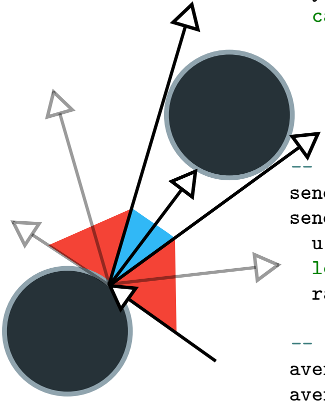
Our primitives

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

-- | Get a random number
uniform01 :: Rand Float

-- | take a Float, do *something*, and return nothing
score :: Float -> Rand ()
```

Raytracing (Default)

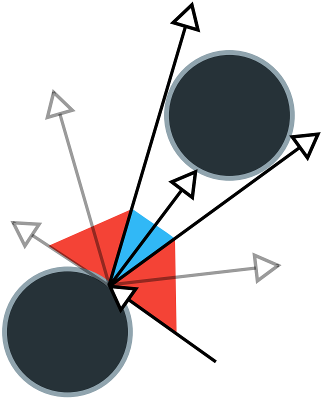


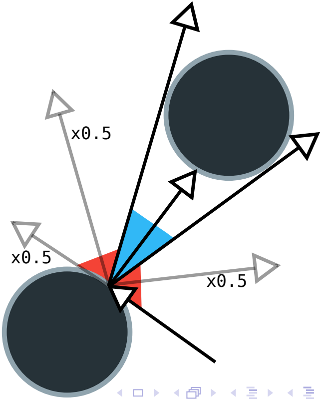
```
-- | recursively raytrace
raytrace :: Ray -> Rand Color
raytrace r = do
  case getCollision r of
    Some (surface, loc) ->
     color' <- averageRays loc
     return $ mixColor surface color'
    None -> return backgroundColor

-- | 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 -- Reduce
      return backgroundColor
```





Program optimisation

```
equivRandomProgram' :: Program -> Rand (Performance, Progra
equivRandomProgram' p = do
 (perf, p) <- equivRandomProgram p
 score perf
 return (perf, p')

equivRandomProgram :: Program -> Rand (Performance, Progran
equivRandomProgram p = do
  p' <- modifyProgram p
  if semanticsEqual p p'
  then return (performance p', p')
  else equivRandomProgram p -- ^ try again


optimise :: Program -> Program
optimise p =
  let ps' = sample 100 (equivRandomProgram p)
  in snd $ maximumBy (compare . fst) ps'
```

Learning from prior experience

```haskell
-- | Naive understanding, in the beginning of the process
prior :: Rand a
prior = ...

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

Motivation

```haskell
-- | fair dice
dice :: Rand Int
dice = choose [1, 2, 3, 4, 5, 6]

tossDice :: Rand Int
tossDice = do
    d1 <- dice
    d2 <- dice
    return $ d1 + d2
```

If the dice roll is prime,

```haskell
tossDicePrime :: Rand Int
tossDicePrime = do
    d <- tossDice
    score $ if prime d then 1 else 0
    return $ d
```

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

```haskell
-- | 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

```haskell
-- | Trace all random choices made when generating this va
data Trace a =
  Trace { tval :: a, -- ^ The value itself
          tscore :: Double, -- ^ The total score
          trs :: [Double] -- ^ The ranom numbers used
        }
-- | Lift a pure value into a Trace value
mkTrace :: a -> Trace a
mkTrace a = Trace a 1.0 []
-- | multiply a score to a trace
scoreTrace :: Double -> Trace a -> Trace a
scoreTrace f Trace{..} = Trace{tscore = tscore * f, ..}
-- | Prepend randomness
recordRandomness :: Double -> Trace a -> Trace a
recordRandomness r Trace{..} = Trace { trs = r:trs, ..}

-- | Trace a random computation.
-- We know what randomness is used
traceR :: Rand x -> Rand (Trace x)
traceR (Ret x) = Ret (mkTrace x)
```

– — Return a trace-adjusted MH computation

```haskell
mhStep :: Rand (Trace x) -- ^ proposal
       -> Trace x -- ^ current position
       -> Rand (Trace x)
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 = traceAcceptance trace' / traceAcceptance trac
  r <- sample01
  return $ if r < ratio then trace' else trace

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

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

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

-- | run the computatation after taking weights into accou
weighted :: MCMC x => Int -> Rand x -> Rand [x]
weighted 0 _ = return []
weighted 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
```

Payoff!

```haskell
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]
```

More fun stuff: sample from arbitrary distributions

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