

Probabilistic Programming: Use cases and implementation

Siddharth Bhat

IIIT Hyderabad

November 15, 2019
(FunctionalConf '19)

Our primitives

```
-- | Convert a pure value into a Rand value
```

```
return :: a -> Rand a
```

```
-- | Get a random number
```

```
uniform01 :: Rand Float
```

```
-- | Take 'n' samples from a random variable
```

```
samples :: Int -> Rand a -> [a]
```

```
-- | take a Float, do *something*, and return nothing
```

```
score :: Float -> Rand ()
```

First example – The same as `System.Random`

```
-- | dice
dice :: Rand Int
dice = do
  u <- uniform01
  return $ floor (7*u)

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

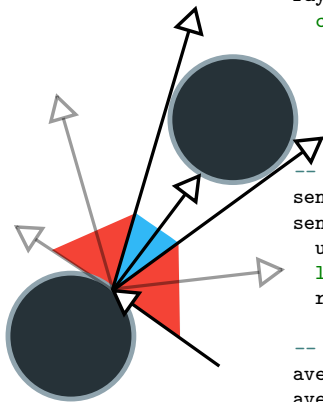
main :: IO ()
main = do
  print $ sample 10 tossDice
  drawHistogram $ sample 100 tossDice
```

Output:

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



Raytracing (Default)



```
-- / 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

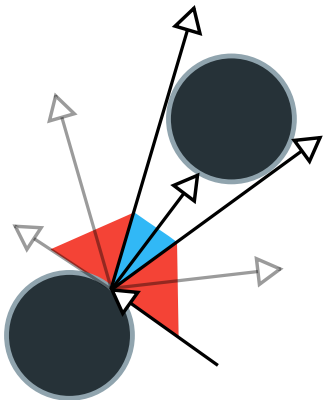
-- / Send a random ray
sendRandRay :: Position -> Rand Color
sendRandRay p =
  u <- uniform01
  let angle = 360 * u
  raytrace (makeRay p angle)

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

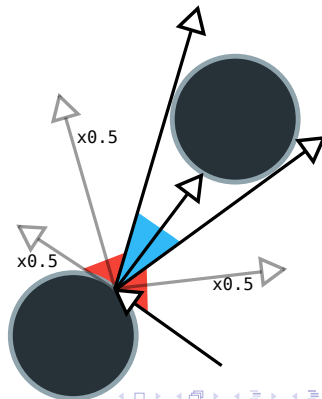
-- / Default background color.
backgroundColor = white
```

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
```



```
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

```
-- | 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

```
-- | 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 (Scored)

```
equivRandomProgram' :: Program -> Rand (Performance, Program)
equivRandomProgram' p = do
  (perf, p) <- equivRandomProgram p
  let perf =
    if semanticsEqual p p'
    then performance p'
    else 0
  score perf -- ^ Correct programs are more likely
  return (perf, p')
```

```
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.
```

<http://stoke.stanford.edu/>

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

Use case: Transformations discovered by STOKe

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

Use case: Transformations discovered by STOKe

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

Use case: 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])`

Use case: 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`

Use case: Transformations discovered by STOKe

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

Use case: Transformations discovered by STOKe

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

Optimisation on a complicated landscape

```
-- / 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 :: Double -> Rand x -> Rand x

instance Functor Rand
instance Applicative Rand
instance Monad Rand

(Rand is a free monad)
```



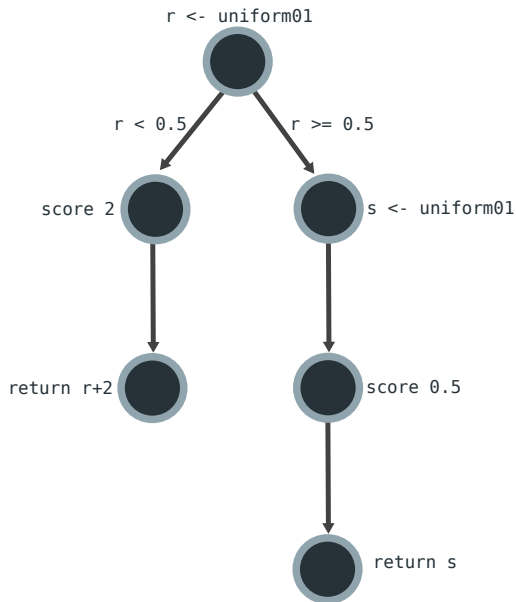
```

-- / Run the computation _unweighted_,
-- / ignores scores.
sample :: RandomGen g => g -> Rand a -> (a, g)
sample g (Ret a) = (a, g)
sample g (SampleUniform01 f2my) =
    let (f, g') = random g in sample g' (f2my f)
sample g (Score f mx) = sample g mx -- Ignore score

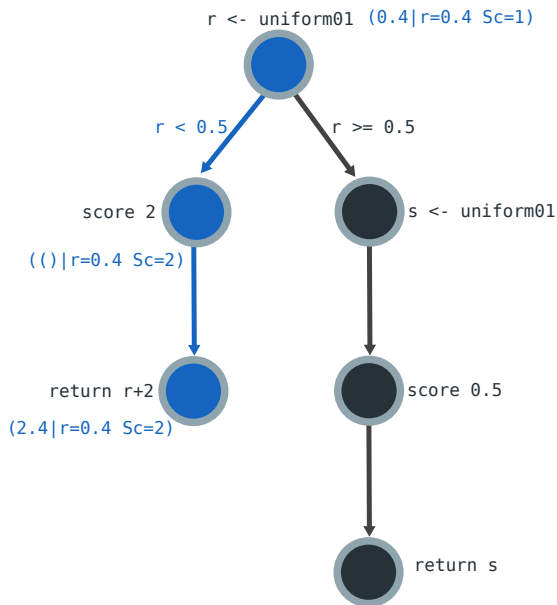
```

MCMC methods

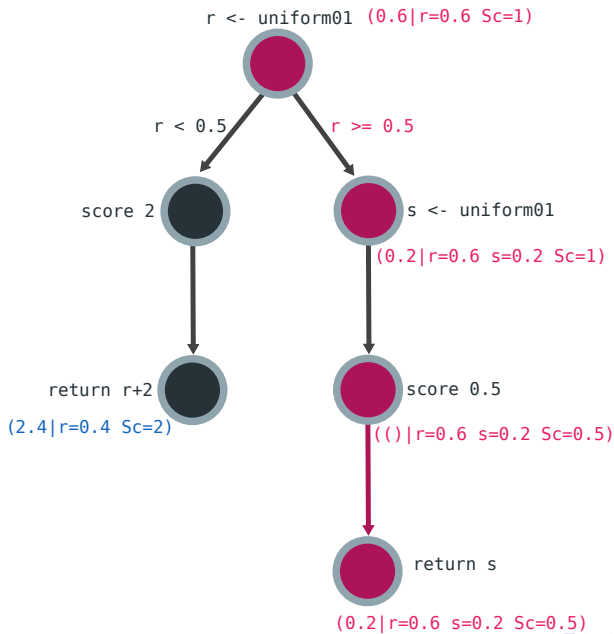
Traced Computations - Program



Traced Computations - Trace 1



Traced Computations - Trace 2



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
        }

-- / Trace a random computation.
traceR :: Rand x -> Rand (Trace x)
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
  trx <- traceR $ mx
  return $ trx { tscore = tscore*s}
```

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

```
-- | 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
```


Thank you!

Questions?



tweag.io

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

References I

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
```

