Probabilistic Programming: Use cases and implementation

Siddharth Bhat

IIIT Hyderabad

November 16, 2019
(FunctionalConf '19)

Outline

- How is this different from `System.Random`?
- What is Probabilistic programming, and when can I use it?
- A sketch of the algorithm.
- Hunt me down post-talk for a wealth of details.

Our primitives

```haskell
data Rand a = ...

instance Functor Rand
instance Applicative Rand
instance Monad Rand

-- | Convert a pure value into a Rand value
return :: a -> Rand a

-- | Get a random number in [0, 1)
uniform01 :: Rand Float

-- | Take `n` samples from a random variable
samples :: Int -> Rand a -> [a]

-- | take a Float, do *something*, and return no nothing
score :: Float -> Rand ()
```

First example – The same as System.Random

```haskell
-- | dice
dice :: Rand Int
dice = do
  u <- uniform01
  return $ 1 + floor (6*u)
```

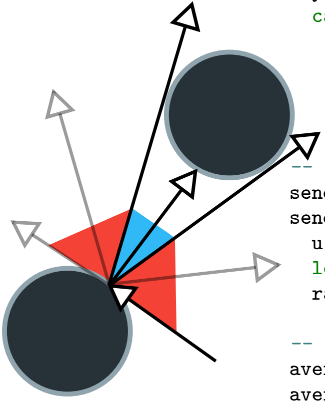
```haskell
-- | sum of dice
tossDice :: Rand Int
tossDice = do
    d1 <- dice
    d2 <- dice
    return $ d1 + d2
```

```haskell
main :: IO ()
main = do
  print $ sample 10 tossDice
  drawHistogram $ sample 100 tossDice
```

Output:

```
[7,6,7,10,5,8,12,8,9,6]
```

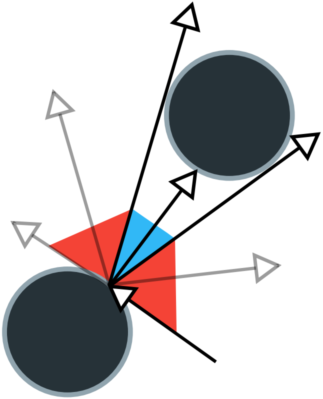
Raytracing (Default)



```haskell
-- | recursively raytrace
raytrace :: Ray -> Rand Color
raytrace r = do
  case getCollision r of
    Some (surface, loc) ->
     color' <- averageRays loc
     return $ mixColor surface color'
    None -> return backgroundColor

-- | Send a random ray
sendRandRay :: Position -> Rand Color
sendRandRay p =
  u <- uniform01
  let angle = 360 * u
  raytrace (makeRay p angle)

-- | Average rays sent from a location
averageRays :: Position -> Rand Color
averageRays p = do
  -- | computationally wasteful
  colors <- replicateM 100 (sendRandRay p)
  return $ averageColors colors

-- | Default background color.
backgroundColor = white
```

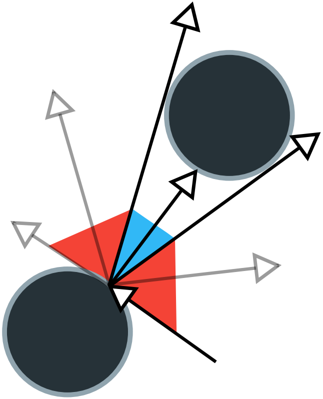
Raytracing (Scored)

```
raytrace :: Ray -> Rand Color
raytrace r = do
  case getCollision r of
    Some (surface, loc) ->
     color' <- averageRays loc
     return $ mixColor surface color'
    None -> return backgroundColor
```

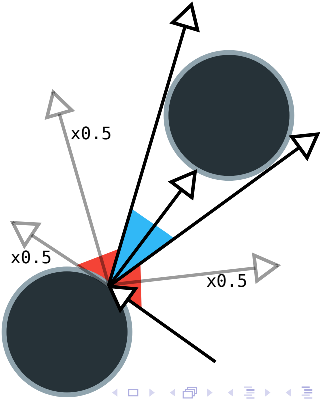


Raytracing (Scored)

```
raytrace :: Ray -> Rand Color
raytrace r = do
  case getCollision r of
    Some (surface, loc) ->
     color' <- averageRays loc
     return $ mixColor surface color'
    None -> return backgroundColor
```



```
raytrace' :: Ray -> Rand Color
raytrace' r = do
  case getCollision r of
    Some (surface, loc) ->
     color' <- averageRays loc
     return $ mixColor surface color'
    None -> do
     score 0.5 -- New!
     return backgroundColor
```



Exploring a complicated landscape

```haskell
-- | Naive understanding / Little knowledge when we begin
prior :: Rand a
prior = ...

-- | Learn as you go!
learn :: Rand a
learn = do
  value <- prior
  score (usefulness value)
  return value

-- | Generate samples according to unknown distribution
-- (Rays from the raytracing)
landscape :: [a]
landscape = samples 1000 learn
```

Program optimisation

- A hypothetical stack-based machine (eg. JVM bytecode!)
- Instructions are `Push`, `Add`, `Dup`, `Sub`

```
f(x) = x*2
Program(f) = [Push 2; Mul]
[BOT x] -Push 2-> [BOT x; 2] -Mul-> [BOT x*2]
```

Program optimisation

- A hypothetical stack-based machine (eg. JVM bytecode!)
- Instructions are `Push`, `Add`, `Dup`, `Sub`

  ```
  f(x) = x*2
  Program(f) = [Push 2; Mul]
  [BOT x] -Push 2-> [BOT x; 2] -Mul-> [BOT x*2]
  ```

- Want to automatically discover optimizations

  ```
  f(x) = x*2 = x+x
  Program(f) = [Dup; Add]
  [BOT x] -Dup-> [BOT x; x] -Add-> [BOT x+x]
  ```

  Faster (addition is faster than multiplication)

Program optimisation

- A hypothetical stack-based machine (eg. JVM bytecode!)
- Instructions are `Push`, `Add`, `Dup`, `Sub`

```
f(x) = x*2
Program(f) = [Push 2; Mul]
[BOT x] -Push 2-> [BOT x; 2] -Mul-> [BOT x*2]
```

- Want to automatically discover optimizations

```
f(x) = x*2 = x+x
Program(f) = [Dup; Add]
[BOT x] -Dup-> [BOT x; x] -Add-> [BOT x+x]
```

  Faster (addition is faster than multiplication)
- Neural nets do not work - no gradient $\frac{\partial loss}{\partial program}$

Program optimization pseudocode (Random)

Key idea: Randomly change the initial program and pick the best.

```
-- | Randomly change programs and return their performance
equivRandomProgram :: Program -> Rand (Performance, Program)
equivRandomProgram p = do
  p' <- modifyProgram p
  if semanticsEqual p p'
  then return (performance p', p')
  else return (0, p') -- A program that does not work has 0 perf.

-- | Take the random samples and pick the good performing ones
optimise :: Program -> Program
optimise p =
  let ps' = sample 100 (equivRandomProgram p)
  in snd $ maximumBy (\a b -> compare (fst a) (fst b)) ps'
```

Program optimisation pseudocode (Scored)

```
equivRandomProgram :: Program -> Rand (Performance, Program)
equivRandomProgram p = do
  p' <- modifyProgram p
  if semanticsEqual p p'
  then return (performance p', p')
  else return (0, p') -- A program that does not work has 0 perf.
```

Program optimisation pseudocode (Scored)

```
equivRandomProgram :: Program -> Rand (Performance, Program)
equivRandomProgram p = do
  p' <- modifyProgram p
  if semanticsEqual p p'
  then return (performance p', p')
  else return (0, p') -- A program that does not work has 0 perf.

equivRandomProgram' :: Program -> Rand (Performance, Program)
equivRandomProgram' p = do
 (perf, p) <- equivRandomProgram p
 let perf =
    if semanticsEqual p p'
      then 5 + performance p' -- Correct programs are given high score
      else performance p' -- Fast incorrect programs are also allowed
 score perf -- ^ Correct programs are more likely
 return (perf, p')
```

http://stoke.stanford.edu/

https://github.com/bollu/blaze/blob/master/notebooks/tutorial.ipynb

Transformations discovered by STOKE

- `f = 2 + 3`
  `(nparams: 0 | [IPush 2,IPush 3,IAdd])`

Transformations discovered by STOKE

- ```
  f = 2 + 3
  (nparams: 0 | [IPush 2,IPush 3,IAdd])
  ```
  ```
  [IPush 5] | score: 2.5
  // constant folding: 2 + 3 -> 5
  ```
  ```

Transformations discovered by STOKE

- `f = 2 + 3`
  `(nparams: 0 | [IPush 2,IPush 3,IAdd])`

  `[IPush 5] | score: 2.5`
  `// constant folding: 2 + 3 -> 5`
- `f(x) = 2 * x`
  `(nparams: 1 | [IPush 2,IMul])`

Transformations discovered by STOKE

- ```
  f = 2 + 3
  (nparams: 0 | [IPush 2,IPush 3,IAdd])
  ```
  ```
  [IPush 5] | score: 2.5
  // constant folding: 2 + 3 -> 5
  ```
- ```
  f(x) = 2 * x
  (nparams: 1 | [IPush 2,IMul])
  ```
  ```
  [IDup,IAdd] | score: 2.25
  // strength reduction: 2 * x -> x + x
  ```

Transformations discovered by STOKE

- ```
  f = 2 + 3
  (nparams: 0 | [IPush 2,IPush 3,IAdd])
  ```
  ```
  [IPush 5] | score: 2.5
  // constant folding: 2 + 3 -> 5
  ```
- ```
  f(x) = 2 * x
  (nparams: 1 | [IPush 2,IMul])
  ```
  ```
  [IDup,IAdd] | score: 2.25
  // strength reduction: 2 * x -> x + x
  ```
- ```
  f(x) = x & x
  (nparams: 1 | progInsts = [IDup,IAnd])
  ```
  ```

Transformations discovered by STOKE

- ```
  f = 2 + 3
  (nparams: 0 | [IPush 2,IPush 3,IAdd])
  ```
  ```
  [IPush 5] | score: 2.5
  // constant folding: 2 + 3 -> 5
  ```
- ```
  f(x) = 2 * x
  (nparams: 1 | [IPush 2,IMul])
  ```
  ```
  [IDup,IAdd] | score: 2.25
  // strength reduction: 2 * x -> x + x
  ```
- ```
  f(x) = x & x
  (nparams: 1 | progInsts = [IDup,IAnd])
  ```
  ```
  [] | score: 3.0
  // algebraic rewrite: x & x == x
  ```
  ```

STOKE: The intuition

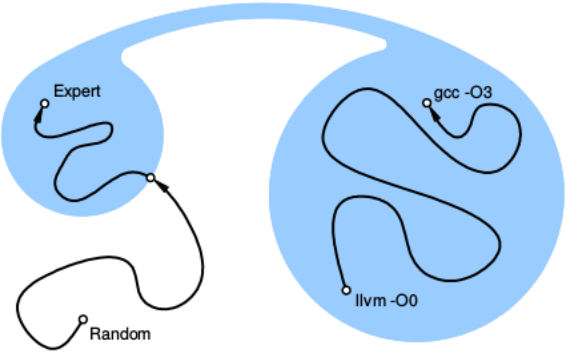


Figure 4. Abstract depiction of the search space for the Montgomery multiplication benchmark. O0 and O3 optimized codes occupy a densely connected part of the space which is easily traversed. Expert code occupies an entirely different region of the space which is reachable only by way of an extremely low probability path.

STOKE: The intuition

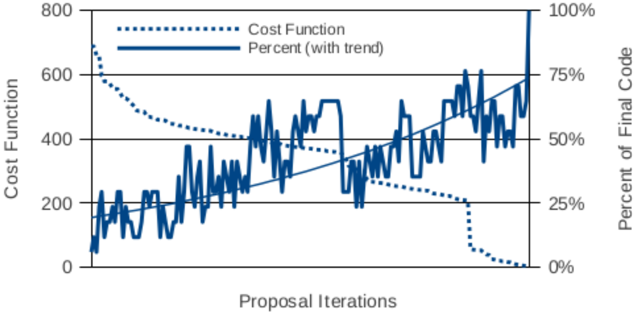


Figure 8. Cost function versus percentage of instructions which appear in the final zero-cost rewrite. Random search is an effective method for performing synthesis insofar as it is able to discover partially correct rewrites incrementally.

STOKE: A real-world example

```
# gcc -O3

.L0:
  movq rsi, r9
  movl ecx, ecx
  shrq 32, rsi
  andl 0xffffffff, r9d
  movq rcx, rax
  movl edx, edx
  imulq r9, rax
  imulq rdx, r9
  imulq rsi, rdx
  imulq rsi, rcx
  addq rdx, rax
  jae .L2
  movabsq 0x100000000, rdx
  addq rdx, rcx
.L2:
  movq rax, rsi
  movq rax, rdx
  shrq 32, rsi
  salq 32, rdx
  addq rsi, rcx
  addq r9, rdx
  adcq 0, rcx
  addq r8, rdx
  adcq 0, rcx
  addq rdi, rdx
  adcq 0, rcx
  movq rcx, r8
  movq rdx, rdi
```

```
# STOKE

.L0:
  shlq 32, rcx
  movl edx, edx
  xorq rdx, rcx
  movq rcx, rax
  mulq rsi
  addq r8, rdi
  adcq 0, rdx
  addq rdi, rax
  adcq 0, rdx
  movq rdx, r8
  movq rax, rdi
```

Gradient-free optimisation on a complicated landscape

```haskell
-- | Naive understanding / Little knowledge when we begin
prior :: Rand a
prior = ...

-- | Learn as you go!
learn :: Rand (Score, a)
learn = do
  value <- prior
  let s = score (usefulness value)
  return (s, value)

-- | Sample and pick best value (random programs)
-- | Works because sampler will "move" towards
-- scored regions!
best :: (Score, a)
best = maximumBy (\a b -> compare (fst a) (fst b))
         (samples 1000 learn)
```

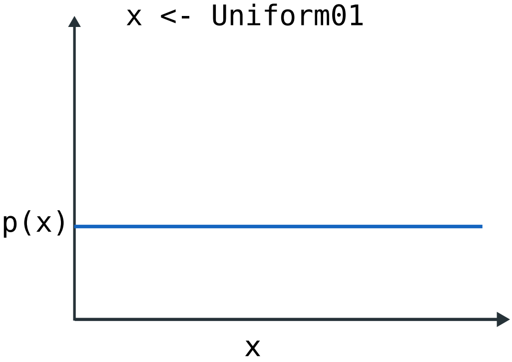
```
data Rand x where
    Ret :: x -> Rand x
    SampleUniform01 :: (Double -> Rand x) -> Rand x
    Score :: Float -> Rand x -> Rand x

instance Functor Rand
instance Applicative Rand
instance Monad Rand
```

(Rand is a free monad)

```
return :: x -> Rand x
return = Ret

uniform01 :: Rand Float
uniform01 = SampleUniform01 Ret

score :: Float -> Rand ()
score s = Score s (return ())
```

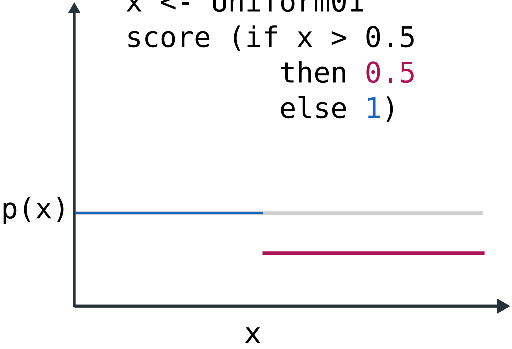
```haskell
-- | Run the computation _unweighted_,
-- | ignores scores.
sampleUnweighted :: RandomGen g => g -> Rand a -> (a, g)
sampleUnweighted g (Ret a) = (a, g)
sampleUnweighted g (SampleUniform01 f2my) =
  let (f, g') = random g in sample g' (f2my f)
sampleUnweighted g (Score f mx) =
  error "unable to handle Score"
```

Semantics of Score: Uniform probability



Semantics of Score: Custom ratios of probabilities!

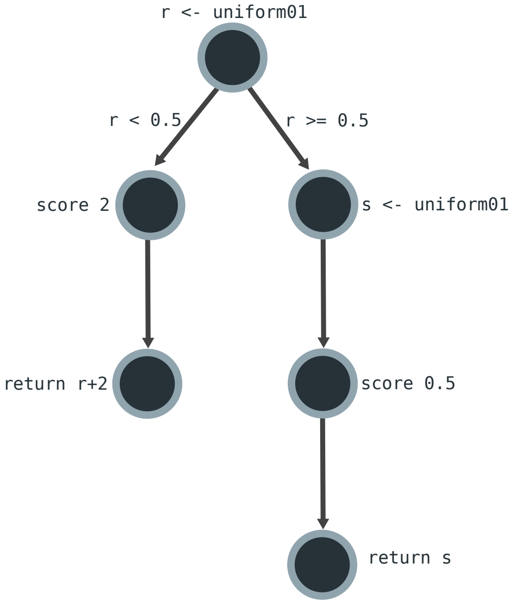
```
x <- Uniform01
score (if x > 0.5
          then 0.5
          else 1)
```



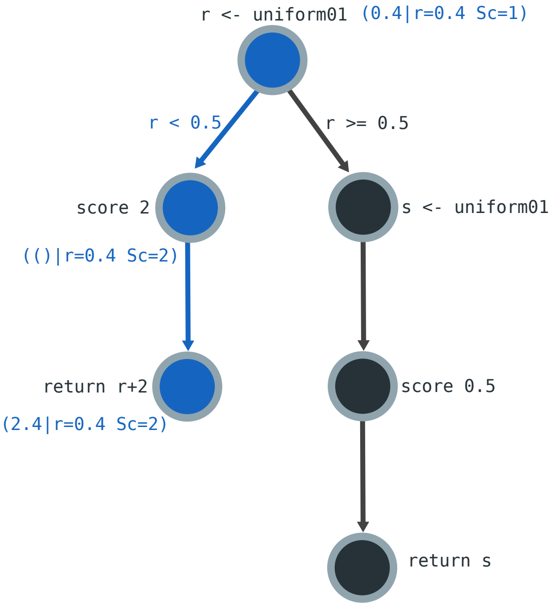
How do we handle Score?

- Game plan: convert `Rand a` to `Rand (Trace a)`.
- `Rand a` may contain `Score`, `Rand (Trace a)` will not.
- `Rand (Trace a)` will have extra information to make `Score` work.
- Can use `sampleUnweighted` on `Rand (Trace a)`. It will respect `Score`, but not use `Score`.

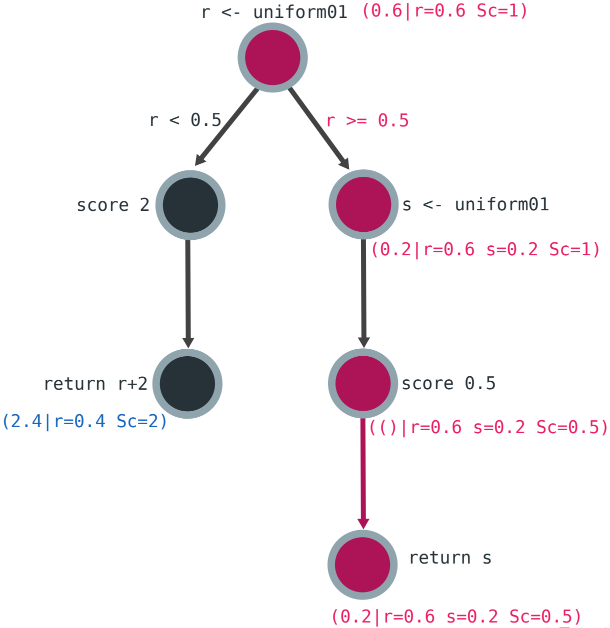
Traced Computations - Program



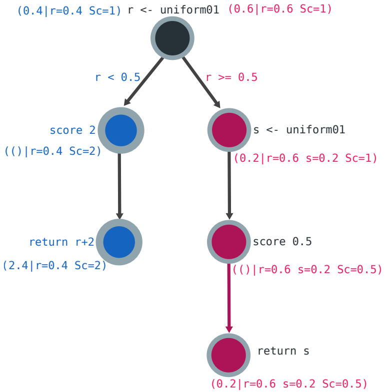
Traced Computations - Trace 1



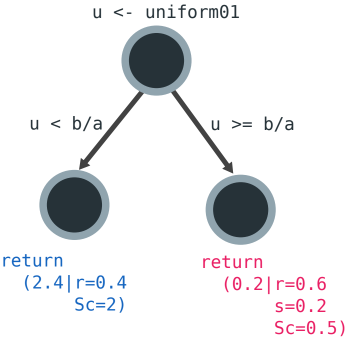
Traced Computations - Trace 2



Traced Computations - Trace 3




```
a = Accept(2.4|r=0.4 Sc=2)
  = 0.4 x len([0.4])
  = 0.4 x 1 = 0.4

b = Accept(0.2|r=0.6 s=0.2 Sc=0.5)
  = 0.5 x len([0.6 0.2])
  = 0.5 x 2
  = 1
```



Tracing: the data structure

```
-- | Trace all random choices made
data Trace a =
  Trace { tval :: a, -- ^ The value itself
          tscore :: Double, -- ^ The total score
          trs :: [Double] -- ^ The random numbers used
        }
```

Introspection of a Free monad

```
-- | Trace a random computation (remove Score)
traceR :: Rand x  -- ^ Original computation
  -> Rand (Trace x) -- ^ Traced computation
traceR (Ret x) = Ret (Trace x 1.0 [])
traceR (SampleUniform01 mx) = do
  r <- sample01
  trx <- traceR $ mx r
  return $ trx { trs=trs ++ [r]}
traceR (Score s mx) = do -- ^ No score in the RHS
  trx <- traceR $ mx
  return $ trx { tscore = tscore*s}
```

Introspection of a Free monad

```haskell
-- | Trace a random computation (remove Score)
traceR :: Rand x  -- ^ Original computation
  -> Rand (Trace x) -- ^ Traced computation
traceR (Ret x) = Ret (Trace x 1.0 [])
traceR (SampleUniform01 mx) = do
  r <- sample01
  trx <- traceR $ mx r
  return $ trx { trs=trs ++ [r]}
traceR (Score s mx) = do -- ^ No score in the RHS
  trx <- traceR $ mx
  return $ trx { tscore = tscore*s}

-- | Send a particular sequence of randomness
feedRandomness :: [Float] -- ^ Randomness to use
  -> Rand a -- ^ Original computation
  -> Rand a -- ^ Fixed computation
feedRandomness _ (Ret x) = Ret x
feedRandomness
  (r:rs) (SampleUniform01 f) =
    feedRandomness rs (f r)
feedRandomness rs (Score s plx) =
  Score s $ injectRandomness rs plx
```

Metropolis Hastings

```
mhStep :: Rand (Trace x) -- ^ proposal
        -> Trace x -- ^ current position
        -> Rand (Trace x) -- ^ new
mhStep r trace = do
  -- | Return the original randomness, perturbed
  rands' <- perturbRandomness (trs trace)
  -- | Run the original computation with the perturbation
  trace' <- feedRandomness rands' r
  let ratio = trAccept trace' / trAccept trace
  r <- sample01
  return $ if r < ratio then trace' else trace
```

```
trAccept :: Trace x -> Double
trAccept tx =
  tscore tx *
  fromIntegral (length (trs tx))
```

```
perturbRandomness :: [Double]
  -> Rand [Double]
perturbRandomness rands = do
  -- | Random index
  ix <- choose [0..(length rands-1)]
  r <- sample01 -- ^ random val
  -- | Replace random index
  -- with random val.
  return $ replaceListAt ix r rands
```

Odds and Ends

```haskell
-- | run the computation after taking weights into account
samples :: Int -> Rand x -> Rand [x]
samples 0 _ = return []
samples n r =
  let tracedR = traceR r
      -- go :: Int -> Rand (Trace x) -> Rand (Trace [x])
      go 0 _ = return []
      go n tx = do
        tx' <- repeatM 10 (mhStep tracedR) $ tx -- !
        txs <- go (n-1) tx'
        return (tx:txs)
  in do
      seed <- findNonZeroTrace $ tracedR
      tracedRs <- go n seed
      return $ map tval tracedRs

-- | Find a starting position that does not have probability 0
findNonZeroTrace :: Rand (Trace x) -> Rand (Trace x)
findNonZeroTrace tracedR = do
  trace <- tracedR
  if tscore trace /= 0
  then return $ trace
  else findNonZeroTrace tracedR
```

Takeaways

- Probabilistic Programming is a powerful model of computation.
- Free monads is essential: allows for introspection.
- Implementation is deceptively simple (hopefully!)

Thank you!

Questions?

(A huge thank you to everyone at `tweag.io` who read the literature with me!)

References I

Noah Goodman et al. "Church: a language for generative models". In: *arXiv preprint arXiv:1206.3255* (2012).

Adam Michał Ścibior. "Formally justified and modular Bayesian inference for probabilistic programs". PhD thesis. University of Cambridge, 2019.

Adam Ścibior, Zoubin Ghahramani, and Andrew D Gordon. "Practical probabilistic programming with monads". In: *ACM SIGPLAN Notices*. Vol. 50. 12. ACM. 2015, pp. 165–176.

Use case: Bayesian updates

```
predictCoinBias :: [Int] -> Rand Double
predictCoinBias flips = do
  b <- sample01
  forM_ flips $ \f -> do
    -- | Maximum a posterior
    score $ if f == 1 then b else (1 - b)
  return $ b
```

```
predictCoinBiasNoData :: Rand Double
predictCoinBiasNoData = predictCoinBias []
```

```
predictCoinBias0 :: Rand Double
predictCoinBias0 = predictCoinBias [0]
```

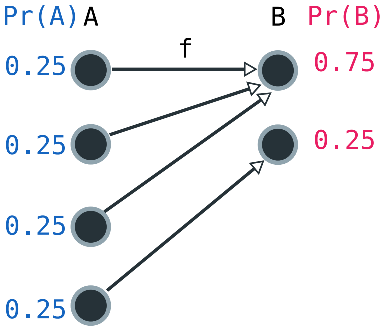
```
predictCoinBias01 :: Rand Double
predictCoinBias01 = predictCoinBias [0, 1]
```

Use case: Sample from arbitrary distributions

```
sampleSinSq :: Rand Double
sampleSinSq = do
  x <- (6 *) <$> sample01
  score $ (sin x) * (sin x)
  return $ x
```

Deep dive into MCMC methods

Probability Distributions: the tension



- $f : A \to B$ is known.
- $f^{-1} : B \to A$ is required for the density.
- This is intractable.