The Fisher–Yates shuffle

Manuel Eberl

September 13, 2023

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

theory *Fisher-Yates*
imports *HOL−Probability.Probability*
begin

lemma *integral-pmf-of-multiset*:
$A \neq \{\#\} \Longrightarrow (\int x.\ (f\ x :: real)\ \partial measure\text{-}pmf\ (pmf\text{-}of\text{-}multiset\ A)) =$
$(\sum x \in set\text{-}mset\ A.\ of\text{-}nat\ (count\ A\ x) * f\ x)\ /\ of\text{-}nat\ (size\ A)$
by (*subst integral-measure-pmf*[**where** $A = set\text{-}mset\ A$])
(*simp-all add*: *sum-divide-distrib mult-ac*)

lemma *pmf-bind-pmf-of-multiset*:
$A \neq \{\#\} \Longrightarrow pmf\ (pmf\text{-}of\text{-}multiset\ A \gg= f)\ y =$
$(\sum x \in set\text{-}mset\ A.\ real\ (count\ A\ x) * pmf\ (f\ x)\ y)\ /\ real\ (size\ A)$
by (*simp add*: *pmf-bind integral-pmf-of-multiset*)

lemma *pmf-map-inj-inv*:
assumes *inj-on f* (*set-pmf p*)
assumes $\bigwedge x.\ f'\ (f\ x) = x$
shows $pmf\ (map\text{-}pmf\ f\ p)\ x = (if\ x \in range\ f\ then\ pmf\ p\ (f'\ x)\ else\ 0)$
proof (*cases* $x \in f$ ‘ *set-pmf p*)
case *True*
from *this* **obtain** *y* **where** *y*: $y \in set\text{-}pmf\ p\ x = f\ y$ **by** *blast*
with *assms*(*1*) **have** $pmf\ (map\text{-}pmf\ f\ p)\ x = pmf\ p\ y$
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 $x \notin set\text{-}pmf\ (map\text{-}pmf\ f\ p)$ **by** *simp*
hence $pmf\ (map\text{-}pmf\ f\ p)\ x = 0$ **by** (*simp add*: *set-pmf-eq*)
also from *False* **have** $0 = (if\ x \in range\ f\ then\ pmf\ p\ (f'\ x)\ else\ 0)$
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*]: $swap\ xs\ i\ j = [] \longleftrightarrow xs = []$
by (*simp add*: *swap-def*)

lemma *nth-swap*: $i < length\ xs \Longrightarrow j < length\ xs \Longrightarrow$

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*: *swap-def 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-swap 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*) (*subst nth-list-update* | *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* =

```
    (∑x∈set-mset A. real (count A x) * pmf (map-pmf ((#) x)
      (pmf-of-set (permutations-of-multiset (A - {#x#})))) xs) / real (size A)
(is - = ?S / -)
    unfolding map-pmf-def [symmetric] by (simp add: pmf-bind-pmf-of-multiset)
  also have ?S =
    (∑x∈set-mset A. if x = hd xs then real (count A (hd xs)) /
      real (card (permutations-of-multiset (A - {#hd xs#}))) else 0)
    using range-Cons hd-tl
    by (intro sum.cong refl, subst pmf-map-inj-inv[where f' = tl]) auto
  also have ... = real (count A (hd xs)) /
                  real (card (permutations-of-multiset (A - {#hd xs#})))
    using hd-tl by (simp add: sum.delta)
  also from hd-tl have ... = real (size A) / real (card (permutations-of-multiset
A))
     by (simp add: divide-simps real-card-permutations-of-multiset-remove[of hd
xs])
  also have  ... / real (size A) = pmf (pmf-of-set (permutations-of-multiset A))
xs
    using assms True by simp
  finally show ?thesis ..
 qed
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 ⇒ 'a list pmf where
  shuffle xs = pmf-of-set (permutations-of-multiset (mset xs))

lemma shuffle-empty [simp]: shuffle [] = return-pmf []
  by (simp add: shuffle-def pmf-of-set-singleton)

lemma shuffle-singleton [simp]: shuffle [x] = 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-yates-step:
  assumes xs-nonempty [simp]: xs ≠ []
  shows shuffle xs =
        do {i ← pmf-of-set {..<length xs};
            let ys = swap xs 0 i;
            zs ← shuffle (tl ys);
```

```
            return-pmf (hd ys # zs)
          }
proof −
  have shuffle xs = do {x ← pmf-of-multiset (mset xs);
          xs ← pmf-of-set (permutations-of-multiset (mset xs − {#x#}));
          return-pmf (x#xs)
        } unfolding shuffle-def
    by (simp add: pmf-of-set-permutations-of-multiset-nonempty)
  also have pmf-of-multiset (mset xs) =
          pmf-of-multiset (image-mset ((!) xs) (mset (upt 0 (length xs))))
    by (subst mset-map [symmetric]) (simp add: map-nth)
  also have ... = map-pmf ((!) xs) (pmf-of-set {..<length xs})
    by (subst map-pmf-of-set) (auto simp add: map-pmf-of-set atLeast0LessThan
lessThan-empty-iff)
  also have do {x ← map-pmf ((!) xs) (pmf-of-set {..<length xs});
              ys ← pmf-of-set (permutations-of-multiset (mset xs − {#x#}));
              return-pmf (x # ys)
            } =
          do {i ← pmf-of-set {..<length xs};
              ys ← pmf-of-set (permutations-of-multiset (mset xs − {#xs ! i#}));
              return-pmf (xs ! i # ys)
            }
    by (simp add: map-pmf-def bind-assoc-pmf bind-return-pmf)
  also have ... = do {i ← pmf-of-set {..<length xs};
                  let ys = swap xs 0 i;
                  zs ← shuffle (tl (swap xs 0 i));
                  return-pmf (hd ys # zs)
                } unfolding Let-def shuffle-def
    by (intro bind-pmf-cong refl, subst (asm) set-pmf-of-set)
      (auto simp: lessThan-empty-iff mset-tl mset-swap hd-swap-0)
  finally show ?thesis by (simp add: Let-def)
qed
```

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.

```
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)})
by auto
termination by (relation Wellfounded.measure (λ(i,xs). length xs − i)) simp-all

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 (*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 *xs = 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 {*i..<length xs*} *≫=* (*λj. fisher-yates-aux* (*i+1*) (*swap xs i j*))
using *False* **by** (*subst fisher-yates-aux.simps*) *simp*
also have {*i..<length xs*} = ((*λj. j + i*) ‘ {*..<length xs − i*})
using *False* **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 j* **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 (simp add: ys-def take-hd-drop)
    also have drop (i+1) ys = tl (drop i ys) by (simp add: ys-def tl-drop drop-Suc)
     also from False j have drop i ys = swap (drop i xs) 0 j
      by (simp add: ys-def swap-def drop-update-swap add-ac)
     also from False j have take i ys = take i xs
      by (simp add: ys-def swap-def)
     finally show fisher-yates-aux (i + 1) ys = ?T j
      by (simp add: ys-def map-pmf-def Let-def bind-assoc-pmf bind-return-pmf)
   qed
   also from False have ... = map-pmf (λzs. take i xs @ zs) (shuffle (drop i xs))
    by (subst shuffle-fisher-yates-step[of drop i xs])
      (simp-all add: map-pmf-def Let-def bind-return-pmf bind-assoc-pmf)
   finally show ?thesis .
  qed
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 (subst fisher-yates-aux-correct) (simp-all add: map-pmf-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 = (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-aux (length xs − i − 1) (rev xs))
proof (induction i xs rule: fisher-yates-alt-aux.induct)
  case (1 i xs)
  show ?case
  proof (cases i = 0)
    case False
   with 1.prems have map-pmf rev (fisher-yates-aux (length xs − i − 1) (rev xs))
=
      pmf-of-set {length xs − Suc i..<length xs} ⤜
       (λx. fisher-yates-aux (Suc (length xs − Suc i))
        (swap (rev xs) (length xs − Suc i) x) ⤜
         (λx. return-pmf (rev x)))
```

```
      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 ... = map-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)))
=
              pmf-of-set {..i} ⤜ (λx. map-pmf rev (
                fisher-yates-aux (length xs − i) (rev (swap xs i 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.simps[of i]) simp-all
    finally show ?thesis ..
  qed (insert 1.prems, simp-all add: fisher-yates-aux.simps fisher-yates-alt-aux.simps)
qed

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-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

[1] R. A. Fisher and F. Yates. *Statistical tables for biological, agricultural and medical research*, pages 26–27. Oliver & Boyd, Third edition, 1948.

[2] D. E. Knuth. In *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*. Addison-Wesley Longman Publishing Co., Inc., Third edition, 1997.

[3] Wikipedia. Fisher–Yates shuffle – Wikipedia, the free encyclopedia, 2016. [Online; accessed 5 October 2016].