

# Self-applicable probabilistic inference without interpretive overhead

Oleg Kiselyov  
FNMOC  
oleg@pobox.com

Chung-chieh Shan  
Rutgers University  
ccshan@rutgers.edu

Tufts University  
12 February 2010

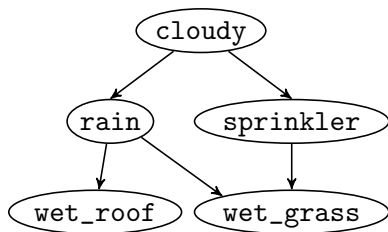
# Probabilistic inference

Model (what)

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

Inference (how)

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



$\Pr(\text{rain} \mid \text{wet\_grass} = \text{true})$

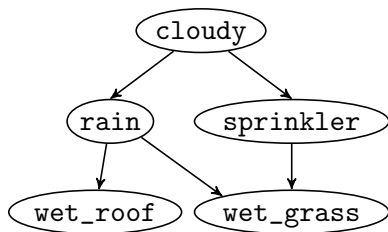
# Declarative probabilistic inference

Model (what)

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

Inference (how)

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



$\Pr(\text{rain} \mid \text{wet\_grass} = \text{true})$

## 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
<b>Today:</b> <b>Best of both</b>	<b>Payoff: expressive model</b> + models <i>of inference</i> : bounded-rational theory of mind	<b>Payoff: fast inference</b> + 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

- Memoization

- Nested inference

## Implementation

- Reifying a model into a search tree

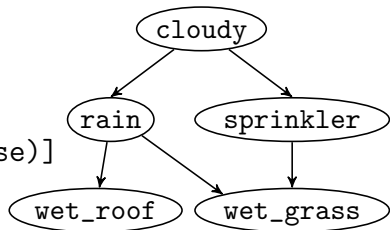
- Importance sampling with look-ahead

## Applications



## Grass model

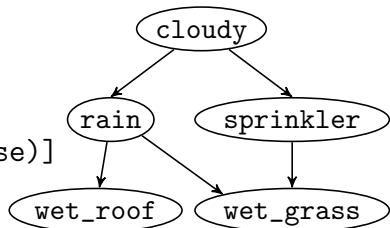
```
let flip = fun p ->  
  dist [(p, true); (1.-.p, false)]
```



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

## Grass model

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

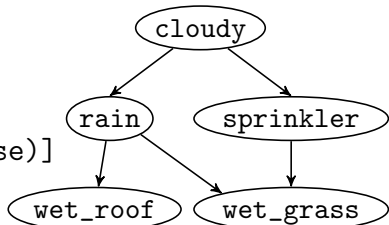


Models are ordinary code (in OCaml) using a library function `dist`.

## 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 ()
```

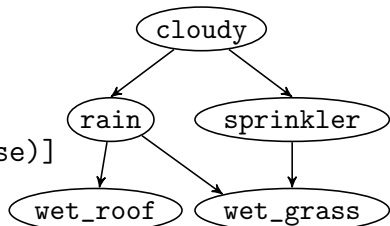


Models are ordinary code (in OCaml) using a library function `dist`.  
Random variables are ordinary variables.

## 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
```

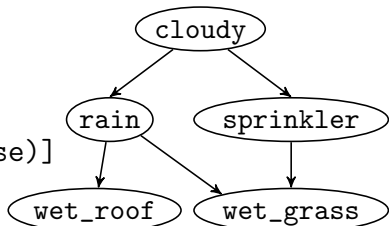
```
  let sprinkler   = flip (if cloudy then 0.1 else 0.5) in
```

```
  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.

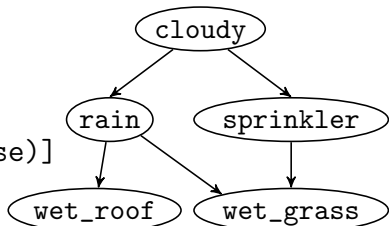
Inference applies to *thunks* and returns a distribution.

## Grass model

```
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.

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

# 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)



## 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.

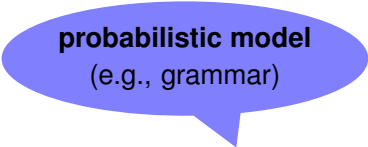
```
  at_least 0.3 true (sample 2 coin) )
```

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

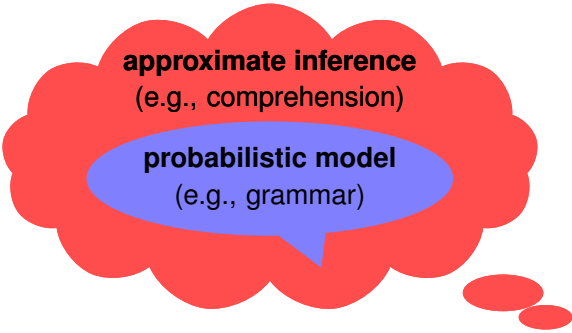
Inference procedures are OCaml code using `dist`, like models.

Works with observation, recursion, memoization.

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



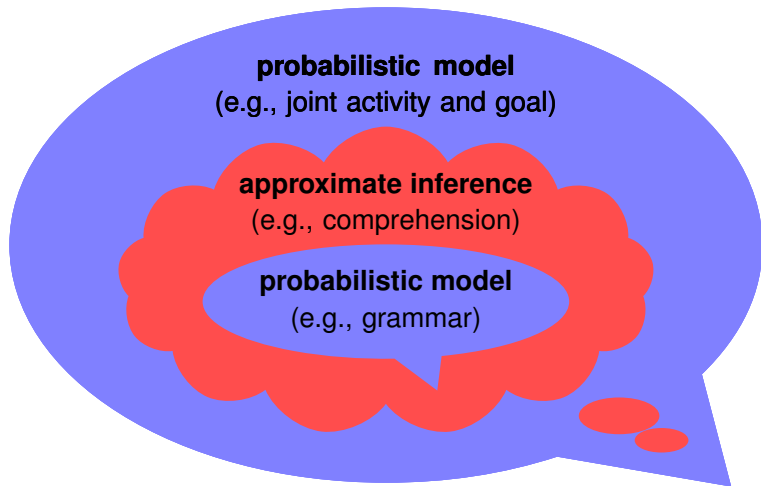
**probabilistic model**  
(e.g., grammar)



**approximate inference**  
(e.g., comprehension)

**probabilistic model**  
(e.g., grammar)





**approximate inference**  
(e.g., plan utterance)

**probabilistic model**  
(e.g., joint activity and goal)

**approximate inference**  
(e.g., comprehension)

**probabilistic model**  
(e.g., grammar)

# Outline

## Expressivity

Memoization

Nested inference

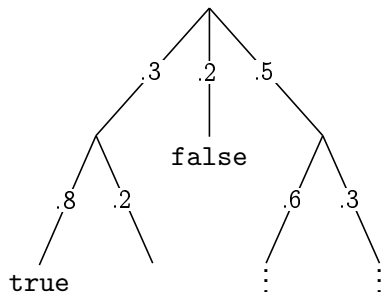
## ► **Implementation**

Reifying a model into a search tree

Importance sampling with look-ahead

## Applications

## Reifying a model into a search tree



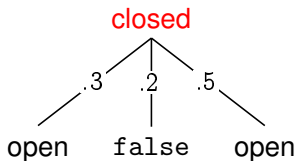
Exact inference by depth-first brute-force enumeration.  
Rejection sampling by top-down random traversal.

## Reifying a model into a search tree

open

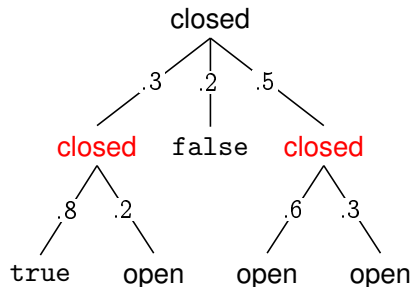
Exact inference by depth-first brute-force enumeration.  
Rejection sampling by top-down random traversal.

## Reifying a model into a search tree



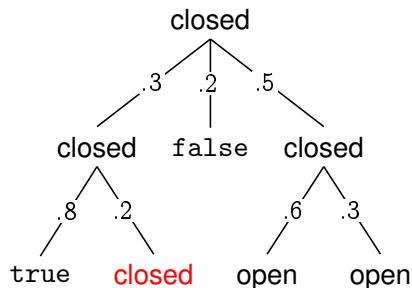
Exact inference by depth-first brute-force enumeration.  
Rejection sampling by top-down random traversal.

## Reifying a model into a search tree



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

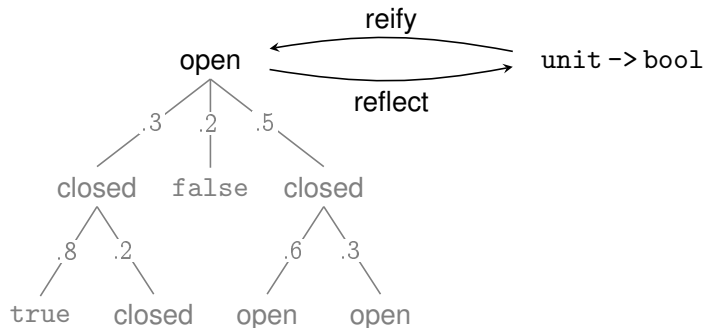
## Reifying a model into a search tree



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



## Reifying a model into a search tree

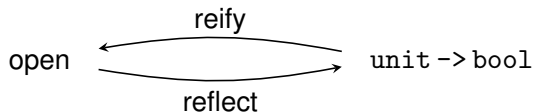


Inference procedures cannot access models' source code.

Reify then reflect:

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

## Reifying a model into a search tree



**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.

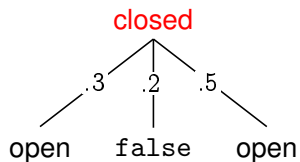
Analogy: Virtualize (not emulate) a CPU. Nesting works.

## Importance sampling with look-ahead

open

Probability mass  $p_c = 1$

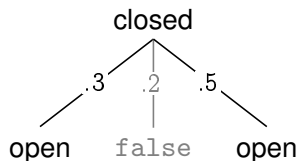
## Importance sampling with look-ahead



Probability mass  $p_c = 1$

1. Expand one level.

## Importance sampling with look-ahead

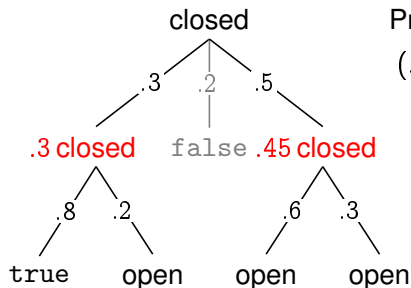


Probability mass  $p_c = 1$

**(.2, false)**

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

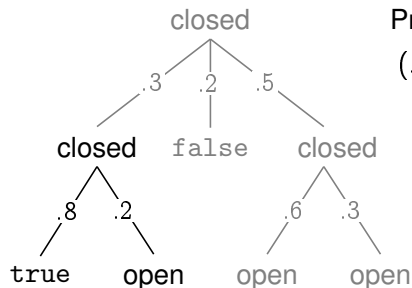
## Importance sampling with look-ahead



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

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

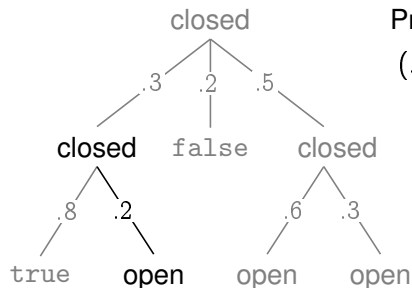
## Importance sampling with look-ahead



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



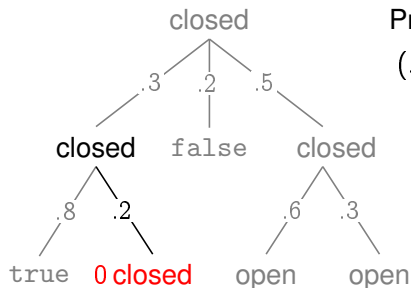
Probability mass  $p_c = .75$

(.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.



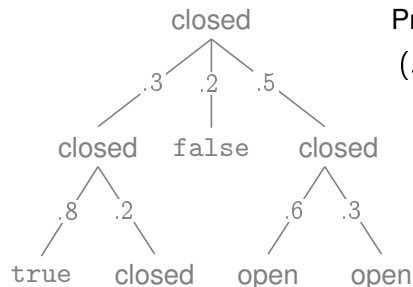
## Importance sampling with look-ahead



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

- Memoization

- Nested inference

## Implementation

- Reifying a model into a search tree

- Importance sampling with look-ahead

## ► Applications

## Conversational implicature in coordination discourse

**Alice:** some of our kids are coming home for dinner tonight.

**Bob:** (cooks food for  $n - 1$  kids)

## Conversational implicature in coordination discourse

Linguist: Does 'some' mean 'some but not all'?

Alice: **some** of our kids are coming home for dinner tonight.

Bob: (cooks food for  $n - 1$  kids)

# Conversational implicature in coordination discourse

Linguist: Does 'some' mean 'some but not all'?

Alice: **some** of our kids are coming home for dinner tonight.

Bob: (cooks food for  $n - 1$  kids)  
—process complex utterances less accurately

# Conversational implicature in coordination discourse

Linguist: Does 'some' mean 'some but not all'?

Alice: **some** of our kids are coming home for dinner tonight.  
—trade off informativity against complexity

Bob: (cooks food for  $n - 1$  kids)  
—process complex utterances less accurately

# Conversational implicature in coordination discourse

**Linguist:** Does 'some' mean 'some but not all'?  
—express nested probabilistic models intuitively

**Alice:** **some** of our kids are coming home for dinner tonight.  
—trade off informativity against complexity

**Bob:** (cooks food for  $n - 1$  kids)  
—process complex utterances less accurately



## The bounded-rational hearer's program

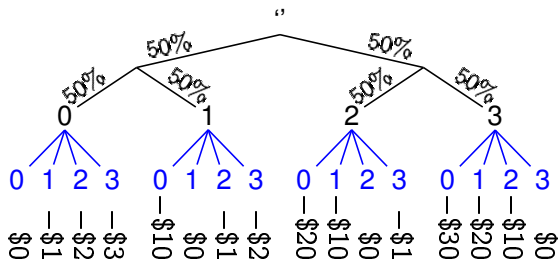
```
let count = (if flip 0.5 then 2 else 0) +
            (if flip 0.5 then 1 else 0) in
let conjunction = flip 0.5 in
if (not (some && not_all) || conjunction) &&
   (not some                || count > 0 ) &&
   (not not_all             || count < 3 )
then let action = ... in (action, utility action)
else fail ()
```

## The bounded-rational hearer's program

```
let count = (if flip 0.5 then 2 else 0) +
            (if flip 0.5 then 1 else 0) in
let conjunction = flip 0.5 in
if (not (some && not_all) || conjunction) &&
   (not some                || count > 0 ) &&
   (not not_all             || count < 3 )
then let action = ... in (action, utility action)
else fail ()
```

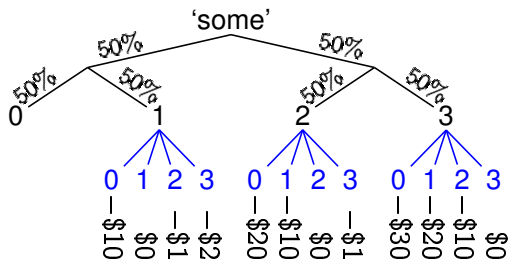
## The bounded-rational hearer's program

```
let count = (if flip 0.5 then 2 else 0) +  
            (if flip 0.5 then 1 else 0) in  
let conjunction = flip 0.5 in  
if (not (some && not_all) || conjunction) &&  
    (not some || count > 0 ) &&  
    (not not_all || count < 3 )  
then let action = ... in (action, utility action)  
else fail ()
```



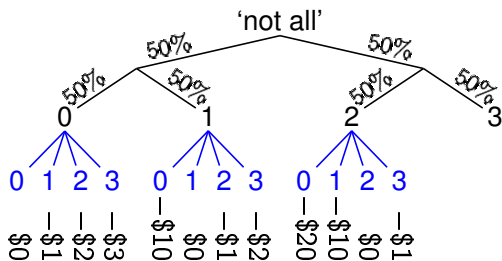
## The bounded-rational hearer's program

```
let count = (if flip 0.5 then 2 else 0) +  
            (if flip 0.5 then 1 else 0) in  
let conjunction = flip 0.5 in  
if (not (some && not_all) || conjunction) &&  
    (not some || count > 0 ) &&  
    (not not_all || count < 3 )  
then let action = ... in (action, utility action)  
else fail ()
```



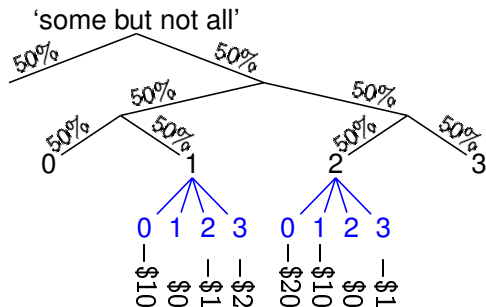
## The bounded-rational hearer's program

```
let count = (if flip 0.5 then 2 else 0) +  
            (if flip 0.5 then 1 else 0) in  
let conjunction = flip 0.5 in  
if (not (some && not_all) || conjunction) &&  
    (not some || count > 0 ) &&  
    (not not_all || count < 3 )  
then let action = ... in (action, utility action)  
else fail ()
```



## The bounded-rational hearer's program

```
let count = (if flip 0.5 then 2 else 0) +  
            (if flip 0.5 then 1 else 0) in  
let conjunction = flip 0.5 in  
if (not (some && not_all) || conjunction) &&  
    (not some || count > 0 ) &&  
    (not not_all || count < 3 )  
then let action = ... in (action, utility action)  
else fail ()
```



# 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 contains 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 A4-B4 and B4-C5 pairs, and a larger red bracket encompasses the A4-B4-C5-D5 sequence. Black brackets are drawn under the E5-F#5-G5-A5 sequence and a larger black bracket encompasses the entire sequence from G4 to A5. The bottom staff shows a simplified version of the motif, consisting of a quarter note G4, a quarter rest, a quarter note G4, 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.

# Motivic development in Beethoven sonatas

(Pfeffer 2007)

Source motif

The image displays two staves of musical notation in treble clef. The top staff, labeled "Source motif", contains a sequence of notes: G4, A4, B4, G4, A4, B4, C#5, B4, A4. The first five notes (G4-A4-B4-G4-A4) are grouped by black brackets, and the last four notes (B4-C#5-B4-A4) are grouped by red brackets. The bottom staff shows a development of this motif, starting with a single G4 note, followed by a four-note sequence: B4, C#5, B4, A4. The notes B4, C#5, B4, and A4 in the bottom staff are colored red, corresponding to the red brackets in the source motif above.

# Motivic development in Beethoven sonatas

(Pfeffer 2007)

Source motif

↑ infer

Destination motif

The diagram illustrates the process of inferencing from a source motif to a destination motif. The source motif is shown on a treble clef staff with the notes G4, A4, B4, C5, B4, A4, G4, F#4, G4. Brackets underneath group these notes into three segments: G4-A4-B4, C5-B4-A4, and B4-A4-G4-F#4-G4. The destination motif is shown on a lower treble clef staff with the notes G4, A4, B4, C5, B4, A4, G4. An upward-pointing arrow labeled 'infer' connects the destination motif to the source motif, indicating that the destination motif is derived from the source motif through a process of inferencing.

# Motivic development in Beethoven sonatas

(Pfeffer 2007)

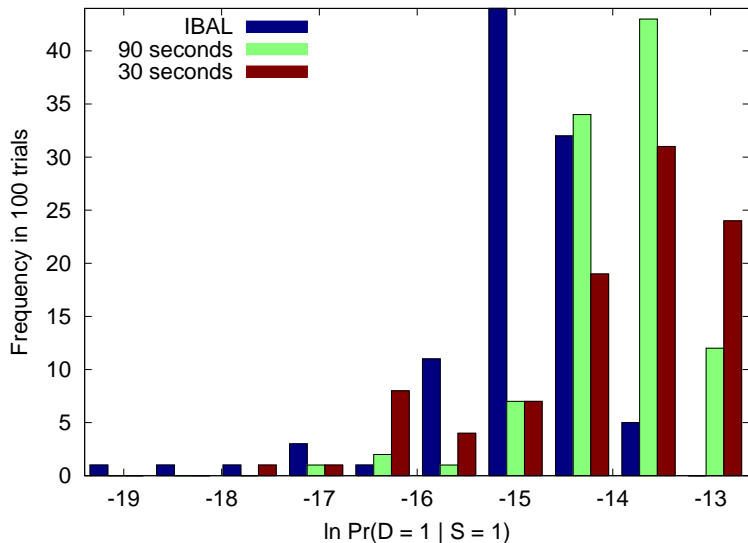
Source motif

infer

Destination motif

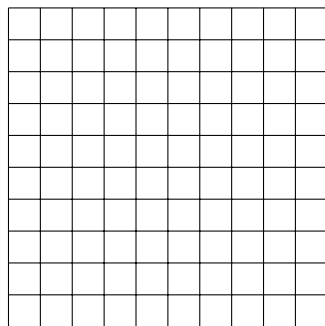
Implemented using lazy stochastic lists.

Motif pair	1	2	3	4	5	6	7
<b>% correct using importance sampling</b>							
● Pfeffer 2007 (30 sec)	93	100	28	80	98	100	63
● Us (90 sec)	98	100	29	87	94	100	77
● Us (30 sec)	92	99	25	46	72	95	61



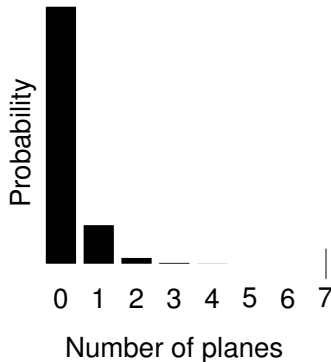
# Noisy radar blips for aircraft tracking

(Milch et al. 2007)

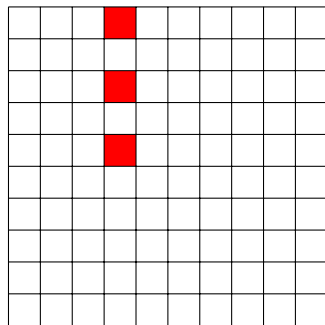


Blips present and absent

infer

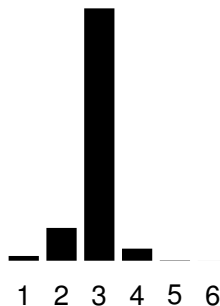


Particle filter. Implemented using lazy stochastic coordinates.



infer

Probability



Blips present and absent

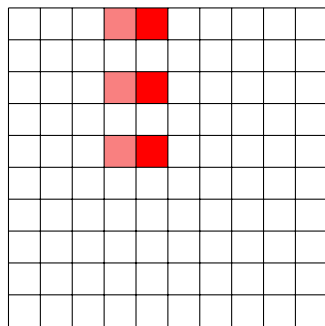
$t = 1$

Number of planes

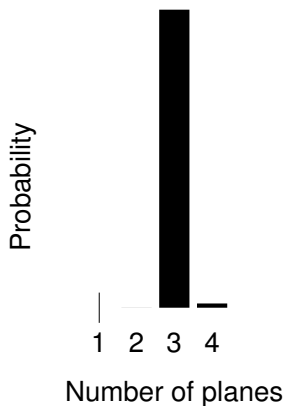
Particle filter. Implemented using lazy stochastic coordinates.

# Noisy radar blips for aircraft tracking

(Milch et al. 2007)



infer



Blips present and absent

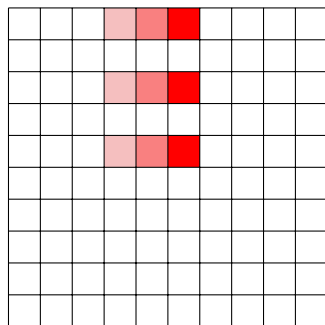
$t = 1, t = 2$

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.

# Summary

	Model (what)	Inference (how)
Toolkit	+ use existing libraries, types, debugger	+ easy to add custom inference
Language	+ random variables are ordinary variables	+ compile models for faster inference
<b>Today:</b>	<b>Payoff: expressive model</b>	<b>Payoff: fast inference</b>
<b>Best of both</b>	+ models <i>of inference</i> : bounded-rational theory of mind	+ 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.**

HANSEI <http://okmij.org/ftp/kakuritu/>