# Monolingual probabilistic programming using generalized coroutines

Oleg Kiselyov
FNMOC
oleg@pobox.com

Chung-chieh Shan
Rutgers University
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 (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 | invoke → | ← interpret |

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

# Declarative probabilistic inference

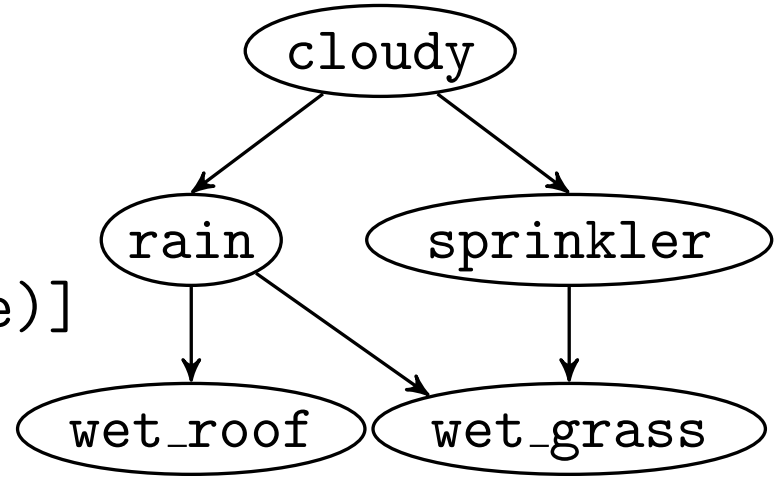
| | Model (what) | Inference (how) |
|---|---|---|
| Toolkit (BNT, PFP) | + use existing libraries, types, debugger | + easy to add custom inference |
| Language (BLOG, IBAL, Church) | + random variables are ordinary variables | + compile models for faster inference |
| Today: Best of both | Payoff: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

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

# Outline

# Grass model

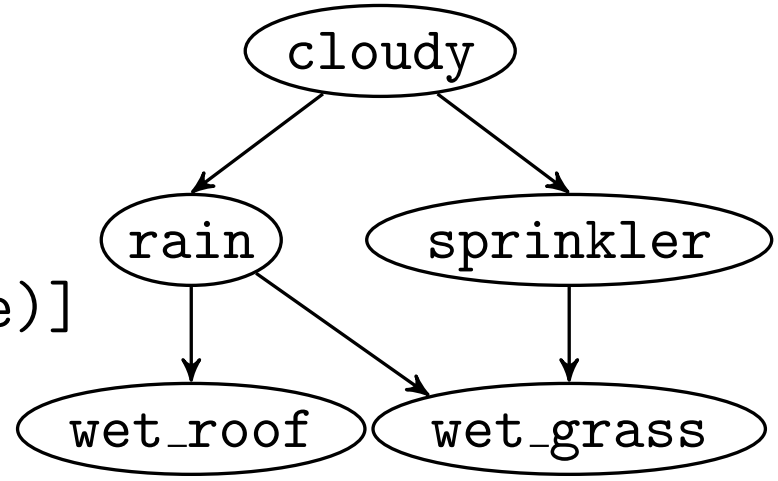
```
let flip = fun p ->
 dist [(p, true); (1.-.p, false)]
```




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

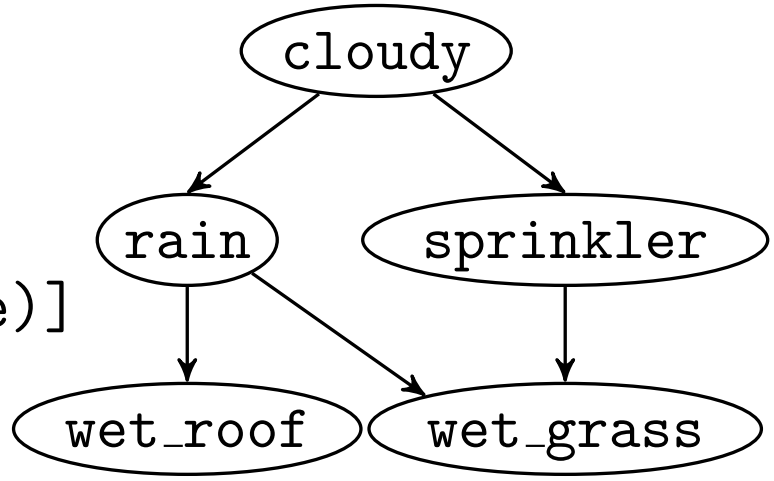
# Grass model

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




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

# Grass model

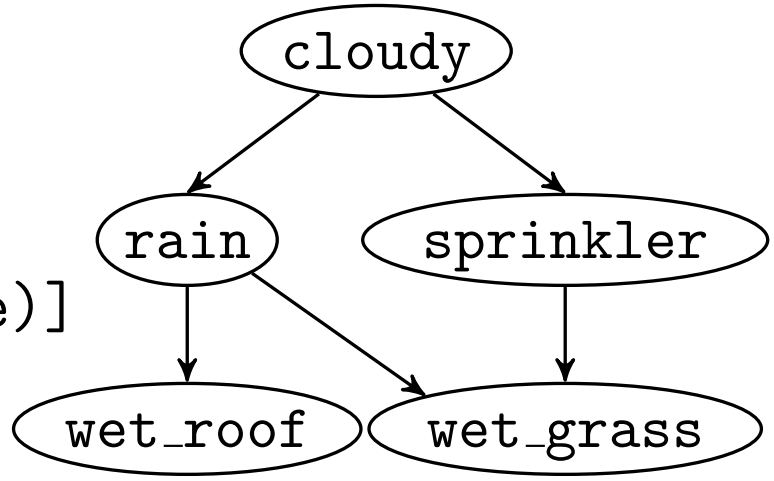



```
let flip = fun p ->
 dist [(p, true); (1.-.p, false)]


 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

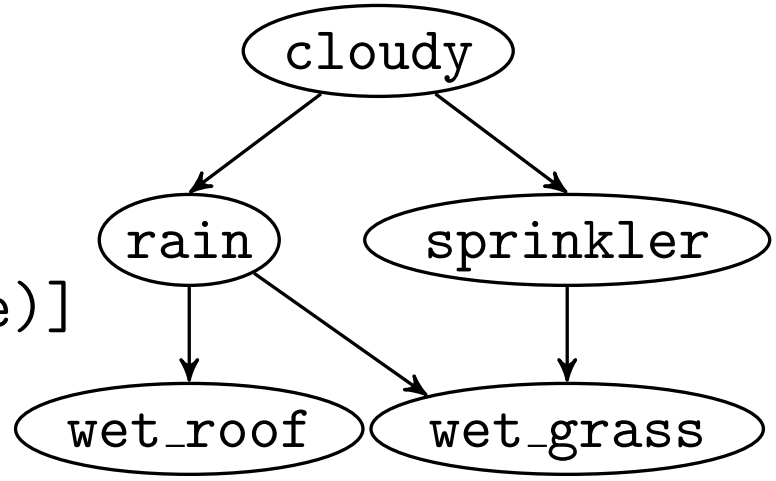



```
let flip = fun p ->
 dist [(p, true); (1.-.p, false)]

 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



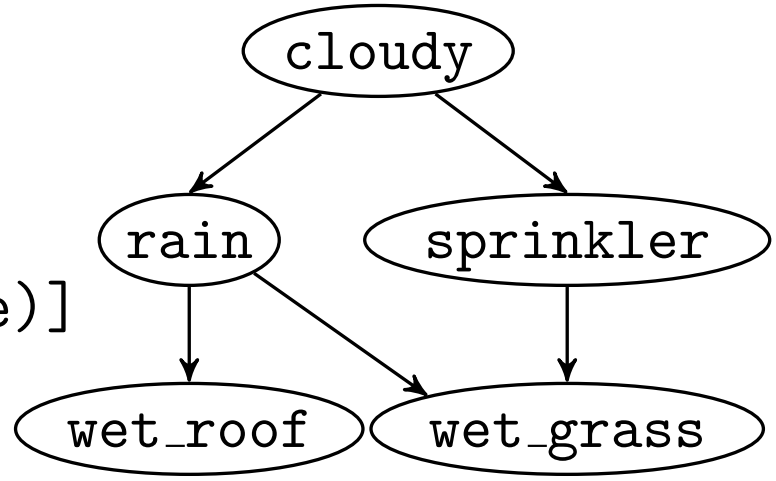

```
let flip = fun p ->
 dist [(p, true); (1.-.p, false)]

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

# 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)

# 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
```

# 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`?

# 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)
```

# 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)          )
```

# 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)                    )
```

# 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.

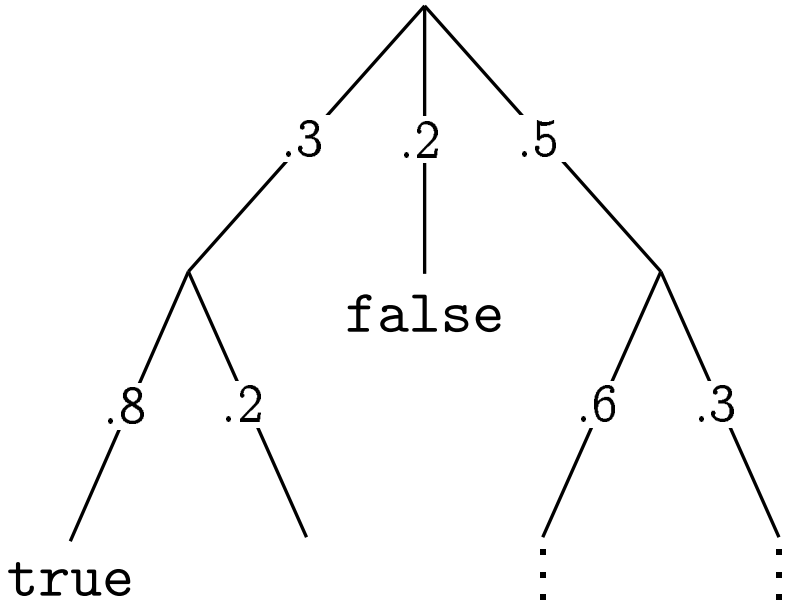
```
  at_least 0.3 true (sample 2 coin)                    )
```

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
  - Memoization
  - Nested inference
- **Implementation**
  - Reifying a model into a search tree
  - Importance sampling with look-ahead
- Performance

# Reifying a model into a search tree



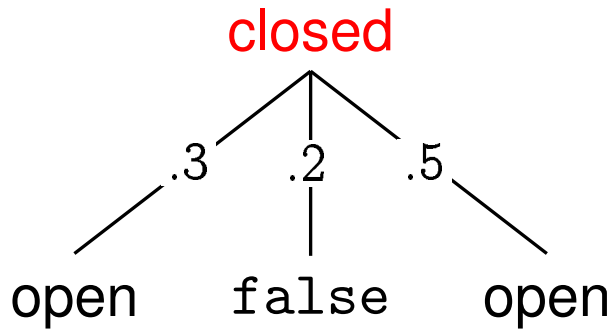

Exact inference by depth-first brute-force enumeration.
Rejection sampling by top-down random traversal.

# Reifying a model into a search tree

open

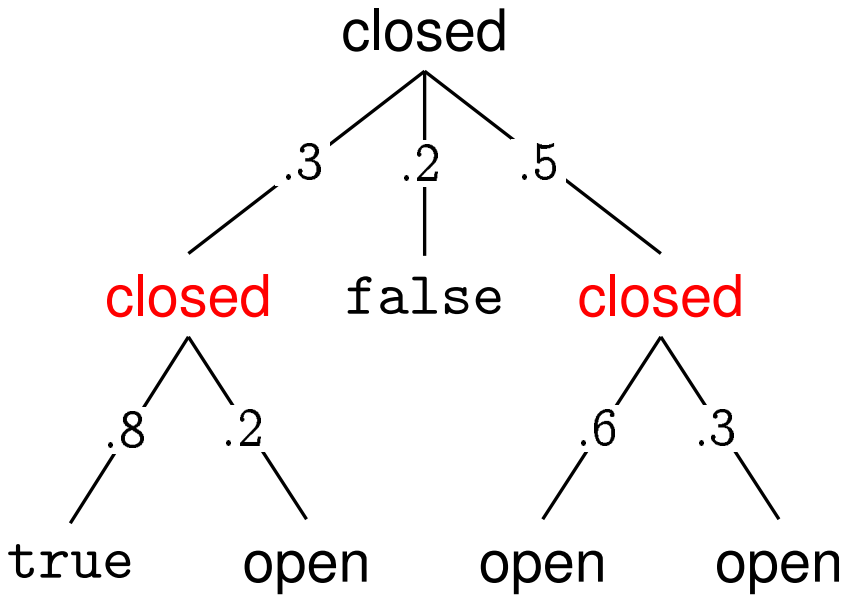
Exact inference by depth-first brute-force enumeration.
Rejection sampling by top-down random traversal.

# Reifying a model into a search tree



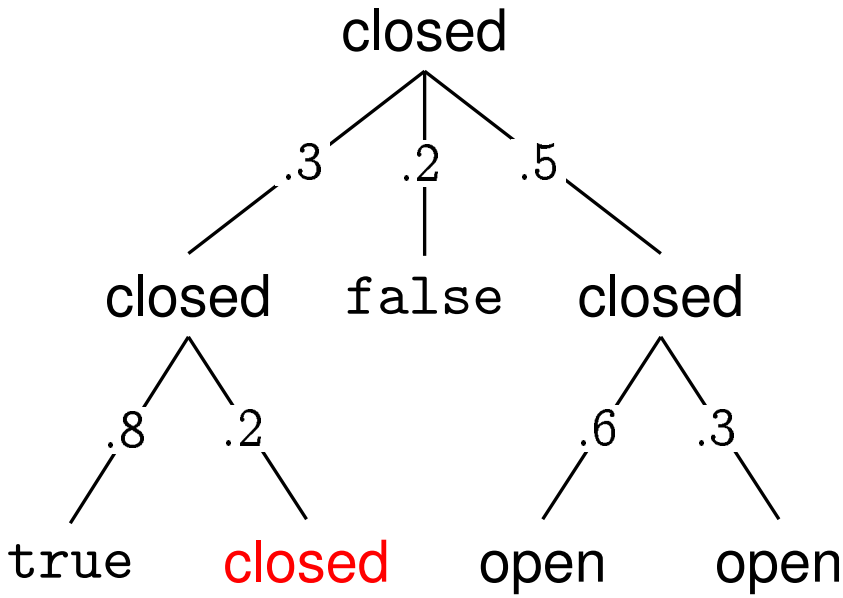

Exact inference by depth-first brute-force enumeration.
Rejection sampling by top-down random traversal.

# Reifying a model into a search tree



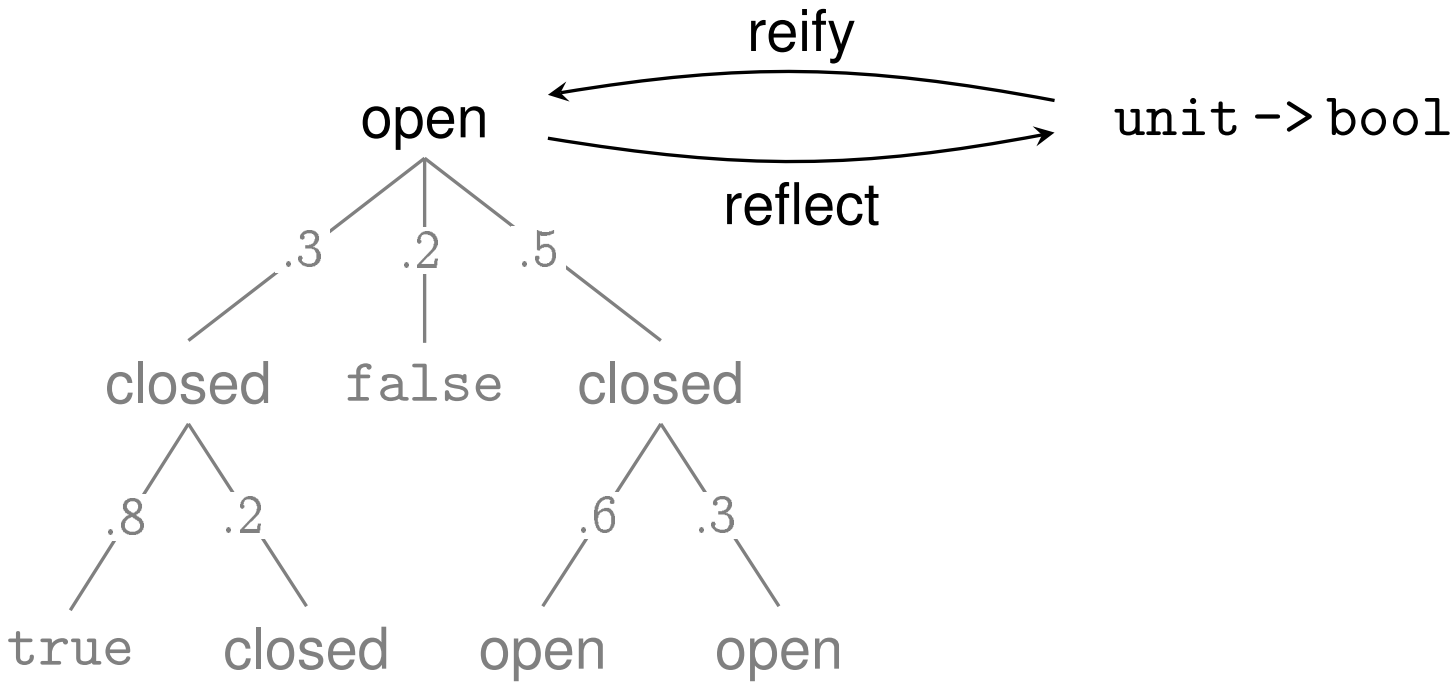

Exact inference by depth-first brute-force enumeration.
Rejection sampling by top-down random traversal.

# 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

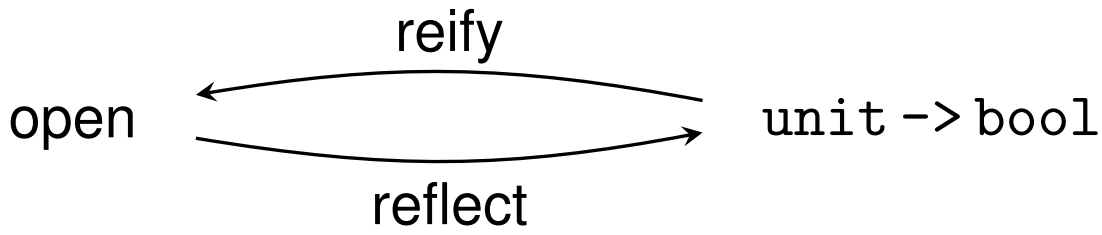



Inference procedures cannot access models' source code.

Reify then reflect:

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

# Reifying a model into a search tree




**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)

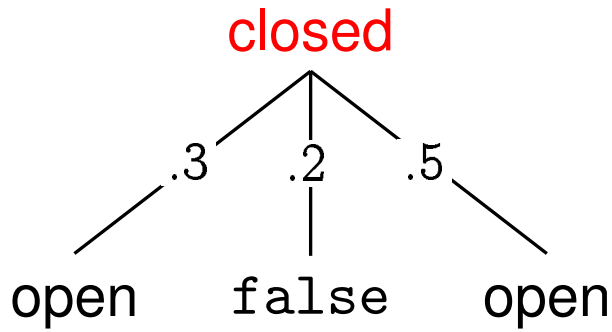
**Implementation:** using clonable user-level threads

- 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 $p_c = 1$

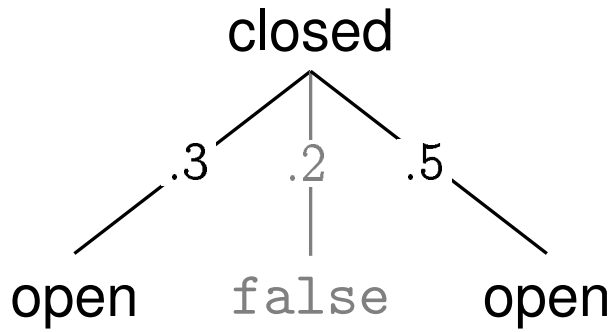
# Importance sampling with look-ahead




Probability mass $p_c = 1$

1. Expand one level.

# Importance sampling with look-ahead

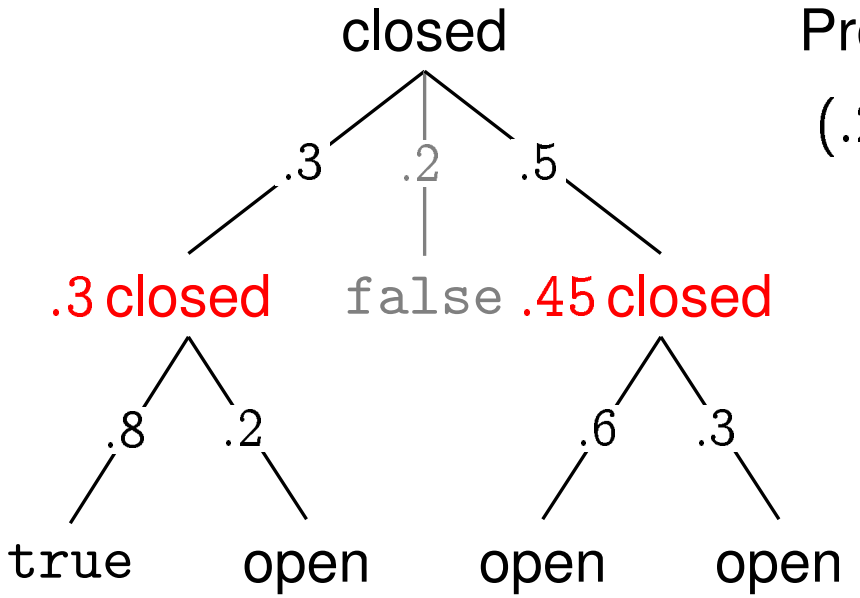



Probability mass $p_c = 1$

$(.2, \texttt{false})$

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

# Importance sampling with look-ahead

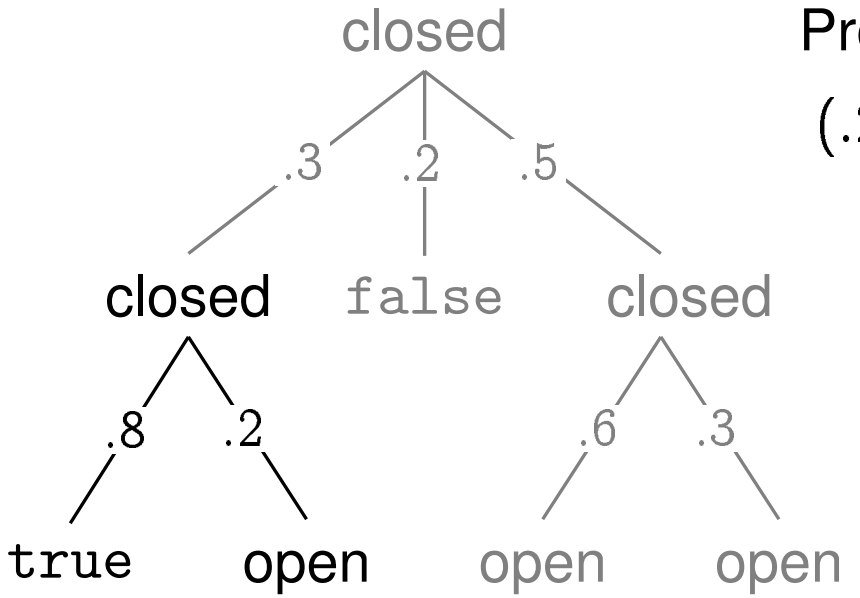



Probability mass $p_c = .75$

$(.2, \texttt{false})$

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

# Importance sampling with look-ahead

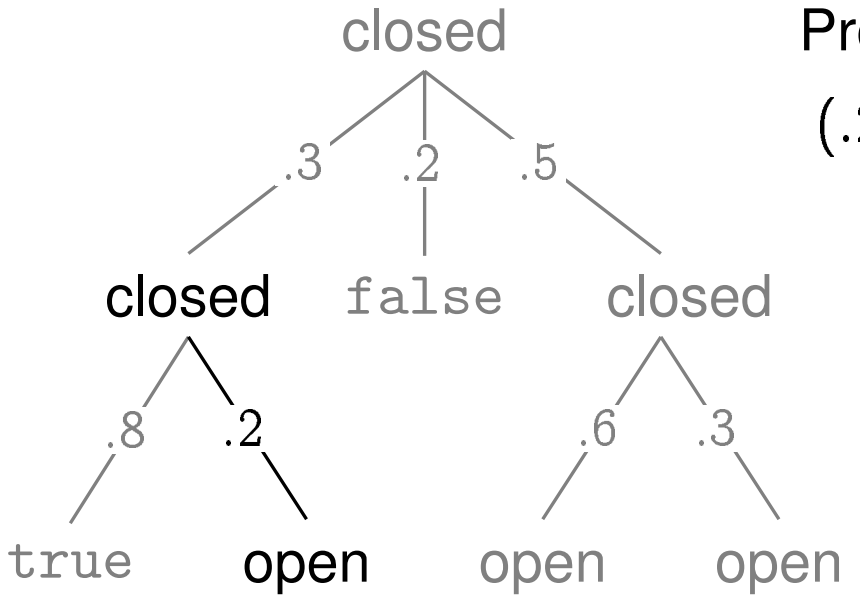



Probability mass $p_c = .75$

$(.2, \texttt{false})$

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

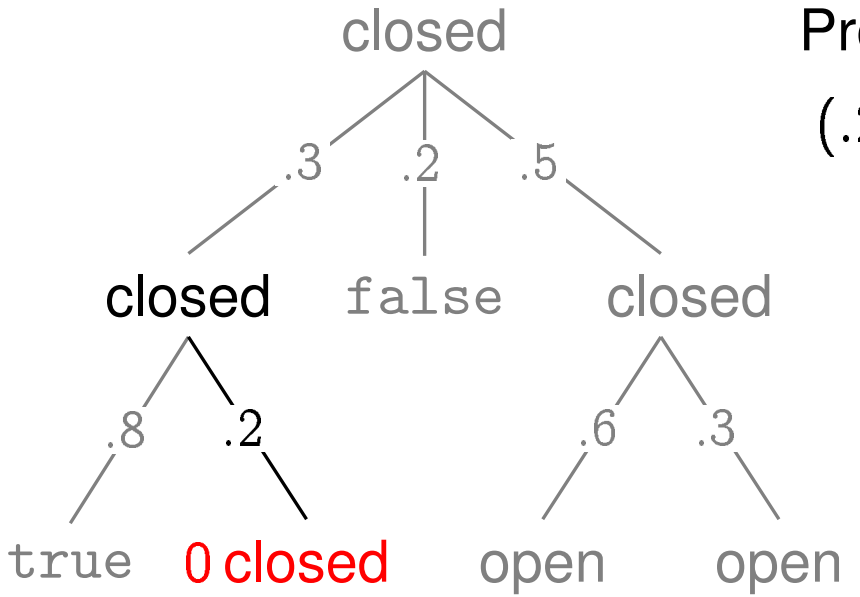



Probability mass $p_c = .75$

$(.2, \texttt{false})$ $(.6, \texttt{true})$

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

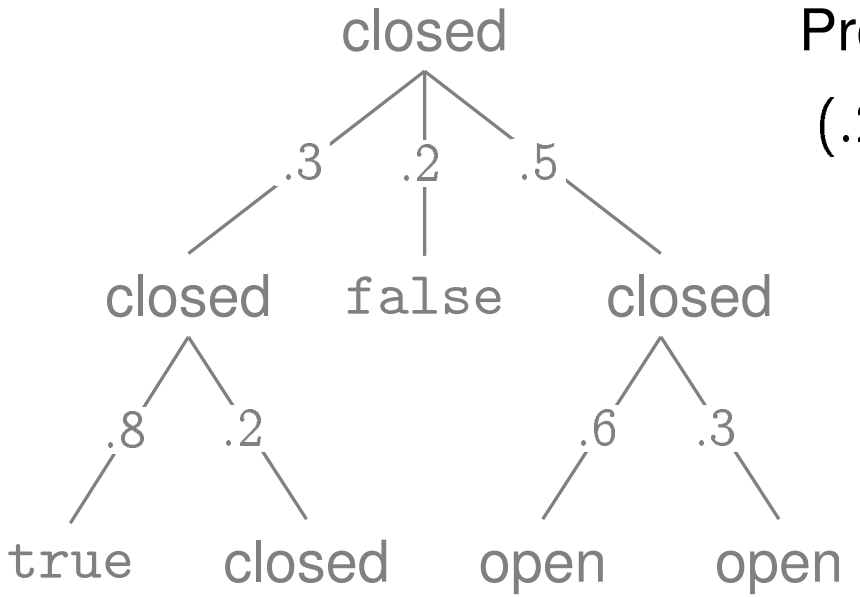



Probability mass $p_c = 0$

$(.2, \texttt{false})$ $(.6, \texttt{true})$

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 $p_c = 0$

$(.2, \texttt{false})$ $(.6, \texttt{true})$

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

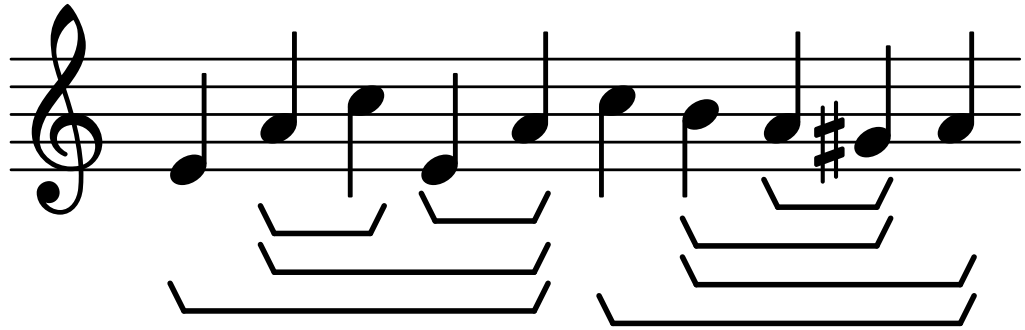
# Motivic development in Beethoven sonatas (Pfeffer 2007)

Source motif 

# Motivic development in Beethoven sonatas

(Pfeffer 2007)

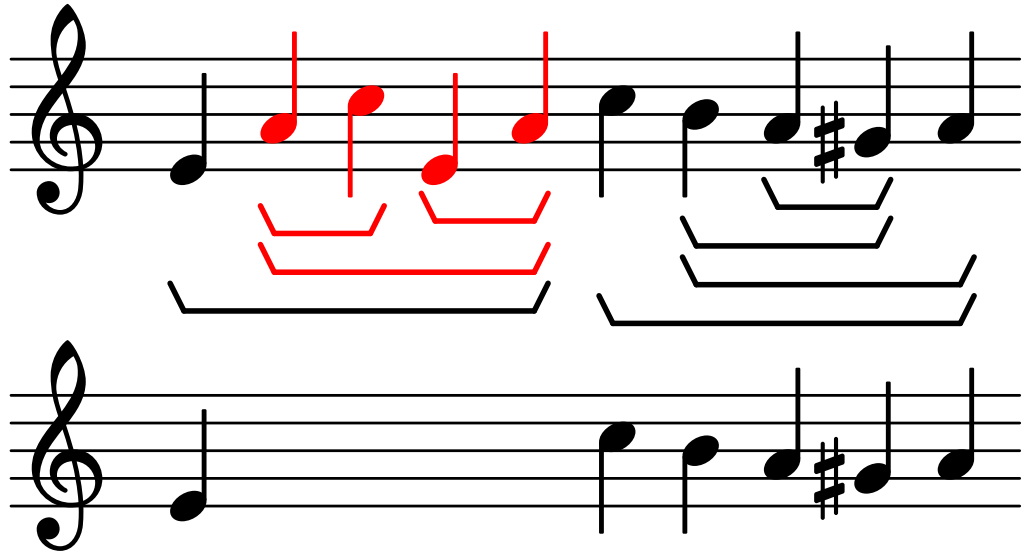
Source motif



# Motivic development in Beethoven sonatas

(Pfeffer 2007)

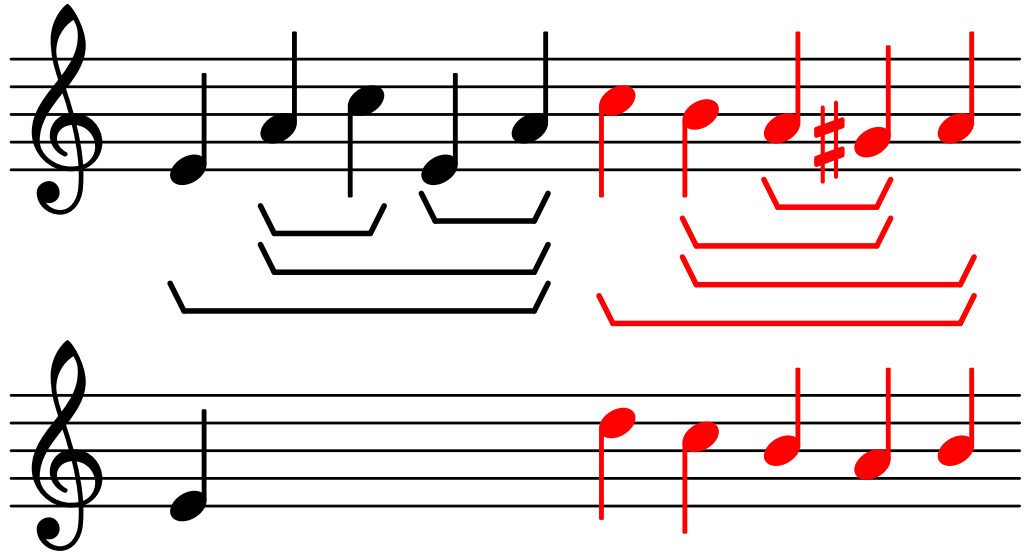
Source motif



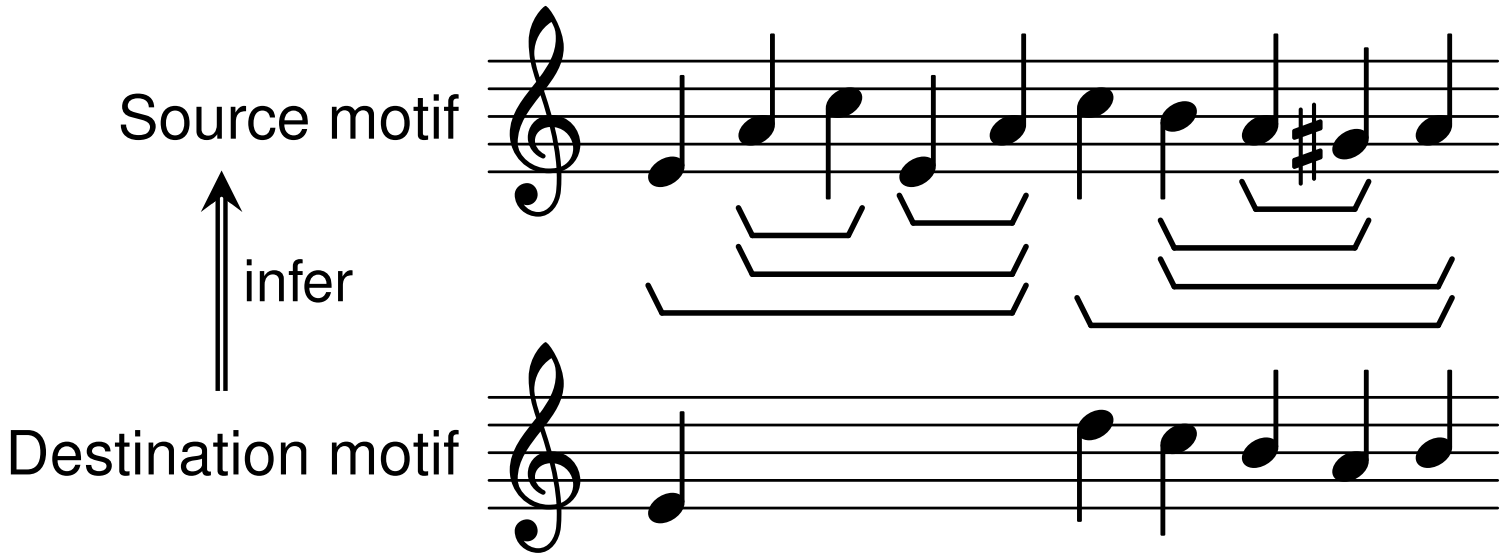
# Motivic development in Beethoven sonatas

(Pfeffer 2007)

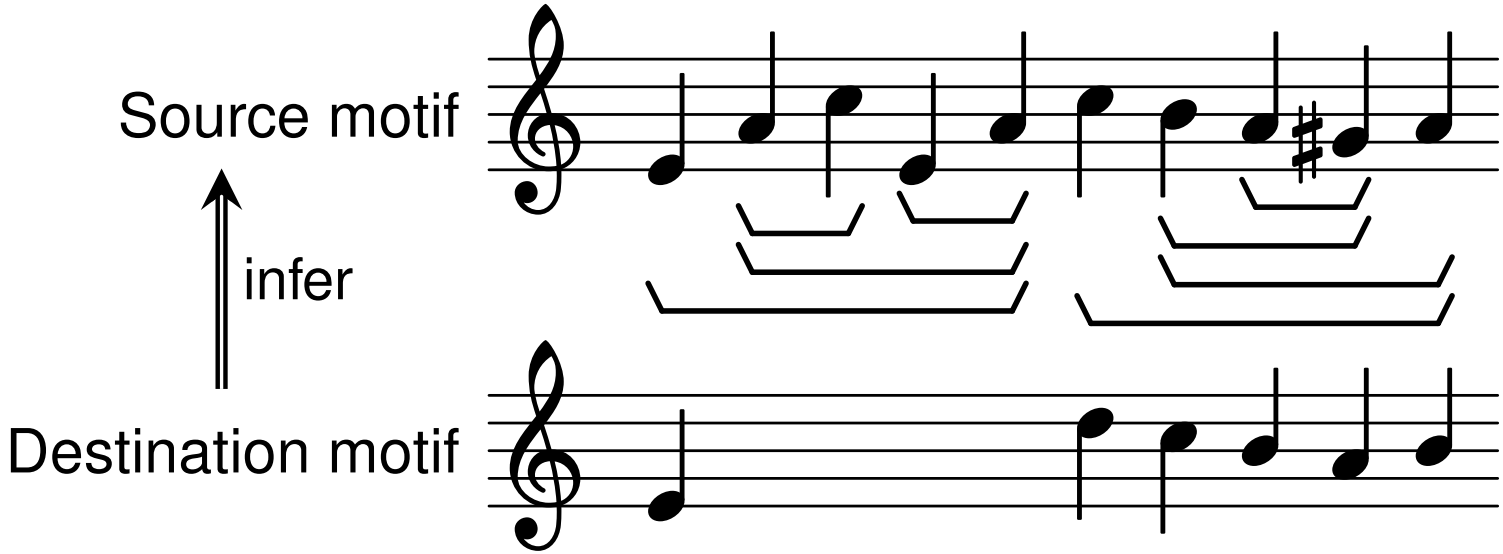
Source motif



# Motivic development in Beethoven sonatas (Pfeffer 2007)



# Motivic development in Beethoven sonatas (Pfeffer 2007)

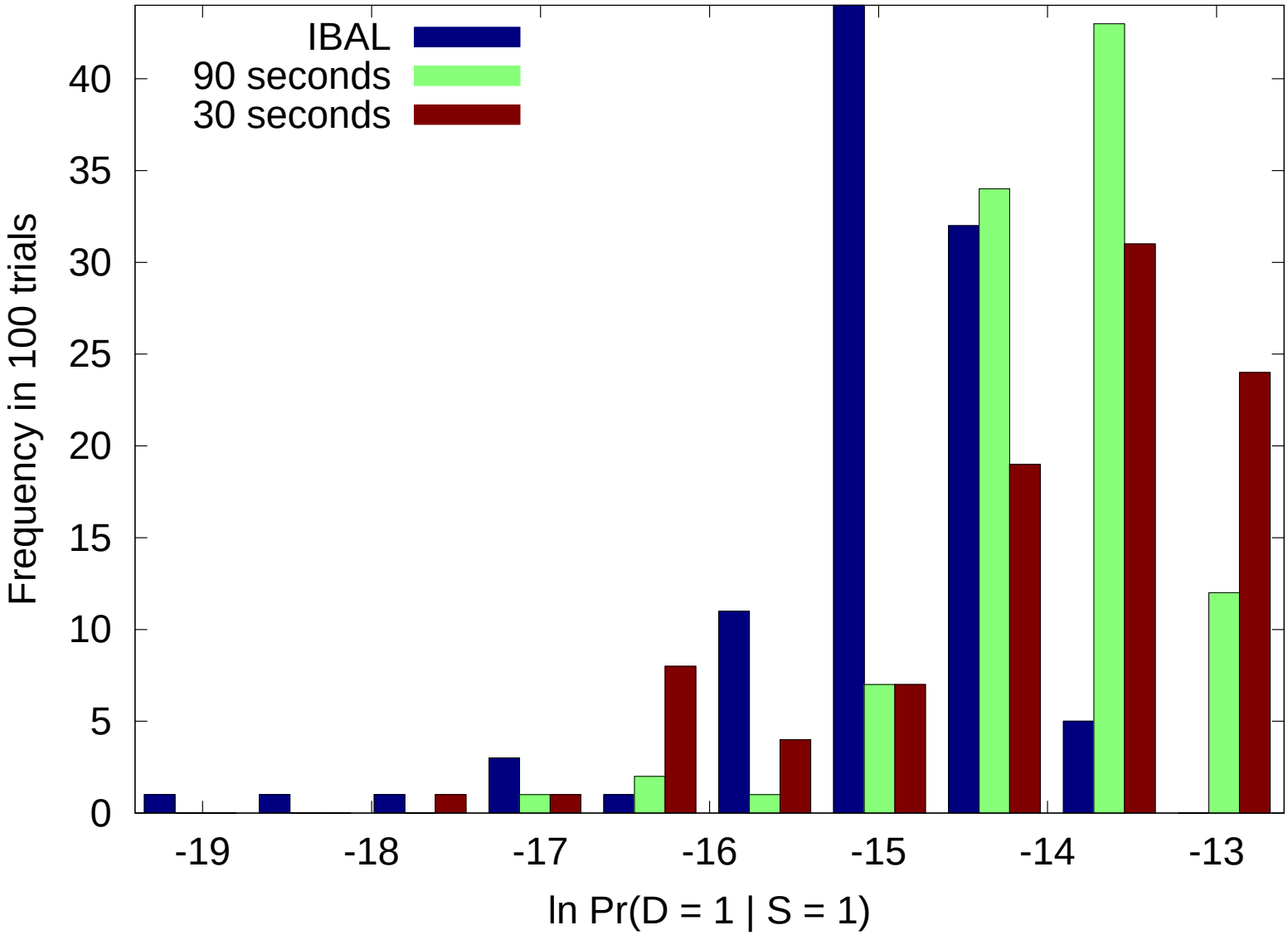



Implemented using lazy stochastic lists.

| 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 |

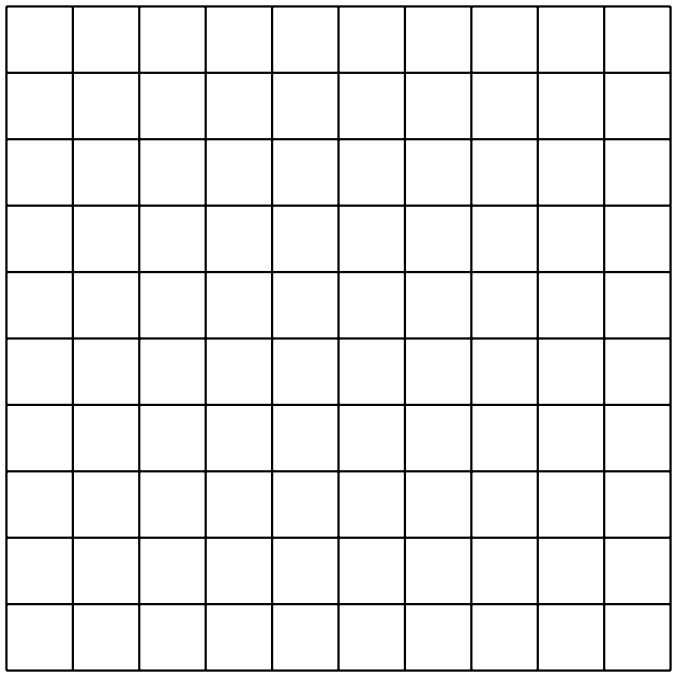
# Motivic development in Beethoven sonatas

(Pfeffer 2007)

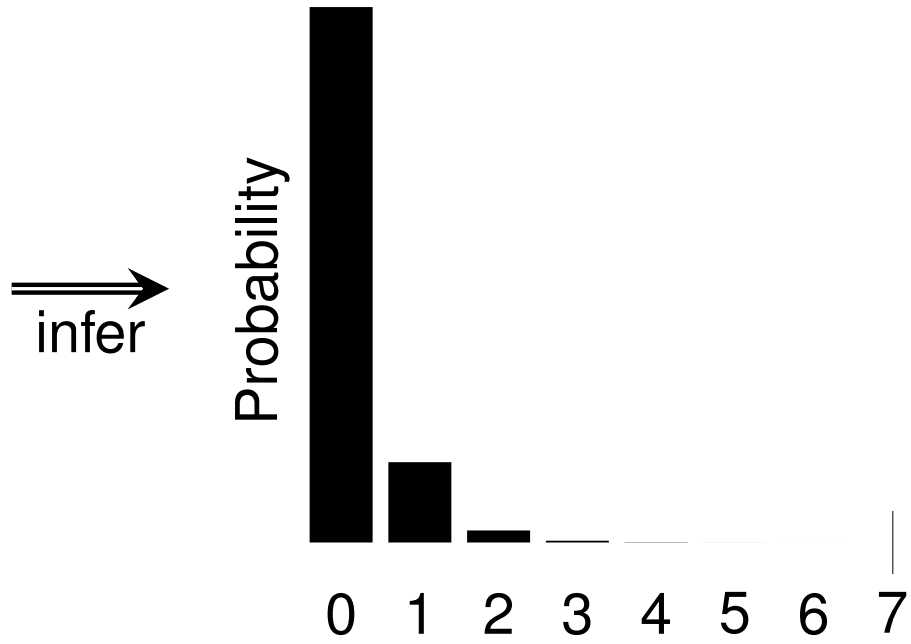


# Noisy radar blips for aircraft tracking

(Milch et al. 2007)



Blips present and absent

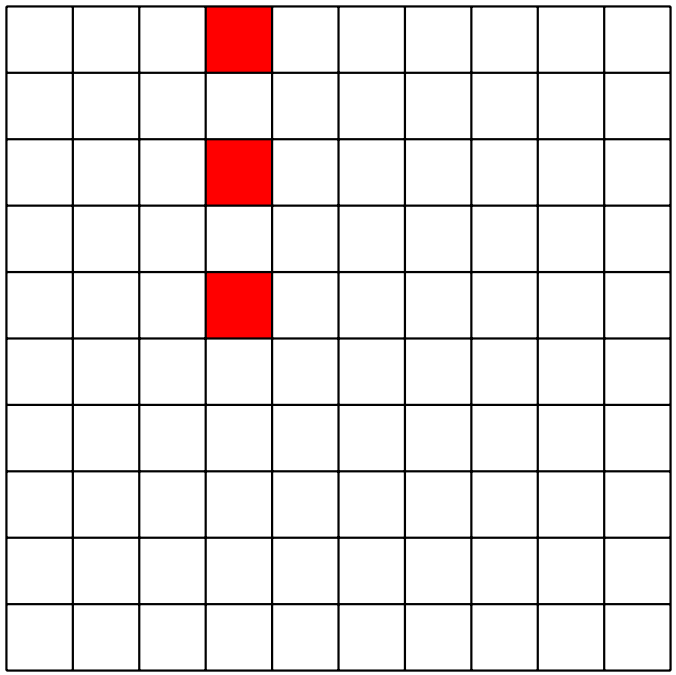



Number of planes

Particle filter. Implemented using lazy stochastic coordinates.

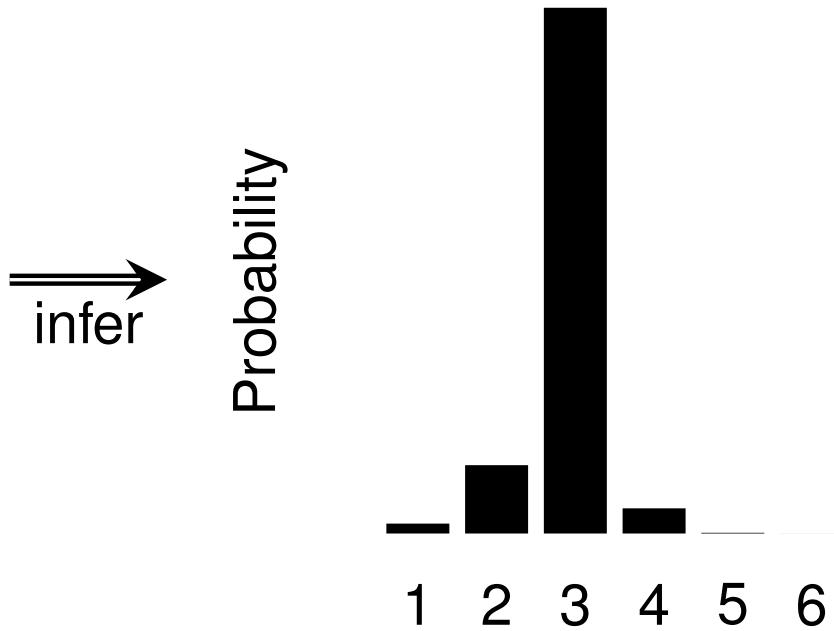
# Noisy radar blips for aircraft tracking

(Milch et al. 2007)



Blips present and absent

$t = 1$

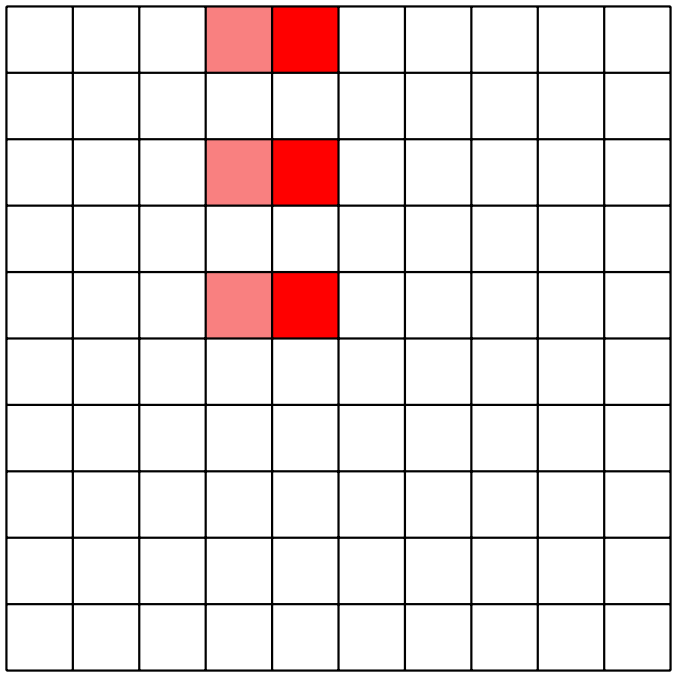



Number of planes

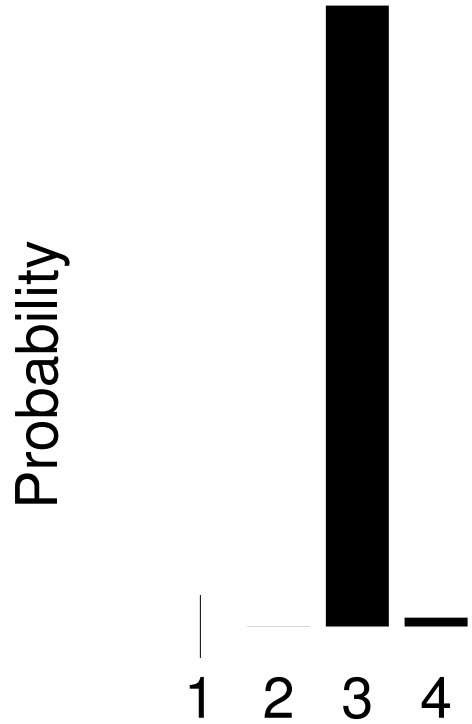
Particle filter. Implemented using lazy stochastic coordinates.

# Noisy radar blips for aircraft tracking

(Milch et al. 2007)



infer




Blips present and absent

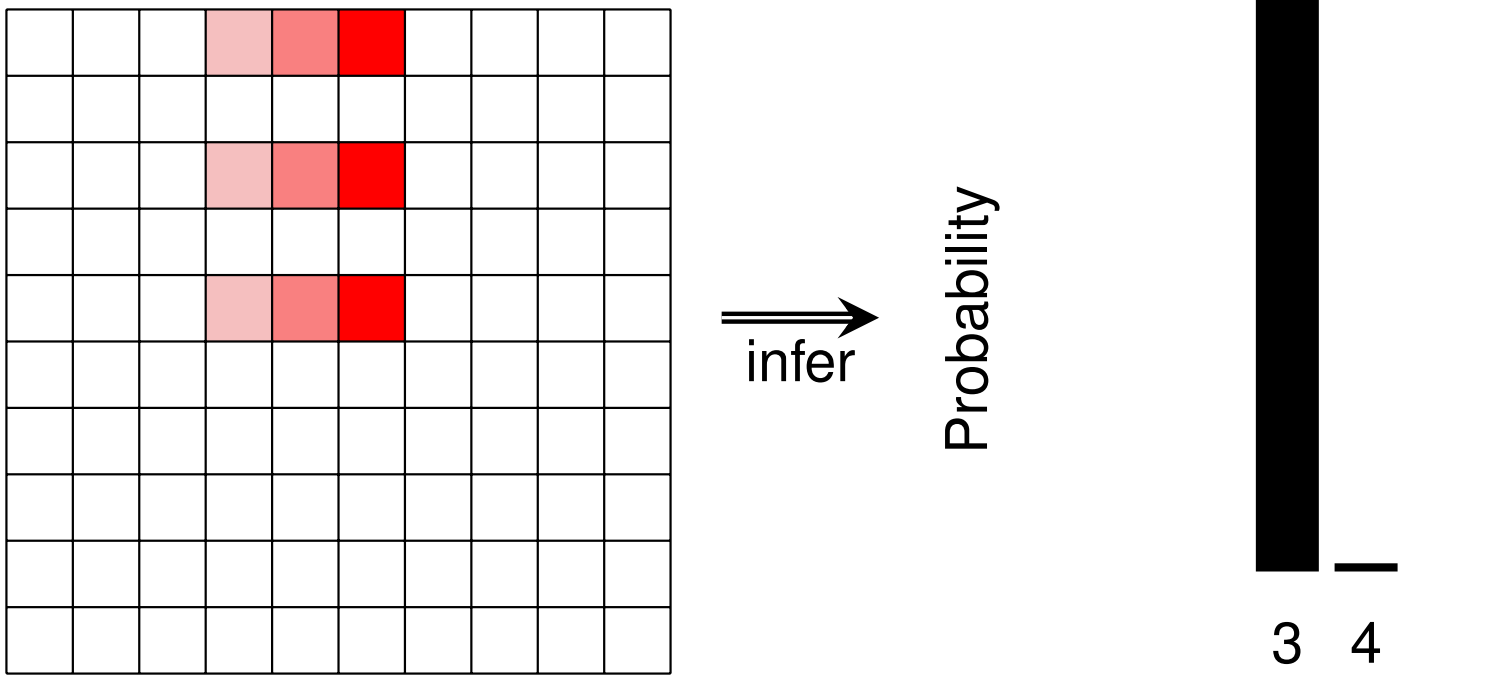
$t = 1, \quad t = 2$

Number of planes

Particle filter. Implemented using lazy stochastic coordinates.

# Noisy radar blips for aircraft tracking

(Milch et al. 2007)




Blips present and absent

$t = 1, \quad t = 2, \quad t = 3$

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 |
| Today: Best of both | Payoff: expressive model<br>+ models *of inference*: bounded-rational theory of mind | Payoff: fast inference<br>+ deterministic parts of models run *at full speed*<br>+ importance sampling |

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