Modular construction of Bayesian inference algorithms

Adam Ścibior[1,2], Zoubin Ghahramani[1]
University of Cambridge[1], MPI Tübingen[2]

UNIVERSITY OF CAMBRIDGE

Outline

Implementing inference algorithms is difficult

- new algorithms often build on existing ones
- modularity in implementation helps with prototyping
- we can achieve modularity with monad transformers
- easier to implement, easier to test
- proof-of-concept library in Haskell

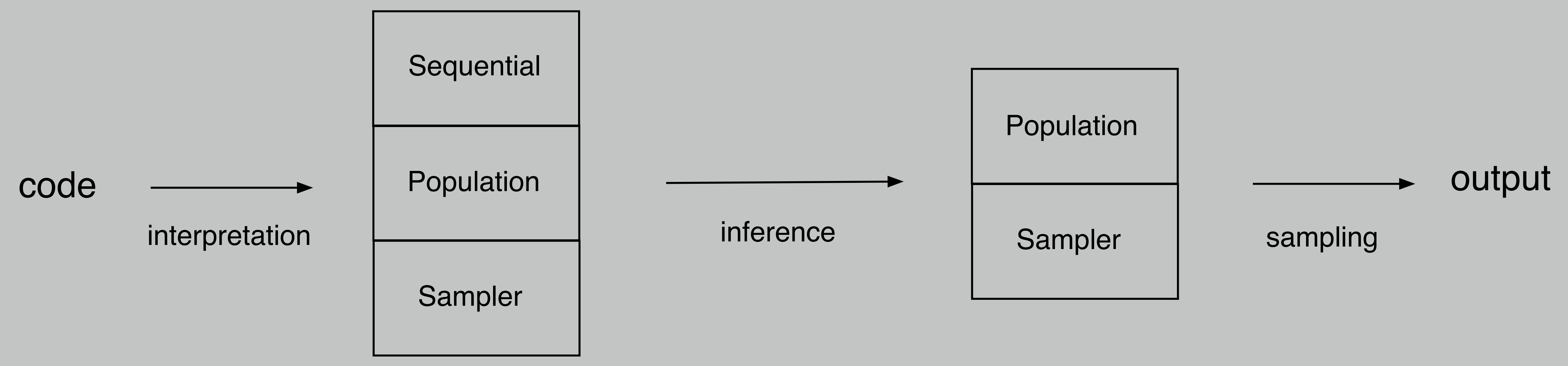
Probability monads

A probability monad has the following interface:

- create a Dirac distribution
- apply the sum rule or the product rule
- draw a random variable from a simple distribution
- accumulate likelihood

This is sufficient to interpret any probabilistic program.

Illustration



Building blocks

Layers (monad transformers)

`Sampler`	`pseudo-random sampler (no conditioning)`
`Weighted`	`accumulates likelihood as a weight`
`Enumerator`	`exhaustively enumerates discrete variables`
`Population`	`maintains a population of weighted values`
`Sequential`	`suspendable models (e.g. time series)`
`Trace`	`maintains the full execution trace`
`Conditional`	`conditions on selected variables`
`Prior`	`discards all observations`
`Rejection`	`rejects configurations with zero likelihood`

Inference transformations

```
sampleIO    :: Samp a   -> IO a             draw a sample
weighted    :: Weig m a -> m (a,R)          importance sample
enumerate   :: Enum m a -> m [(a,R)]        enumerate discrete
spawn       :: Int -> Pop m a  -> Pop m a   expand population
resample    :: Pop m a  -> Pop m a          simple resampling
collapse    :: Pop m a  -> m a              pick one value
advance     :: Seq m a  -> Seq m a          one step forward
finish      :: Seq m a  -> m a              run to the end
mhStep      :: Tr m a   -> Tr m a           single mh step
marginal    :: Tr m a   -> m a              discard trace
conditional :: [R] -> Con m a -> m a        conditional dist
density     :: [R] -> Con m a -> m R        pseudo-density
prior       :: Pri m a  -> m a              prior distribution
rejection   :: Rej m a  -> m a              rejection sampling
```

Composition of inference algorithms

```
smc :: Int -> Int -> Seq (Pop Samp) a -> Pop Samp a
smc k n = marginal . flatten . step ^ k . init where
  init = hoistS (spawn n)
  step = advance . hoistS resample
```

```
smcrm :: Int -> Int -> Seq (Tr (Pop Samp)) a -> Pop Samp a
smcrm k n = marginal . flatten . step ^ k . init where
  init = hoistS (hoistT (spawn n))
  step = advance . hoistS (mhStep . hoistT resample)
```

Example: different interpretations of the same model

```
model :: MonadBayes m => m Bool
model = do
  b <- bernoulli 0.4
  x <- if b then normal 0 1 else beta 1 1
  observe x == 0.5
  return b
model :: MonadBayes m => Seq (Pop m) Bool
model = do
  b <- bernoulli 0.4
  w = if b then normalPDF 0 1 0.5 else betaPDF 1 1 0.5
  suspend
  return [(b,w)]
```

```
model :: Weighted Sampler Bool
model = \rng ->
  b = sample rng (bernoulli 0.4)
  w = if b then normalPDF 0 1 0.5 else betaPDF 1 1 0.5
  return (b,w)

model :: Enum Bool
model = [(True, 0.14), (False, 0.6)]
```

Deterministic testing

- MH kernel preserves the posterior
  `enumerate model == enumerate (model >>= kernel)`
- SMC does not introduce bias
  `enumerate model == enumerate (collapse (smc k n model))`

Future work

- more building blocks
- new inference algorithms
- implementation in other languages
- performance evaluation

ams240@cam.ac.uk

https://github.com/adscib/monad-bayes