The Fisher–Yates shuffle

Manuel Eberl

April 19, 2020

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 4
   1.4 Forward Fisher-Yates Shuffle 5
   1.5 Backwards Fisher-Yates Shuffle 7
   1.6 Code generation test 9
1 Fisher–Yates shuffle

theory Fisher-Yates
  imports HOL Probability.Probability
begin

lemma integral-pmf-of-multiset:
  A ≠ (#) ⟹ (∫ 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)
by (subst integral-measure-pmf[where A = set-mset A])
  (simp-all add: sum-divide-distrib mult-ac)

lemma pmf-bind-pmf-of-multiset:
  A ≠ (#) ⟹ pmf (pmf-of-multiset A >>= f) y =
  (∑ x∈set-mset A. real (count A x) * pmf (f x) y) / real (size A)
by (simp add: pmf-bind integral-pmf-of-multiset)

lemma pmf-map-inj-inv:
  assumes inj-on f (set-pmf 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 (cases x ∈ f' set-pmf p)
  case True
  from this obtain y where y ∈ set-pmf p x = f y by blast
  with assms(1) have pmf (map-pmf f p) x = pmf p y
    by (simp add: pmf-map-inj)
  also from y assms(2)[of y] have y = f' x by simp
  finally show ?thesis using y by simp
next
  case False
  hence x /∈ set-pmf (map-pmf f p) by simp
  hence pmf (map-pmf f p) x = 0 by (simp add: set-pmf-eq)
  also from False have 0 = (if x ∈ range f then pmf p (f' x) else 0)
    by (auto simp: assms(2) set-pmf-eq)
  finally show ?thesis .
qed

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
by (simp add: swap-def)

lemma swap-eq-Nil-iff [simp]: swap xs i j = [] ⟷ xs = []
by (simp add: swap-def)

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)
by (auto simp: swap-def nth-list-update)

lemma map-swap: i < length xs ⇒ j < length xs ⇒ map f (swap xs i j) = swap (map f xs) i j
by (simp add: map-update map-nth)

lemma swap-swap: i < length xs ⇒ j < length xs ⇒ swap (swap xs i j) j i = xs
by (intro nth-equalityI) (auto simp: nth-list-update)

lemma mset-swap: i < length xs ⇒ j < length xs ⇒ mset (swap xs i j) = mset xs
by (simp add: mset-update swap-def nth-list-update)

lemma hd-swap-0: i < length xs ⇒ hd (swap xs 0 i) = xs!
i unfolding swap-def by (subst hd-conv-nth) | force+

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) ≠ {#}
shows pmf-of-set (permutations-of-multiset A) =
do {x ← pmf-of-multiset A; xs ← pmf-of-set (permutations-of-multiset (A − {#x#}));
return-pmf (x#xs)} (is ?lhs = ?rhs)
proof (rule pmf-eqI)
fix xs :: 'a list
show pmf ?lhs xs = pmf ?rhs xs
proof (cases xs ∈ permutations-of-multiset A)
case False
with assms have xs ∉ set-pmf ?lhs by simp
moreover from assms False have xs ∉ set-pmf ?rhs
by (auto simp: permutations-of-multiset-Cons-iff)
ultimately show ?thesis by (simp add: set-pmf-eq)
next
case True
with assms have nonempty: xs ≠ [] by (auto dest: permutations-of-multisetD)
hence range-Cons: xs ∈ range ((#) x) “→” hd xs = x for x
by (cases xs) auto
from True nonempty
have hd-tl: hd xs ∈# A ∨ tl xs ∈ permutations-of-multiset (A − {#hd xs#})
by (cases xs) (auto simp: permutations-of-multiset-Cons-iff)
from assms have pmf ?rhs xs =
\[
\sum_{x \in \text{set-mset} A} \text{real} \left( \text{count} A \, x \right) \cdot \text{pmf} \left( \text{map-pmf} \left( \left( \# \right) \, x \right) \right) \\
\left( \text{pmf-of-set} \left( \text{permutations-of-multiset} \left( A - \{\#x\#\} \right) \right) \right) \text{xs} \right) \bigg/ \text{real} \left( \text{size} A \right)
\]

(is - = ?S / -)

unfolding \text{map-pmf-def} \ [\text{symmetric}] \ by \ (\text{simp add: pmf-bind-pmf-of-multiset})

also have \ ?S =

\[
\sum_{x \in \text{set-mset} A} \text{if} \ x = \text{hd} \ \text{xs} \text{then} \ \text{real} \left( \text{count} A \, (\text{hd} \ \text{xs}) \right) \bigg/ \text{real} \left( \text{card} \left( \text{permutations-of-multiset} \left( A - \{\#\text{hd} \ \text{xs}\#\} \right) \right) \right) \text{else} \ 0
\]

using \text{range-Cons hd-tl}

by (intro \text{sum.cong refl, subst pmf-inj-inv[where f' = tl]) auto}

also have \ \ldots = \text{real} \left( \text{count} A \, (\text{hd} \ \text{xs}) \right) \bigg/ \text{real} \left( \text{card} \left( \text{permutations-of-multiset} \left( A - \{\#\text{hd} \ \text{xs}\#\} \right) \right) \right)

using \text{hd-tl by (simp add: sum.delta)}

also from \text{hd-tl have} \ \ldots = \text{real} \left( \text{size} A \right) \bigg/ \text{real} \left( \text{card} \left( \text{permutations-of-multiset} \left( A \right) \right) \right)

\text{xs}

using \text{assms True by simp}

finally show \ ?thesis ..

qed

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


lemma shuffle-empty [simp]: \text{shuffle} \ [] = \text{return-pmf} \ []

by (simp add: shuffle-def pmf-of-set-singleton)

lemma shuffle-singleton [simp]: \text{shuffle} \ [x] = \text{return-pmf} \ [x]

by (simp add: shuffle-def pmf-of-set-singleton)

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-gates-step:

assumes \text{xs-nonempty} [simp]: \text{xs} \ \neq \ []

shows \text{shuffle} \ \text{xs} =

\[
\begin{array}{l}
\text{do} \ \{ i \leftarrow \text{pmf-of-set} \left( \text{..<} \text{length} \ \text{xs} \right) ; \\
\text{let} \ \text{ys} = \text{swap} \ \text{xs} \ \theta \ i ; \\
\text{zs} \leftarrow \text{shuffle} \ (\text{tl} \ \text{ys}) ; \\
\}\end{array}
\]

\text{is} - = \ ?S / -

unfolding \text{map-pmf-def} \ [\text{symmetric}] \ by \ (\text{simp add: pmf-bind-pmf-of-multiset})

also have \ ?S =

\[
\sum_{x \in \text{set-mset} A} \text{if} \ x = \text{hd} \ \text{xs} \text{then} \ \text{real} \left( \text{count} A \, (\text{hd} \ \text{xs}) \right) \bigg/ \text{real} \left( \text{card} \left( \text{permutations-of-multiset} \left( A - \{\#\text{hd} \ \text{xs}\#\} \right) \right) \right) \text{else} \ 0
\]

using \text{range-Cons hd-tl}

by (intro \text{sum.cong refl, subst pmf-inj-inv[where f' = tl]) auto}

also have \ \ldots = \text{real} \left( \text{count} A \, (\text{hd} \ \text{xs}) \right) \bigg/ \text{real} \left( \text{card} \left( \text{permutations-of-multiset} \left( A - \{\#\text{hd} \ \text{xs}\#\} \right) \right) \right)

\text{xs}

using \text{assms True by simp}

finally show \ ?thesis ..

qed

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


lemma shuffle-empty [simp]: \text{shuffle} \ [] = \text{return-pmf} \ []

by (simp add: shuffle-def pmf-of-set-singleton)

lemma shuffle-singleton [simp]: \text{shuffle} \ [x] = \text{return-pmf} \ [x]

by (simp add: shuffle-def pmf-of-set-singleton)

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-gates-step:

assumes \text{xs-nonempty} [simp]: \text{xs} \ \neq \ []

shows \text{shuffle} \ \text{xs} =

\[
\begin{array}{l}
\text{do} \ \{ i \leftarrow \text{pmf-of-set} \left( \text{..<} \text{length} \ \text{xs} \right) ; \\
\text{let} \ \text{ys} = \text{swap} \ \text{xs} \ \theta \ i ; \\
\text{zs} \leftarrow \text{shuffle} \ (\text{tl} \ \text{ys}) ; \\
\}\end{array}
\]
proof –

have \( \text{shuffle} \, xs = \) do \( \{ x \leftarrow \text{pmf-of-multiset} \, (\text{mset} \, xs); \) \)
\( x_{} \leftarrow \text{pmf-of-set} \, (\text{permutations-of-multiset} \, (\text{mset} \, xs - \{ \#x\\})); \)
\( \text{return-pmf} \, (x \#xs) \) \}
\section*{unfolding shuffle-def}
\by (simp add: \text{pmf-of-set-permutations-of-multiset-nonempty})
\begin{align*}
\text{also have} & \quad \text{pmf-of-multiset} \, (\text{mset} \, xs) = \\
& \text{pmf-of-multiset} \, (\text{image-mset} \, (\text{map} \, (\text{mset} \, (\text{upt} \, (\text{length} \, xs)))))
\end{align*}
\by (subst \text{mset-map \text{[symmetric]}}) (simp add: \text{map-nth})
\begin{align*}
\text{also have} & \quad \ldots = \text{map-pmf} \, (\text{!!} \, xs) \, (\text{pmf-of-set} \, \{..<\text{length} \, xs\}) \\
& \text{by (subst \text{map-pmf-of-set}) (auto simp add: \text{map-pmf-of-set \text{atLeast0LessThan lessThan-empty-iff})}}
\end{align*}
\begin{align*}
\text{also have} & \quad \ldots = \text{do} \{ i \leftarrow \text{pmf-of-set} \, \{..<\text{length} \, xs\}; \\
& \quad ys \leftarrow \text{pmf-of-set} \, (\text{permutations-of-multiset} \, (\text{mset} \, xs - \{\#xs \, !i\})); \\
& \quad \text{return-pmf} \, (xs \, !i \# ys) \} \\
& \text{by (simp add: \text{map-pmf-def bind-assoc-pmf bind-return-pmf})}
\end{align*}
\begin{align*}
\text{also have} & \quad \ldots = \text{do} \{ i \leftarrow \text{pmf-of-set} \, \{..<\text{length} \, xs\}; \\
& \quad \text{let} \, ys = \text{swap} \, xs \, 0 \, i; \\
& \quad \quad zs \leftarrow \text{shuffle} \, (\text{tl} \, (\text{swap} \, xs \, 0 \, i)); \\
& \quad \quad \text{return-pmf} \, (\text{hd} \, ys \# zs) \}
\end{align*}
\section*{unfolding Let-def shuffle-def}
\by (intro \text{bind-pmf-cong refl, subst \text{asm set-pmf-of-set})}
\begin{align*}
& \text{(auto simp: \text{lessThan-empty-iff \text{mset-tl \text{mset-swap \text{hd-swap-0})}})}
\end{align*}
\begin{align*}
\text{finally show} & \quad \ldots \text{thesis by (simp add: \text{Let-def})}
\end{align*}
qed

\section{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.

\section*{function fisher-yates-aux where}

\begin{align*}
\text{fisher-yates-aux} \, i \, xs = & \quad \text{(if} \, i + 1 \geq \text{length} \, xs \, \text{then} \, \text{return-pmf} \, xs \, \text{else} \\
& \quad \text{do} \{ j \leftarrow \text{pmf-of-set} \, \{..<\text{length} \, xs\}; \\
& \quad \quad \text{fisher-yates-aux} \, (i + 1) \, (\text{swap} \, xs \, j) \} \}
\end{align*}
\by auto
\begin{align*}
\text{termination by} \quad (\text{relation Wellfounded.measure (}\lambda(i, xs). \, \text{length} \, xs - i)\, \text{simp-all)}
\end{align*}

\begin{align*}
\text{declare} & \quad \text{fisher-yates-aux.simps [simp del]} \end{align*}
lemma fisher-yates-aux-correct:

fisher-yates-aux i xs = map-pmf (λys. take i xs @ ys) (shuffle (drop i xs))

proof (induction i xs rule: fisher-yates-aux.induct)

case (1 i xs)

show ?case

proof (cases i + 1 ≥ length xs)

case True

show ?thesis

proof (cases i ≥ length xs)

case False

with True have length xs = Suc i and i: i = length xs - 1 by simp-all

hence xs ≠ [] by auto

hence xs = butlast xs @ [last xs] by (rule append-butlast-last-id [symmetric])

also have butlast xs = take i xs by (simp add: butlast-conv-take i)

finally have eq: take i xs @ [last xs] = xs ..

moreover have zi = take i xs @ drop i xs by simp

ultimately have take i xs @ [last xs] = take i xs @ drop i xs by (rule trans)

hence drop i xs = [last xs] by (subst (asm) same-append-eq) simp-all

with True show ?thesis by (simp add: eq fisher-yates-aux.simps)

qed (simp-all add: fisher-yates-aux.simps)

next

case False

from False have xs-nonempty [simp]: xs ≠ [] by auto

have fisher-yates-aux i xs =

  pmf-of-set {..<length xs} ≡ (λj. fisher-yates-aux (i+1) (swap xs i j))

  using False by (subst fisher-yates-aux.simps) simp

also have {..<length xs} ∈ (λj. j + i) ' {..<length xs - i} by (simp add: lessThan-atLeast0)

also from False have pmf-of-set ... = map-pmf (λj. j + i) (pmf-of-set {..<length xs - i})

  by (subst map-pmf-of-set-inj) (simp-all add: lessThan-empty-iff)

also from False have length xs - i = length (drop i xs) by simp

also have map-pmf (λj. j + i) (pmf-of-set {..<length (drop i xs)}) ≡

  (λj. fisher-yates-aux (i + 1) (swap xs i j)) =

  pmf-of-set {..<length (drop i xs)} ≡ (λj. fisher-yates-aux (i + 1) (swap xs i (j+i)))

  by (simp add: map-pmf-def bind-return-pmf bind-assoc-pmf)

also have ... = do {j ← pmf-of-set {..<length (drop i xs)};

let ys = swap (drop i xs) 0 j;

zs ← shuffle (tl ys);

return-pmf (take i xs @ hd ys # zs)} (is_ = bind-pmf - ?T)

proof (intro bind-pmf-cong refl)

fix j assume j ∈ set-pmf (pmf-of-set {..<length (drop i xs)})

with False have j: j < length (drop i xs) by (simp-all add: lessThan-empty-iff)

define ys where ys = swap xs i (j + i)

have fisher-yates-aux (i + 1) ys = map-pmf ((@) (take (i+1) ys)) (shuffle (drop (i+1) ys))

  using False unfolding ys-def by (intro 1.IH) simp-all

also from False have take (i+1) ys = take i ys @ [hd (drop i ys)]
by \((\text{simp add: ys-def take-hd-drop})\)
also have \(\text{drop } (i+1) \text{ ys } = \text{tl } (\text{drop } i \text{ ys})\) by \((\text{simp add: ys-def tl-drop drop-Suc})\)
also from \(\text{False } j\) have \(\text{drop } i \text{ ys } = \text{swap } (\text{drop } i \text{ xs}) 0\) \(j\) by \((\text{simp add: ys-def swap-def drop-update-swap add-ac})\)
also from \(\text{False } j\) have \(\text{take } i \text{ ys } = \text{take } i \text{ xs}\) by \((\text{simp add: ys-def swap-def})\)
finally show \(\text{fisher-yates-aux } (i + 1) \text{ ys } = ?T\) \(j\) by \((\text{simp add: ys-def map-pmf-def Let-def bind-assoc-pmf bind-return-pmf})\)
qed
also from \(\text{False}\) have \(\ldots = \text{map-pmf } (\lambdazs. \text{take } i \text{ xs } @ zs) (\text{shuffle } (\text{drop } i \text{ xs}))\) by \((\text{subst shuffle-fisher-yates-step[of drop i xs]})\)
finally show \(?\text{thesis}\.\)
qed

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

lemma fisher-yates-correct: fisher-yates \(xs\) = shuffle \(xs\)
unfolding fisher-yates-def
by \((\text{subst fisher-yates-aux-correct}) (\text{simp-all add: map-pmf-def Let-def bind-return-pmf})\)

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.

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

declare fisher-yates-alt-aux.simps [simp def]

lemma fisher-yates-alt-aux-altdef:\n\(i < \text{length } xs \implies \text{fisher-yates-alt-aux } i \text{ xs } = \text{map-pmf } \text{rev } (\text{fisher-yates-aux } (\text{length } xs - i - 1) \text{ rev } xs)\)
proof \((\text{induction } i \text{ xs rule: fisher-yates-alt-aux.induct})\)
case \((1 \ i \ xs)\)
show \(?\text{case}\)
proof \((\text{cases } i = 0)\)
case False
with \(1.\text{prems}\) have \(\text{map-pmf } \text{rev } (\text{fisher-yates-aux } (\text{length } xs - i - 1) \text{ rev } xs)\) = \(\text{pmf-of-set } \{\text{length } xs - \text{Suc } i..<\text{length } xs\} \gg\)
\(\lambda x. \text{fisher-yates-aux } (\text{Suc } (\text{length } xs - \text{Suc } i))\)
\(\text{swap } \text{rev } xs) \ (\text{length } xs - \text{Suc } i) \ x\) \(\gg\)
\(\lambda x. \text{return-pmf } (\text{rev } x))\)

7
by (subst fisher-yates-aux-simps) (auto simp: map-pmf-def bind-return-pmf
  bind-assoc-pmf)
also from 1.prems False
have bij: bij-betw (λj. length xs − Suc j) {..i} {length xs − Suc i..<length xs}
  by (intro bij-betwI [where g = λj. length xs − Suc j]) auto
from bij have {length xs − Suc i..<length xs} = (λj. length xs − Suc j) ·
  {..i}
  by (simp add: bij-betw-def)
also from bij have pmf-of-set ... = pmf-pmf (λj. length xs − Suc j) (pmf-of-set
  {..i})
  by (subst map-pmf-of-set-inj) (auto simp: bij-betw-def)
also have map-pmf (λj. length xs − Suc j) (pmf-of-set {..i}) ≫=
  (λx. fisher-yates-aux (Suc (length xs − Suc i))
  (swap (rev xs) (length xs − Suc i) x) ≫= (λx. return-pmf (rev x)))
  using 1.prems False
  by (auto simp add: map-pmf-def bind-assoc-pmf bind-return-pmf Suc-diff-Suc
    swap-def rev-update rev-nth intro!: bind-pmf-cong)
also have ... = pmf-of-set {..i} ≫= (λj. fisher-yates-alt-aux (i − 1) (swap xs
  i j))
  using 1.prems False 1.IH [symmetric] by (auto intro!: bind-pmf-cong)
also from 1.prems False have ... = fisher-yates-alt-aux i xs
  by (subst fisher-yates-alt-aux-altdef)
finally show ?thesis..
qed (insert 1.prems, simp-all add: fisher-yates-aux-simps fisher-yates-alt-aux-simps)

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 (cases xs = [])
  case True
  thus ?thesis
    by (simp add: fisher-yates-alt-def fisher-yates-alt-aux-simps)
next
  case False
  thus ?thesis unfolding fisher-yates-alt-def
    by (subst fisher-yates-alt-aux-altdef
      (simp-all add: fisher-yates-alt-aux-correct shuffle-def map-pmf-of-set-inj)
    )
qed
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.

value [code] shuffle "abcd"
value [code] fisher-yates "abcd"
value [code] fisher-yates-alt "abcd"

end

References

