward-pointing arrow labeled 'infer' connects the 'Destination motif' to the 'Source motif', indicating the process of inferring the source from the destination.

Motivic development in Beethoven sonatas

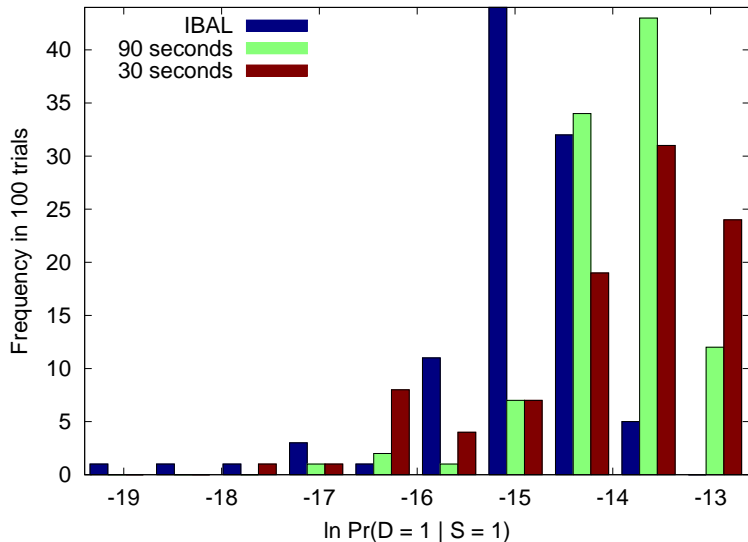
(Pfeffer 2007)

Source motif

Destination motif

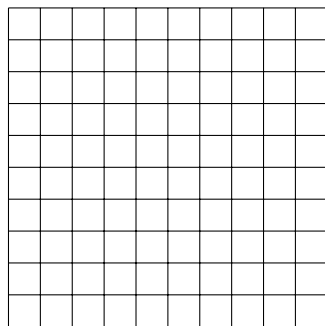
Implemented using lazy stochastic lists.

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



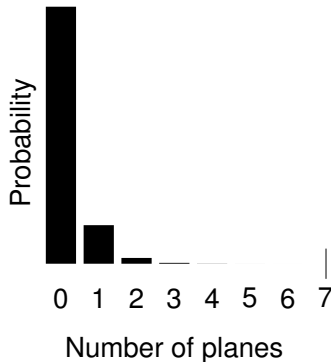
Noisy radar blips for aircraft tracking

(Milch et al. 2007)

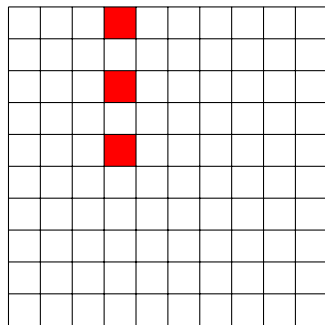


Blips present and absent

infer



Particle filter. Implemented using lazy stochastic coordinates.

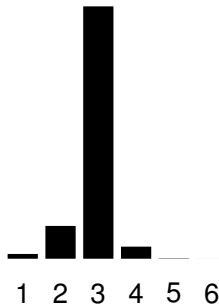


Blips present and absent

$t = 1$

infer

Probability

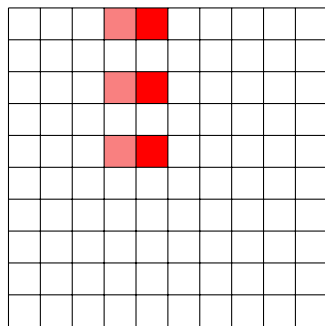


Number of planes

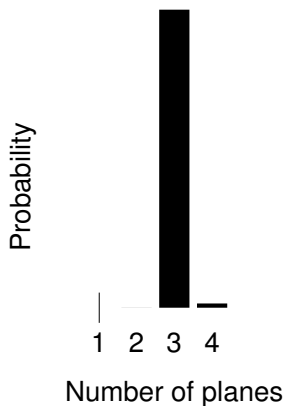
Particle filter. Implemented using lazy stochastic coordinates.

Noisy radar blips for aircraft tracking

(Milch et al. 2007)



infer



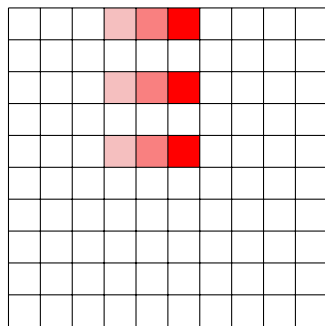
Blips present and absent

$t = 1, t = 2$

Particle filter. Implemented using lazy stochastic coordinates.

Noisy radar blips for aircraft tracking

(Milch et al. 2007)



infer

Probability



Blips present and absent

$t = 1, t = 2, t = 3$

Number of planes

Particle filter. Implemented using lazy stochastic coordinates.

Declarative probabilistic inference

	Model (what)	Inference (how)
Toolkit (BNT, PFP)	+ use existing libraries, types, debugger	+ easy to add custom inference
Language (BLOG, IBAL, Church)	+ random variables are ordinary variables	+ compile models for faster inference
Today: Best of both	Payoff: expressive model + models <i>of inference</i> : bounded-rational theory of mind	Payoff: fast inference + deterministic parts of models run <i>at full speed</i> + importance sampling

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