

Probabilistic programming using first-class stores and first-class continuations

Oleg Kiselyov
FNMOC
oleg@pobox.com

Chung-chieh Shan
Rutgers University
ccshan@rutgers.edu

ML workshop
September 26, 2010

Probabilistic inference

I have exactly two kids.
At least one is a girl.

What is the probability that
my older kid is a girl?

Probabilistic inference

Model (what)

Inference (how)

$\Pr(\text{Reality})$

$\text{Reality} \rightarrow \text{Obs, Result}$
obs

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

I have exactly two kids.
At least one is a girl.

What is the probability that
my older kid is a girl?

Declarative probabilistic inference

Model (what)

Inference (how)

$\Pr(\text{Reality})$

$\text{Reality} \rightarrow \text{Obs, Result}$
obs

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

I have exactly two kids.
At least one is a girl.

What is the probability that
my older kid is a girl?

Declarative probabilistic inference

(UAI 2009, DSL 2009)

Model (what)

Inference (how)

$\Pr(\text{Reality})$

$\text{Reality} \rightarrow \text{Obs, Result}$
obs

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

I have exactly two kids.
At least one is a girl.

What is the probability that
my older kid is a girl?

Models and inference
as interacting programs
in the same general-
purpose language

Model (what)

Inference (how)

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

$\text{Reality} \rightarrow \text{Obs}, \text{Result}$
 obs

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

I have exactly two kids.
At least one is a girl.

What is the probability that
my older kid is a girl?

```
let flip = fun p ->  
  dist [(p, true);  
        (1.-.p, false)]  
in let girl1 = flip 0.5 in  
   let girl2 = flip 0.5 in  
   if girl1 || girl2  
   then girl1 else fail ()
```

Models and inference
as interacting programs
in the same general-
purpose language

Model (what)

Inference (how)

 $\text{Pr}(\text{Reality})$ Reality \rightarrow Obs, Result
obs} $\text{Pr}(\text{Result} \mid \text{Obs} = \text{obs})$ I have exactly two kids.
At least one is a girl.What is the probability that
my older kid is a girl?`normalize (exact_reify (fun () ->`

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

```
in let girl1 = flip 0.5 in
  let girl2 = flip 0.5 in
  if girl1 || girl2
  then girl1 else fail ()))
```

true	1/2
false	1/4

Models and **inference**
as interacting programs
in the same general-
purpose language

Model (what)

Inference (how)

 $\Pr(\text{Reality})$ Reality \rightarrow Obs, Result
obs} $\Pr(\text{Result} \mid \text{Obs} = \text{obs})$ I have exactly two kids.
At least one is a girl.What is the probability that
my older kid is a girl?`normalize (exact_reify (fun () ->`

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

true	2/3
false	1/3

true	1/2
false	1/4

```
in let girl1 = flip 0.5 in
  let girl2 = flip 0.5 in
  if girl1 || girl2
  then girl1 else fail ()))
```

Models and **inference**
as interacting programs
in the same general-
purpose language

Model (what)

Inference (how)

 $\text{Pr}(\text{Reality})$ $\text{Reality} \rightarrow \text{Obs}, \text{Result}$
 obs $\left. \begin{array}{l} \text{Pr}(\text{Reality}) \\ \text{Reality} \rightarrow \text{Obs}, \text{Result} \\ \text{obs} \end{array} \right\} \text{Pr}(\text{Result} \mid \text{Obs} = \text{obs})$ I have exactly two kids.
At least one is a girl.What is the probability that
my older kid is a girl?

```

let flip = fun p ->
  dist [(p, true);
        (1.-.p, false)]
in let girl1 = flip 0.5 in
   let girl2 = flip 0.5 in
     if girl1 || girl2
     then girl1 else fail ()))

```

normalize (exact_reify (fun () ->

Expressive models
and efficient inference
as interacting programs
in the same general-
purpose language

Outline

▶ **Expressive models**

- Reuse existing infrastructure

- Nested inference

Efficient inference

- First-class continuations

- First-class stores

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. The notes are grouped into two main sections by large brackets below the staff. The first section contains the first four notes (G, A, B, G), and the second section contains the last five notes (A, B, C#, B, A). Within each section, smaller brackets group individual notes or pairs of notes, illustrating the hierarchical structure of the motif.

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, 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 and F#5-G5-A5 pairs, and a larger black bracket encompasses the E5-F#5-G5-A5 sequence. 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 six notes (G4-A4-B4-G4-A4-B4) are grouped by black brackets, and the last three 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 red vertical line, and then a sequence of red notes: B4, C#5, B4, A4. This sequence of red notes is bracketed in red, indicating its derivation from the source motif.

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 below the staff group these notes into three pairs: (G4, A4), (B4, C5), and (B4, A4). The destination motif is shown on a treble clef staff with the notes G4, B4, C5, B4, A4. 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

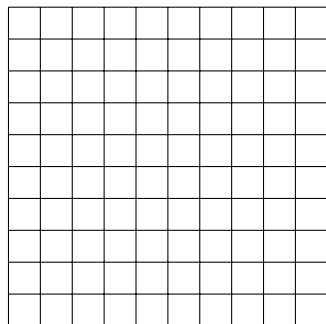
Destination motif

Motif pair	1	2	3	4	5	6	7
% correct							
Pfeffer 2007 (30 sec)	93	100	28	80	98	100	63
HANSEI (90 sec)	98	100	29	87	94	100	77
HANSEI (30 sec)	92	99	25	46	72	95	61

Importance sampling using lazy stochastic lists.

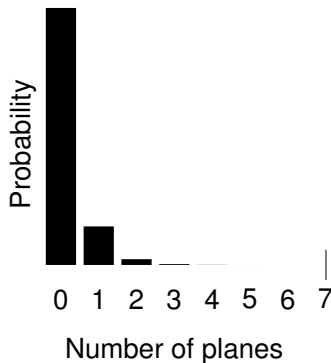
Noisy radar blips for aircraft tracking

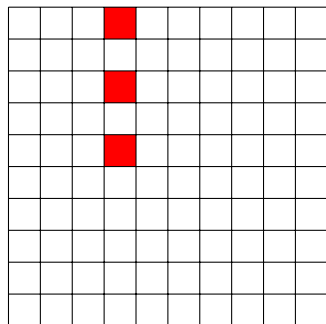
(Milch et al. 2007)



Blips present and absent

infer



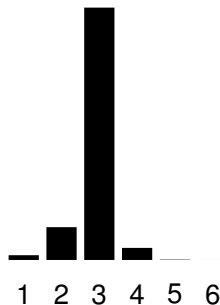


Blips present and absent

$t = 1$

infer

Probability

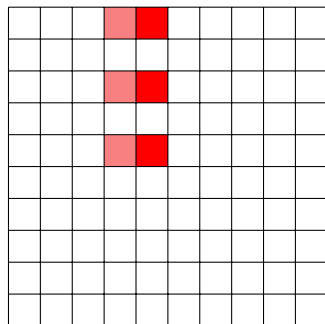


Number of planes

Particle filter using lazy stochastic coordinates.

Noisy radar blips for aircraft tracking

(Milch et al. 2007)



Blips present and absent

$t = 1, t = 2$

infer

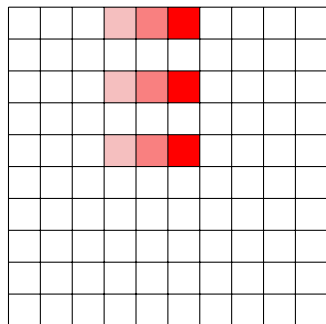
Probability



Particle filter using lazy stochastic coordinates.

Noisy radar blips for aircraft tracking

(Milch et al. 2007)



Blips present and absent

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

infer

Probability



Number of planes

Particle filter using lazy stochastic coordinates.

Models as programs in a general-purpose language

Reuse existing infrastructure!

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

Implemented independently in Haskell, Scheme, Ruby, Scala ...

Models that invoke 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
```

Models that invoke 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?

Models that invoke 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)
```


Models that invoke 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) )
```

Models that invoke 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) )
```

Models that invoke 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, using `dist` like models do.

Works with observation, recursion, memoization.

Metareasoning **without interpretive overhead**.

Outline

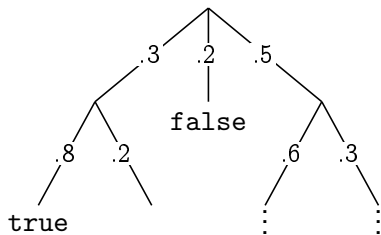
Expressive models

- Reuse existing infrastructure
- Nested inference

► **Efficient inference**

- First-class continuations
- First-class stores

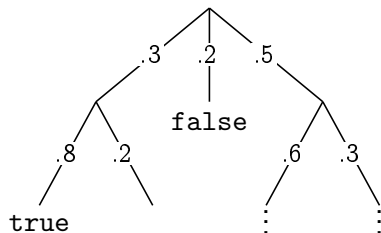
Reifying a model into a lazy search tree



not syntax tree
not call tree

```
type 'a branch = V of 'a | C of (unit -> 'a tree)
and 'a tree    = (prob * 'a branch) list
```

Reifying a model into a lazy search tree



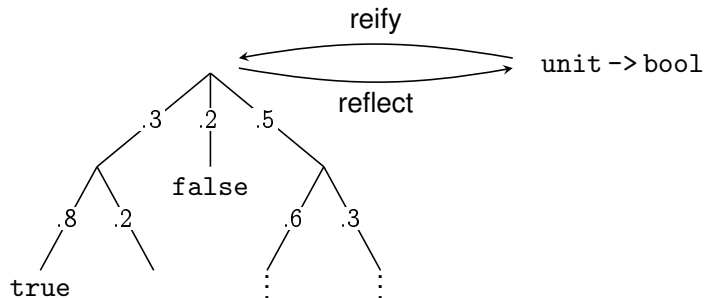
not syntax tree
not call tree

Depth-first enumeration = exact inference

Random dive = rejection sampling

Dive with look-ahead = importance sampling

Reifying a model into a lazy search tree

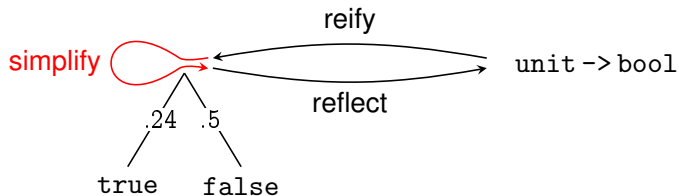


Represent a probability and state monad (Filinski 1994)
using first-class delimited continuations, aka clonable threads:

- ▶ Model runs inside a thread.
- ▶ `dist` clones the thread.
- ▶ `fail` kills the thread.

Models' code stays opaque. Deterministic parts run at full speed.
Nesting works.

Reifying a model into a lazy search tree



reflect ◦ **simplify** ◦ reify = table, chart, bucket

reflect ◦ **sample** ◦ reify = particle filter

The library so far

```
type 'a branch = V of 'a | C of (unit -> 'a tree)
and 'a tree    = (prob * 'a branch) list
```

```
let reify m = reset (fun () -> [(1.0, V (m ()))])
```

```
let dist ch = shift (fun k ->
  List.map (fun (p,v) -> (p, C (fun () -> k v))) ch)
```

The library so far

```
type 'a branch = V of 'a | C of (unit -> 'a tree)
and 'a tree    = (prob * 'a branch) list
```

```
let prompt = new_prompt ()
```

```
let reify m = reset prompt (fun () -> [(1.0, V (m ()))])
```

```
let dist ch = shift prompt (fun k ->
  List.map (fun (p,v) -> (p, C (fun () -> k v))) ch)
```

First-class continuations

```
type req = Done | Choice of (prob * (unit -> req)) list
```

```
let reify m =  
  let answer = ref None in  
  let rec interp req = match req with  
    | Done ->  
      let Some v = !answer in [(1.0, V v)]  
    | Choice ch ->  
      List.map (fun (p,m) ->  
                (p, C (fun () -> interp (m ())))))  
                ch  
  in interp (reset prompt (fun () ->  
                    answer := Some (m ()); Done))  
  
let dist ch = shift prompt (fun k ->  
  Choice (List.map (fun (p,v) -> (p, fun () -> k v)) ch))
```

Memoization

```
type gender = Female | Male

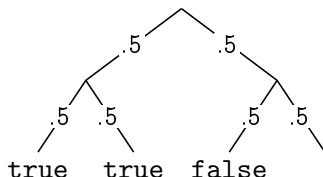
let kid = memo (fun n -> dist [(0.5, Female);
                              (0.5, Male)])
in if kid 1 = Female || kid 2 = Female
   then kid 1 else fail ()
```

Memoization

```
type gender = Female | Male

let kid = memo (fun n -> dist [(0.5, Female);
                              (0.5, Male)])
in if kid 1 = Female || kid 2 = Female
   then kid 1 else fail ()
```

Used to speed up inference (ICFP 2009)



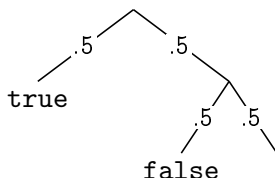
by delaying choices until observed

Memoization

```
type gender = Female | Male

let kid = memo (fun n -> dist [(0.5, Female);
                              (0.5, Male)])
in if kid 1 = Female || kid 2 = Female
   then kid 1 else fail ()
```

Used to speed up inference (ICFP 2009)



by delaying choices until observed

Memoization

```
type gender = Female | Male

let kid = memo (fun n -> dist [(0.5, Female);
                              (0.5, Male)])
in if kid 1 = Female || kid 2 = Female
   then kid 1 else fail ()
```

Used to speed up inference (ICFP 2009)
and to express nonparametric distributions (Goodman et al. 2008)

Lazy evaluation is `memo (fun () -> ...)`

Each search-tree node must keep its own store ('thread-local')
Nesting creates *regions* of memo cells (ICFP 2006)

Memoization

```
type gender = Female | Male

let kid = memo (fun n -> dist [(0.5, Female);
                               (0.5, Male)])

in if kid 1 = Female || kid 2 = Female
   then kid 1 else fail ()
```

Used to speed up inference (ICFP 2009)
and to express nonparametric distributions (Goodman et al. 2008)

Lazy evaluation is m

Each search-tree no

Nesting creates *reg*

Delimited Dynamic Binding

Oleg Kiselyov
FNMOC
oleg@pobox.com

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

Amr Sabry
Indiana University
sabry@indiana.edu

Abstract

Dynamic binding and *delimited* control are useful together in many settings, including Web applications, database cursors, and mobile code. We examine this pair of language features to show that the semantics of their interaction is ill-defined yet not expressive

to any function, dynamic variables let us pass additional data into a function and its callees without bloating its interface. This mechanism especially helps to modularise and separate concerns when applied to parameters such as line width, output port, character encoding, and error handler. Moreover, a dynamic variable lets us not just provide but also change the environment in which a piece of

First-class stores: interface

```
module Memory = struct
  type 'a loc
  type t
  val newm : t
  val new_loc : unit -> 'a loc
  val mref : 'a loc -> t -> 'a (* throws Not_found *)
  val mset : 'a loc -> 'a -> t -> t
end
```

First-class stores: usage

```
let reify m =
  let answer = ref None in
  let rec interp req = match req with
    | Done ->
      let Some v = !answer in [(1.0, V v)]
    | Choice ch ->
      List.map (fun (p,m) ->
        (p, C (fun () -> interp (m ())))))
        ch
  in
  let mem = !thread_local in
  thread_local := Memory.newm;
  let req = reset prompt (fun () ->
    answer := Some (m ()); Done) in
  thread_local := mem;
  interp req
```

Recap

Expressive models and **efficient inference**
as interacting programs
in the same general-purpose language

We want first-class delimited continuations and
(garbage-collector support for) first-class stores

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

Recap

Expressive models and efficient inference
as interacting programs
in the same general-purpose language

We want **first-class delimited continuations** and
(garbage-collector support for) **first-class stores**

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