The Fisher–Yates shuffle

Manuel Eberl

February 23, 2021

Abstract
This work defines and proves the correctness of the Fisher–Yates shuffle [1, 2, 3] for shuffling – i.e. producing a random permutation – of a list. The algorithm proceeds by traversing the list and in each step swapping the current element with a random element from the remaining list.

Contents

1 Fisher–Yates shuffle 2
   1.1 Swapping elements in a list ................. 2
   1.2 Random Permutations .......................... 3
   1.3 Shuffling Lists .................................. 3
   1.4 Forward Fisher-Yates Shuffle ................. 4
   1.5 Backwards Fisher-Yates Shuffle ............... 4
   1.6 Code generation test ........................... 5
1 Fisher–Yates shuffle

theory Fisher-Yates
  imports HOL−Probability.Probability
begin

lemma integral-pmf-of-multiset:
  fixes A :: 'a multiset
  shows (∫ f x. (f x :: real) * measure-pmf (pmf-of-multiset A)) = (∑ x∈set-mset A. of-nat (count A x) * f x) / of-nat (size A)
  ⟨proof ⟩

lemma pmf-bind-pmf-of-multiset:
  fixes A :: 'a multiset
  shows pmf (pmf-of-multiset A >>= f) y = (∑ x∈set-mset A. of-nat (count A x) * pmf (f x) y) / of-nat (size A)
  ⟨proof ⟩

lemma pmf-map-inj-inv:
  assumes inj-on f p
  assumes ∀ x. f' (f x) = x
  shows pmf (map-pmf f p) x = (if x ∈ range f then pmf p (f' x) else 0)
  ⟨proof ⟩

1.1 Swapping elements in a list

definition swap where swap xs i j = xs[i := xs!j, j := xs ! i]

lemma length-swap [simp]: length (swap xs i j) = length xs
  ⟨proof ⟩

lemma swap-eq-Nil-iff [simp]: swap xs i j = [] ←→ xs = []
  ⟨proof ⟩

lemma nth-swap: i < length xs ⇒ j < length xs ⇒
  swap xs i j ! k = (if k = i then xs ! j else if k = j then xs ! i else xs ! k)
  ⟨proof ⟩

lemma map-swap: i < length xs ⇒ j < length xs ⇒ map f (swap xs i j) = swap (map f xs) i j
  ⟨proof ⟩

lemma swap-swap: i < length xs ⇒ j < length xs ⇒ swap (swap xs i j) j i = xs
  ⟨proof ⟩

lemma mset-swap: i < length xs ⇒ j < length xs ⇒ mset (swap xs i j) = mset xs
  ⟨proof ⟩

lemma hd-swap-0: i < length xs ⇒ hd (swap xs 0 i) = xs ! i
1.2 Random Permutations

First, we prove the intuitively obvious fact that choosing a random permutation of a multiset can be done by first randomly choosing the first element and then randomly choosing the rest of the list.

**lemma** pmf-of-set-permutations-of-multiset-nonempty:

**assumes** (\( A :: 'a multiset \) \( \neq \) \{\#\})

**shows** pmf-of-set (permutations-of-multiset \( A \)) =

\[
\begin{align*}
&\text{do } (x \leftarrow \text{pmf-of-multiset } A; \\
&\quad xs \leftarrow \text{pmf-of-set} \left( \text{permutations-of-multiset} \left( A - \{\#x\}\right)\right); \\
&\quad \text{return-pmf} \left( x \# xs \right)
\end{align*}
\]

(\( \text{is } \text{lhs } = \text{rhs} \))

1.3 Shuffling Lists

We define shuffling of a list as choosing from the set of all lists that correspond to the same multiset uniformly at random.

**definition** shuffle :: 'a list \Rightarrow 'a list pmf where

\( \text{shuffle } xs = \text{pmf-of-set} \left( \text{permutations-of-multiset} \left( \text{mset } xs \right)\right) \)

**lemma** shuffle-empty [simp]: shuffle [] = return-pmf []

(\( \text{proof} \))

**lemma** shuffle-singleton [simp]: shuffle [x] = return-pmf [x]

(\( \text{proof} \))

The crucial ingredient of the Fisher–Yates shuffle is the following lemma, which decomposes a shuffle into swapping the first element of the list with a random element of the remaining list and shuffling the new remaining list.

With a random-access implementation of a list – such as an array – all of the required operations are cheap and the resulting algorithm runs in linear time.

**lemma** shuffle-fisher-yates-step:

**assumes** xs-nonempty [simp]: xs \( \neq \) []

**shows** shuffle xs =

\[
\begin{align*}
&\text{do } (i \leftarrow \text{pmf-of-set} \left( \ldots \text{length } xs \right)); \\
&\quad \text{let } ys = \text{swap } xs \ 0 \ i; \\
&\quad zs \leftarrow \text{shuffle} \left( \text{tl } ys \right); \\
&\quad \text{return-pmf} \left( \text{hd } ys \# \ zs \right)
\end{align*}
\]

(\( \text{proof} \))
1.4 Forward Fisher-Yates Shuffle

The actual Fisher–Yates shuffle is now merely a kind of tail-recursive version of decomposition described above. Note that unlike the traditional Fisher–Yates shuffle, we shuffle the list from front to back, which is the more natural way to do it when working with linked lists.

```haskell
function fisher-yates-aux where
  fisher-yates-aux i xs = (if i + 1 ≥ length xs then return-pmf xs else
  do { j ← pmf-of-set {i..<length xs};
           fisher-yates-aux (i + 1) (swap xs i j) })
⟨proof⟩
termination ⟨proof⟩
```

```
declare fisher-yates-aux.simps [simp del]
```

```
lemma fisher-yates-aux-correct:
  fisher-yates-aux i xs = map-pmf (λys. take i xs @ ys) (shuffle (drop i xs))
⟨proof⟩
```

```
definition fisher-yates where
  fisher-yates = fisher-yates-aux 0
```

```
lemma fisher-yates-correct: fisher-yates xs = shuffle xs
⟨proof⟩
```

1.5 Backwards Fisher-Yates Shuffle

We can now easily derive the classical Fisher–Yates shuffle, which goes through the list from back to front and show its equivalence to the forward Fisher–Yates shuffle.

```haskell
fun fisher-yates-alt-aux where
  fisher-yates-alt-aux i xs = (if i = 0 then return-pmf xs else
  do { j ← pmf-of-set {...i};
           fisher-yates-alt-aux (i - 1) (swap xs i j) })
```

```
declare fisher-yates-alt-aux.simps [simp del]
```

```
lemma fisher-yates-alt-aux-altdef:
  i < length xs ⇒ fisher-yates-alt-aux i xs =
  map-pmf rev (fisher-yates-alt-aux (length xs - i - 1) (rev xs))
⟨proof⟩
```

```
definition fisher-yates-alt where
  fisher-yates-alt xs = fisher-yates-alt-aux (length xs - 1) xs
```

```
lemma fisher-yates-alt-aux-correct:
  fisher-yates-alt xs = shuffle xs
⟨proof⟩
```

4
1.6 Code generation test

Isabelle’s code generator allows us to produce executable code both for *shuffle* and for *fisher-yates* and *fisher-yates-alt*. However, this code does not produce a random sample (i.e. a single randomly permuted list) – which is, in fact, the only purpose of the Fisher–Yates algorithm – but the entire probability distribution consisting of $n!$ lists, each with probability $1/n!$.

In the future, it would be nice if Isabelle also had some code generation facility that supports generating sampling code.

```plaintext
value [code] shuffle "abed"
value [code] fisher-yates "abed"
value [code] fisher-yates-alt "abed"
end
```

References

