Verifying the Fisher-Yates Shuffle Algorithm in Dafny

Stefan Zetzsche Jean-Baptiste Tristan Tancrède Lepoint
Mikael Mayer

Amazon Web Services

January 19, 2025

Introduction

The Fisher-Yates Shuffle

x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9

The Fisher-Yates Shuffle

x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9

0
1
2
3
4
5
6
7
8
9

The Fisher-Yates Shuffle

x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9

0
1
2
3
4
5
6
7
8
9

The Fisher-Yates Shuffle

x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9

0
1
2
3
4
5
6
7
8
9

The Fisher-Yates Shuffle

x_7	x_1	x_2	x_3	x_4	x_5	x_6	x_0	x_8	x_9

0
1
2
3
4
5
6
7
8
9

The Fisher-Yates Shuffle

x_7	x_1	x_2	x_3	x_4	x_5	x_6	x_0	x_8	x_9

1
2
3
4
5
6
7
8
9

The Fisher-Yates Shuffle

x_7	x_1	x_2	x_3	x_4	x_5	x_6	x_0	x_8	x_9

1
2
3
4
5
6
7
8
9

The Fisher-Yates Shuffle

x_7	x_1	x_2	x_3	x_4	x_5	x_6	x_0	x_8	x_9

1
2
3
4
5
6
7
8
9

The Fisher-Yates Shuffle

x_7	x_5	x_2	x_3	x_4	x_1	x_6	x_0	x_8	x_9

1
2
3
4
5
6
7
8
9

The Fisher-Yates Shuffle

x_7	x_5	x_2	x_3	x_4	x_1	x_6	x_0	x_8	x_9

2
3
4
5
6
7
8
9

The Fisher-Yates Shuffle

...

The Fisher-Yates Shuffle

x_7	x_5	x_9	x_0	x_1	x_4	x_3	x_6	x_8	x_2

8
9

The Fisher-Yates Shuffle

x_7	x_5	x_9	x_0	x_1	x_4	x_3	x_6	x_8	x_2

8
9

The Fisher-Yates Shuffle

x_7	x_5	x_9	x_0	x_1	x_4	x_3	x_6	x_8	x_2

8
9

The Fisher-Yates Shuffle

x_7	x_5	x_9	x_0	x_1	x_4	x_3	x_6	x_2	x_8

8
9

The Fisher-Yates Shuffle

x_7	x_5	x_9	x_0	x_1	x_4	x_3	x_6	x_2	x_8

9

The Fisher-Yates Shuffle

x_7	x_5	x_9	x_0	x_1	x_4	x_3	x_6	x_2	x_8

9

The Fisher-Yates Shuffle

x_7	x_5	x_9	x_0	x_1	x_4	x_3	x_6	x_2	x_8

The Fisher-Yates Shuffle

x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9

x_7	x_5	x_9	x_0	x_1	x_4	x_3	x_6	x_2	x_8

Formalizing Randomness

We model randomness as countably infinite stream of independent and identically distributed fair random bits of type $\{0,1\}^{\mathbb{N}}$ or `type Bitstream = nat -> bool` in Dafny.

A transformation $T : \{0,1\}^{\mathbb{N}} \to V$ is correct if for all samples $x \in V$, it holds

$$\mu(T^{-1}(\{x\})) = \Pr[X = x]$$

where X is a random variable with distribution D and μ is a probability measure on bitstreams.

Formalizing Randomness

For example, a coin flip can be expressed as:

```
function Coin'(s: Bitstream): bool {
  s(0)
}
```

Under above view, its correctness translates to the equalities

$$\mu(\{s \in \{0,1\}^{\mathbb{N}} \mid s(0) = b\}) = 0.5,$$

where $b \in \{0,1\}$.

An Overview of Our Approach

- A functional model
  - Operates on sequences via the bitstream transformer approach
- A correctness proof for the functional model
  - Establishes that it has the desired distribution
- An executable imperative implementation
  - Operates on arrays by sampling from external random source
  - Proven equivalent to the functional model

A Functional Model

The Hurd Monad

```
datatype Result<T> = Result(value: T, rest: Bitstream)

type Hurd<T> = Bitstream -> Result<T>

function Return<T>(x: T): Hurd<T> {
  (s: Bitstream) => Result(x, s)
}

function Bind<S, T>(h: Hurd<S>, f: S -> Hurd<T>): Hurd<T> {
  (s: Bitstream) =>
    var (x, s') := h(s).Extract();
    f(x)(s')
}
```

Access to randomness can be formalized as the state monad of type `Bitstream`, which we call the *Hurd monad*.

The Hurd Monad

This way, the `Coin'` function extends to:

```
function Coin(): Hurd<bool> {
  (s: Bitstream) => Result(s(0), (n: nat) => s(n + 1))
}
```

More generally, taking an input of type `S` and returning a sample of type `T` can be modelled as:

```
function Sample<S,T>(x: S): Hurd<T>
```

A Probability Space on Bitstreams

We axiomatize the existence of a probability space on bitstreams

```
ghost const eventSpace: iset<iset<Bitstream>>
ghost const prob: iset<Bitstream> -> real

lemma {:axiom} ProbIsProbabilityMeasure()
  ensures IsProbability(eventSpace, prob)
```

and consequently the existence of an uniform sampler

```
ghost function {:axiom} Sample(n: nat): (h: Hurd<nat>)
  requires 0 < n
  ensures forall s :: 0 <= h(s).value < n
  ensures forall i | 0 <= i < n ::
            var e := iset s | h(s).value == i;
            && e in eventSpace
            && prob(e) == 1.0 / (n as real)
```

We suppress independence and measurability assumptions for readability.

A Recursive Model

With the previous machinery, we can introduce a purely functional implementation of Fisher-Yates:

```
ghost function Shuffle<T>(xs: seq<T>, i: nat := 0): Hurd<seq<T>>
  requires i <= |xs|
{
  (s: Bitstream) => ShuffleCurried(xs, s, i)
}

ghost function ShuffleCurried<T>
(xs: seq<T>, s: Bitstream, i: nat := 0): Result<seq<T>>
  requires i <= |xs|
  decreases |xs| - i
{
  if |xs| > 1 + i then
    var (j, s') := IntervalSample(i, |xs|)(s).Extract();
    var ys := Swap(xs, i, j);
    ShuffleCurried(ys, s', i + 1)
  else
    Return(xs)(s)
}
```

A Correctness Proof

Specifying Correctness

We specify the correctness of `Shuffle` as the following lemma:

```
lemma Correctness<T(!new)>(xs: seq<T>, p: seq<T>)
  requires forall a, b | 0 <= a < b < |xs| :: xs[a] != xs[b]
  requires multiset(p) == multiset(xs)
  ensures
    var e := iset s | Shuffle(xs)(s).value == p;
    && e in eventSpace
    && prob(e) == 1.0 / (Factorial(|xs|) as real)
```

Proving Correctness

Instead of attempting at a direct proof of `Correctness`, we establish a slightly more general variant of it:

```
lemma CorrectnessGeneral<T(!new)>(xs:seq<T>, p: seq<T>, i: nat)
  decreases |xs| - i
  requires i <= |xs| && |xs| == |p|
  requires forall a, b | i <= a < b < |xs| :: xs[a] != xs[b]
  requires multiset(p[i..]) == multiset(xs[i..])
  ensures
    var e := iset s | Shuffle(xs, i)(s).value[i..] == p[i..]
    && e in eventSpace
    && prob(e) == 1.0 / (Factorial(|xs|-i) as real)
```

Proving Correctness

At its core, the proof of `CorrectnessGeneral` makes use of two properties of `Sample` – *weak (functional) independence* and *measure-preservation*:

```
ghost function {:axiom} Sample(n: nat): (h: Hurd<nat>)
  requires 0 < n
  ensures IsIndepFunction(h)
  ensures IsMeasurePreserving(eventSpace, prob, eventSpace,
                              prob, s => h(s).rest)
  ensures forall s :: 0 <= h(s).value < n
  ensures forall i | 0 <= i < n ::
            var e := iset s | h(s).value == i;
            && e in eventSpace
            && prob(e) == 1.0 / (n as real)
```

An Imperative Implementation

External Randomness

A functional model like `Shuffle` is well suited for mathematical reasoning about its correctness properties.

In practice, however, a probabilistic program will use a real-world source of random bits rather than an abstract infinite stream of random bits.

We utilise an external source of uniformly distributed random numbers `Sample` that we assume to be correct in the sense that it is an instance of the functional model `Model.Sample`:

```
method Sample(n: nat) returns (i: nat)
  requires ...
  modifies 's
  ensures Model.Sample(n)(old(s)) == Result(i, s)
```

An Executable Implementation

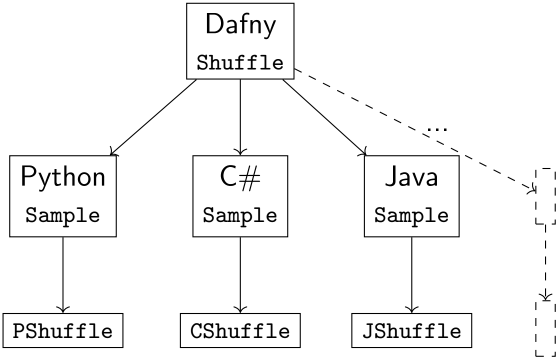
The equivalence between `Model.Sample` and `Sample` can be lifted to an equivalence between the functional model `Model.Shuffle` and the executable implementation `Shuffle` of Fisher-Yates below:

```
method Shuffle<T>(a: array<T>)
 decreases *
 modifies `s, a
 ensures Model.Shuffle(old(a[..]))(old(s)) == Result(a[..], s)
{
  if a.Length > 1 {
    for i := 0 to a.Length - 1 {
      var j := IntervalSample(i, a.Length);
      Swap(a, i, j);
    }
  }
}
```

The proof of equivalence requires an appropriate for-loop invariant and assertions.

Target Language Implementation

The compilation of `Shuffle` to a target language is possible if it implements `Sample` and is supported by the Dafny compiler:



Summary

Final Remarks

- We axiomatized the lemma `ProbIsProbabilityMeasure` and assumed the existence of the function `Model.Sample` and an external method `Sample` that implements it.
- Previous work has shown that it is also possible to instead assume the existence of the function `Model.Coin` and lift it to `Model.Sample`. For simplicity and efficiency, we drew the line of axiomatization a bit higher.
- To compile to Java's `int` instead of `BigInteger`, we actually use Dafny's bounded `int32` instead of the unbounded `int`. There was almost no proof overhead caused by this complication.

Thank You

https://github.com/dafny-lang/Dafny-VMC
https://arxiv.org/abs/2501.06084