Optimizing PPL model evaluation with graphs

and applicative functors

Viktor Palmkvist

Museum of Natural History

Inference in Probabilistic Programming

- Core idea: probabilistic model = probabilistic program
- Inference runs and modifies programs

Markov Chain Monte Carlo (MCMC)

1. Produce a sample (s_0).
2. Produce another sample (s').
3. Compare the two likelihoods.
    1. If proposal is likelier, accept (i.e., $s_1 := s'$).
    2. Otherwise, accept with probability $\frac{L(s')}{L(s_0)}$ (reject means $s_1 := s_0$).

Observation: most models are "smooth" ⇒ "close" samples have similar likelihood.

Problem: how do we efficiently produce a proposal that is "close" to the previous sample?

Defining "Close"

- General, must handle *all* programs. (Definition: "close" means one `assume` changed)

Example:

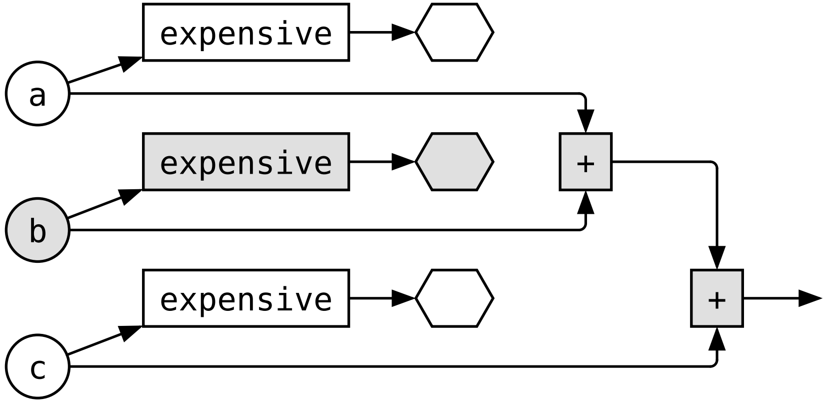
tppl

```
model function example() => Real {
  assume a ~ Gaussian(0.0, 1.0);
  observe expensive(a) ~ Gaussian(0.0, 1.0);
  assume b ~ Gaussian(0.0, 1.0);
  observe expensive(b) ~ Gaussian(0.0, 1.0);
  assume c ~ Gaussian(0.0, 1.0);
  observe expensive(c) ~ Gaussian(0.0, 1.0);
  return a + b + c;
}
```

Lines 2–3: Unchanged

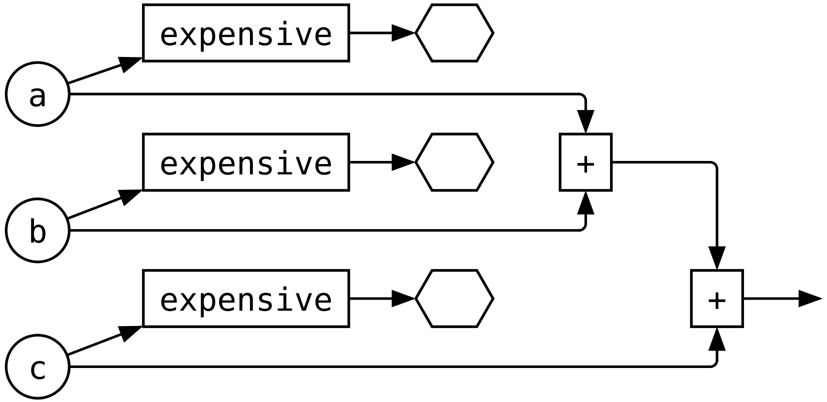
Lines 6–7: Unchanged

Data Dependencies



It's an applicative functor!

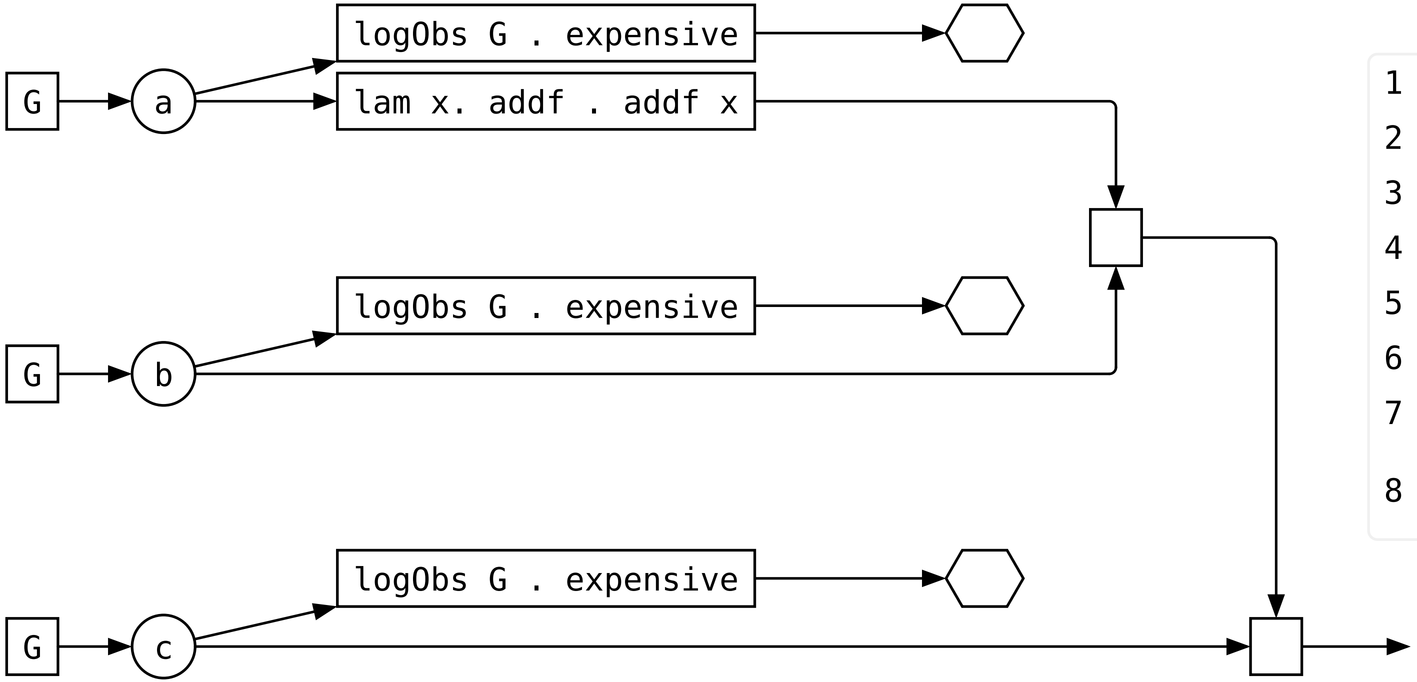
A Probabilistic Applicative Functor



mcore

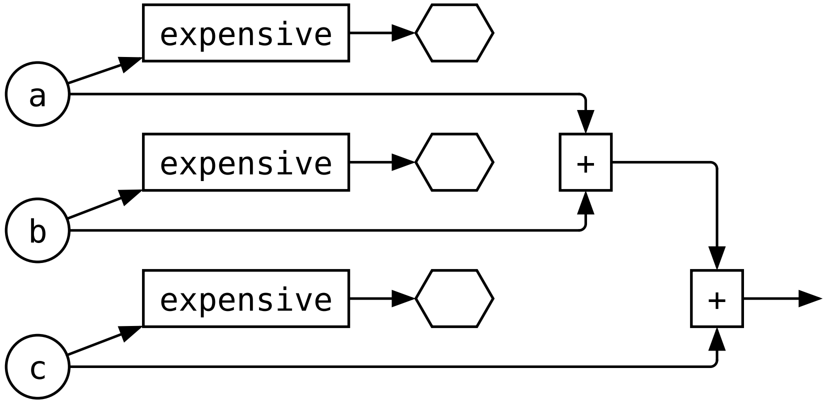
```
type PVal a
let pure   :                         a -> PVal a = ...
let map    :      (a -> b) -> PVal a -> PVal b = ...
let apply : PVal (a -> b) -> PVal a -> PVal b = ...

let assume : PVal (Dist a) -> PVal a = ...
let weight : PVal Float -> () = ...
```

Generating and Optimizing a Graph




```
map f (pure a) =
  pure (f a)
apply (pure f) a =
  map f a
map f (map g a) =
  map (f . g) a
map f (apply a b) =
  apply (map (lam a. f . a))
  a) b
```

Graphical Models



- This looks like a graphical model
  - ...with deterministic nodes
  - ...and observations as hexagons
- What about universality?

TreePPL is a Universal Probabilistic Programming Language

- Statically unbounded number of random variables
- How do we express this?

```tppl
model geometric(p: Real) => Int {
  assume c ~ Bernoulli(p);
  if c {
    return 1 + geometric(p);
  }
  return 1;
}
```

- Probabilistically guarded recursion
- **Problem:** this cannot be expressed with a (finite) applicative functor

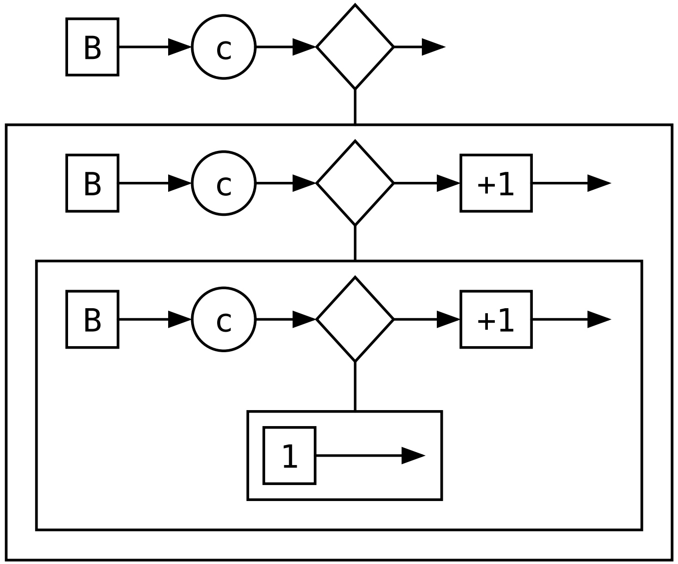
Solution: Monad

mcore

```mcore
type PVal a
let pure  :                            a -> PVal a = ...
let map   :       (a ->       b) -> PVal a -> PVal b = ...
let apply : PVal (a ->       b) -> PVal a -> PVal b = ...
let bind  :       (a -> PVal b) -> PVal a -> PVal b = ...

let assume : PVal (Dist a) -> PVal a = ...
let weight : PVal Float -> () = ...
```

- Intuition:
  - `apply` can run things “in parallel”.
  - `bind` must run things “in sequence”.

Manual Geometric Distribution as a Graph




```
model geometric(p: Real) => Int {
  assume c ~ Bernoulli(p);
  if c {
    return 1 + geometric(p);
  }
  return 1;
}
```
tppl

- `bind` builds and discards sub-graphs as needed
- This can be expensive

Conclusion

Not mentioned here and future work

- Implementation strategies (e.g., mutability or not)
- Integrating pruning, delayed sampling, etc.

Take-aways

- Applicative functor gives us speed
  - ...by modelling data-flow and avoiding recomputation
- Monad gives us expressivity
  - ...by nesting graphs
- Together, they generalize graphical models to handle universality

Thank you!