Running from the past

Dan Piponi

February 2024

1 Preface

Functional programming encourages us to program without mutable state. Instead we compose functions that can be viewed as state transformers. It's a change of perspective that can have a big impact on how we reason about our code. But it's also a change of perspective that can be useful in mathematics and I'd like to give an example: a really beautiful technique that alows you to sample from the infinite limit of a probability distribution without needing an infinite number of operations. (Unless you're infinitely unlucky!)

2 Markov Chains

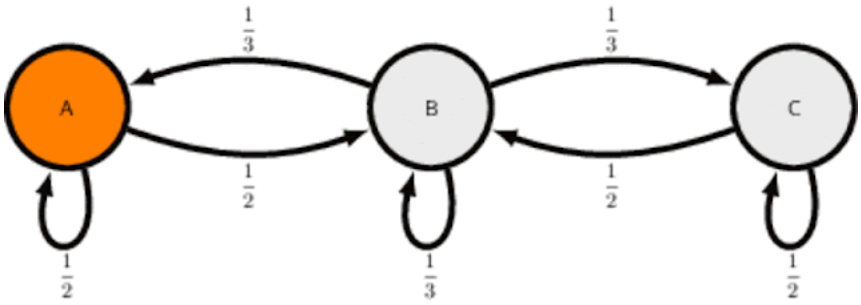
A Markov chain is a sequence of random states where each state is drawn from a random distribution that possibly depends on the previous state, but not on any earlier state. So it is a sequence $X_0, X_1, X_2, \ldots$ such that $P(X_{i+1} = x\texttt{X\_0,X\_1,\ldots,X\_i) = P(X\_\{i+1\}=x}X_i)$ for all $i \geq 0$. A basic example might be a model of the weather in which each day is either sunny or rainy but where it's more likely to be rainy (or sunny) if the previous day was rainy (or sunny). (And to be technically correct: having information about two days or earlier doesn't help us if we know yesterday's weather.)

Like imperative code, this description is stateful. The state at step $i+1$ depends on the state at step i. Probability is often easier to reason about when we work with independent identically drawn random variables and our X_i aren't of this type. But we can eliminate the state from our description using the same method used by functional programmers.

Let's choose a Markov chain to play with. I'll pick one with 3 states called A, B and C and with transition probabilities given by $P(X_{i+1} = y | X_i = x) = T_{xy}$ where

$$T = \begin{pmatrix} \frac{1}{2} & \frac{1}{2} & 0 \\ \frac{1}{3} & \frac{1}{3} & \frac{1}{3} \\ 0 & \frac{1}{2} & \frac{1}{2} \end{pmatrix}$$

Here's a diagram illustrating our states:



3 Implementation

First some imports:

```
{-# LANGUAGE LambdaCase #-}
{-# LANGUAGE TypeApplications #-}

import Data.Sequence(replicateA, filter)
import System.Random hiding (uniform)
import Control.Monad.State
import Control.Monad
import Prelude hiding (filter)
import Data.List hiding (filter)
import Data.Array
```

And now the type of our random variable:

```
data ABC = A | B | C deriving (Eq, Show, Ord, Enum, Bounded)
```

We are now in a position to simulate our Markov chain. First we need some random numbers drawn uniformly from $[0, 1]$:

```
uniform :: (RandomGen gen, MonadState gen m) => m Double
uniform = state random
```

And now the code to take a single step in the Markov chain:

```
step :: (RandomGen gen, MonadState gen m) => ABC -> m ABC
step A = do
    a <- uniform
    if a < 0.5
        then return A
        else return B
step B = do
    a <- uniform
    if a < 1/3.0
        then return A
        else if a < 2/3.0
            then return B
            else return C
step C = do
    a <- uniform
    if a < 0.5
        then return B
        else return C
```

Notice how the `step` function generates a new state at random in a way that depends on the previous state. The `m ABC` in the type signature makes it clear that we are generating random states at each step.

We can simulate the effect of taking n steps with a function like this:

```
steps :: (RandomGen gen, MonadState gen m) =>
         Int -> ABC -> m ABC
steps 0 i = return i
steps n i = do
    i <- steps (n-1) i
```

```
    step i
```

We can run for 100 steps, starting with A, with a line like so:

```
*Main> evalState (steps 3 A) gen
B
```

The starting state of our random number generator is given by `gen`.

Consider the distribution of states after taking n steps. For Markov chains of this type, we know that as n goes to infinity the distribution of the nth state approaches a limiting "stationary" distribution. There are frequently times when we want to sample from this final distribution. For a Markov chain as simple as this example, you can solve exactly to find the limiting distribution. But for real world problems this can be intractable. Instead, a popular solution is to pick a large n and hope it's large enough. As n gets larger the distribution gets closer to the limiting distribution. And that's the problem I want to solve here - sampling from the limit. It turns out that by thinking about random functions instead of random states we can actually sample from the limiting distribution exactly.

4 Some random functions

Here is a new version of our random step function:

```
step' :: (RandomGen gen, MonadState gen m) =>
         m (ABC -> ABC)
step' = do
    a <- uniform
    return $ \case
        A -> if a < 0.5 then A else B
        B -> if a < 1/3.0
                then A
                else if a < 2/3.0 then B else C
        C -> if a < 0.5 then B else C
```

In many ways it's similar to the previous one. But there's one very big difference: the type signature `m (ABC -> ABC)` tells us that it's returning a random function, not a random state. We can simulate the result of

taking 10 steps, say, by drawing 10 random functions, composing them, and applying the result to our initial state:

```
steps' :: (RandomGen gen, MonadState gen m) =>
          Int -> m (ABC -> ABC)
steps' n = do
  fs <- replicateA n step'
  return $ foldr (flip (.)) id fs
```

Notice the use of `flip`. We want to compose functions $f_{n-1} \circ f_{n-2} \circ \ldots \circ f_0$, each time composing on the left by the new f. This means that for a fixed seed `gen`, each time you increase n by 1 you get the next step in a single simulation: (BTW I used `replicateA` instead of `replicateM` to indicate that these are independent random draws. It may be well known that you can use `Applicative` instead of `Monad` to indicate independence but I haven't seen it written down.)

```
*Main> [f A | n <- [0..10], let f = evalState (steps' n) gen]
[A,A,A,B,C,B,A,B,A,B,C]
```

When I first implemented this I accidentally forgot the `flip`. So maybe you're wondering what effect removing the `flip` has? The effect is about as close to a miracle as I've seen in mathematics. It allows us to sample from the limiting distribution in a finite number of steps!

Here's the code:

```
steps_from_past :: (RandomGen gen, MonadState gen m) =>
                   Int -> m (ABC -> ABC)
steps_from_past n = do
  fs <- replicateA n step'
  return $ foldr (.) id fs
```

We end up building $f_0 \circ f_1 \ldots \circ f_{n-1}$. This is still a composition of n independent identically distributed functions and so it's still drawing from exactly the same distribution as `steps'`. Nonetheless, there is a difference: for a particular choice of seed, `steps_from_past n` no longer gives us a sequence of states from a Markov chain. Running with argument n draws a random composition of n functions. But if you increase n by 1 you don't add a new step at the end. Instead you effectively restart the Markov chain with a new first step generated by a new random seed.

Try it and see:

```
*Main> [f A | n <- [0..10], let f =
        evalState (steps_from_past n) gen]
[A, A, A, A, A, A, A, A, A, A]
```

Maybe that's surprising. It seems to get stuck in one state. In fact, we can try applying the resulting function to all three states.

```
*Main> [fmap f [A, B, C] | n <- [0..10],
        let f = evalState (steps_from_past n) gen]
[[A,B,C],[A,A,B],[A,A,A],[A,A,A],[A,A,A],[A,A,A],
 [A,A,A],[A,A,A],[A,A,A],[A,A,A],[A,A,A]]
```

In other words, for n large enough we get the constant function.

Think of it this way: If `f` isn't injective then it's possible that two states get collapsed to the same state. If you keep picking random `f`'s it's inevitable that you will eventually collapse down to the point where all arguments get mapped to the same state. Once this happens, we'll get the same result no matter how large we take n. If we can detect this then we've found the limit of $f_0 \circ f_1 \ldots \circ f_{n-1}$ as n goes to infinity. But because we know composing forwards and composing backwards lead to draws from the same distribution, the limiting backward composition must actually be a draw from the same distribution as the limiting forward composition. That `flip` can't change what probability distribution we're drawing from - just the dependence on the seed. So the value the constant function takes is actually a draw from the limiting stationary distribution.

We can code this up:

```
all_equal :: (Eq a) => [a] -> Bool
all_equal [] = True
all_equal [_] = True
all_equal (a : as) = all (== a) as

test_constant :: (Bounded a, Enum a, Eq a) =>
                 (a -> a) -> Bool
test_constant f =
    all_equal $ map f $ enumFromTo minBound maxBound
```

This technique is called coupling from the past. It's "coupling" because

we've arranged that different starting points coalesce. And it's "from the past" because we're essentially asking answering the question of what the outcome of a simulation would be if we started infinitely far in the past.

```
couple_from_past :: (RandomGen gen, MonadState gen m,
                     Enum a, Bounded a, Eq a) =>
                  m (a -> a) -> (a -> a) -> m (a -> a)
couple_from_past step f = do
    if test_constant f
        then return f
        else do
            f' <- step
            couple_from_past step (f . f')
```

We can now sample from the limiting distribution a million times, say:

```
gen = mkStdGen 669
main = do
  let samples = fmap ($ A) $ evalState (
         replicateA 1000000 (couple_from_past step' id)) gen
```

We can now count how often `A` appears:

```
  print $ fromIntegral (
          length $ filter (== A) samples)/1000000
```

```
0.285748
```

That's a pretty good approximation to $\frac{2}{7}$, the exact answer that can be found by finding the eigenvector of the transition matrix corresponding to an eigenvalue of 1.

5 Notes

The technique of Coupling from the past first appeared in a paper by Propp and Wilson. The paper Iterted random functions by Persi Diaconis gave me a lot of insight into it. Note that the code above is absolutely not how you'd implement this for real. I wrote the code that way so that I

could switch algorithm with the simple removal of a `flip`. In fact, with some clever tricks you can make this method work with state spaces so large that you couldn't possibly hope to enumerate all starting states to detect if convergence has occurred. Or even with uncountably large state spaces. But I'll let you read the Propp-Wilson paper to find out how.