`trait SampleBernoulli`

Vicki Xu, Hanwen Zhang, Zachary Ratliff

February 7, 2024

This proof resides in **"contrib"** because it has not completed the vetting process.

Warning 1 (Code is not constant-time). `sample_bernoulli` takes in a boolean `constant_time` parameter to protect against timing attacks on the Bernoulli sampling procedure. However, the current implementation does not guard against other types of timing side-channels that can break differential privacy, e.g., non-constant time code execution due to branching.

PR History

- Pull Request #473

This document proves that the implementations of `SampleBernoulli` in `mod.rs` at commit f5bb719 (outdated[1]) satisfy the definition of the `SampleBernoulli` trait.

Definition 0.1. The `SampleBernoulli<T>` trait defines a function `sample_bernoulli`, where the data type of the probability is `T`.

For any setting of the input parameters `prob` of type `T` restricted to $[0, 1]$, and `constant_time` of type `bool`, `sample_bernoulli` either

- raises an exception if there is a lack of system entropy or `constant_time` is not supported,
- returns `out` where `out` is $\top$ with probability `prob`, otherwise $\bot$.

If `constant_time` is set, the implementation's runtime is constant.

There are two `impl`'s (implementations): one for float probabilities, and one for rational probabilities. To prove correctness of each `impl`, we prove correctness of the implementation of `sample_bernoulli`.

Contents

[1] See new changes with `git diff f5bb719..2fc0be0 rust/src/traits/samplers/bernoulli/mod.rs`

1 impl for Float Probability

This corresponds to impl<T> SampleBernoulli<T> for bool where T: Float in Rust. At a high level, sample_bernoulli considers the binary expansion of prob into an infinite sequence a_i, like so: prob $= \sum_{i=0}^{\infty} \frac{a_i}{2^{i+1}}$. The algorithm samples $I \sim \mathit{Geom}(0.5)$ using an internal function sample_geometric_buffer, then returns a_I.

1.1 Hoare Triple

Preconditions

- **User-specified types:**
  - Variable prob must be of type T
  - Variable constant_time must be of type bool
  - Type T has trait Float. Float implies there exists an associated type T::Bits (defined in FloatBits) that captures the underlying bit representation of T.
  - Type T::Bits has traits PartialOrd and ExactIntCast<usize>
  - Type usize has trait ExactIntCast<T::Bits>

Pseudocode

```
# returns a single bit with some probability of success
def sample_bernoulli(prob : T, constant_time : bool) -> bool:
    if prob == 1:
        return True

    # prepare for sampling first heads index by coin flipping
    max_coin_flips = usize.exact_int_cast(T::EXPONENT_BIAS) + \
                     usize.exact_int_cast(T::MANTISSA_BITS)

    # find number of bits to sample, rounding up to nearest byte (smallest sample size)
    buffer_len = max_count_flips.inf_div(8)

    # repeatedly flip fair coin and identify 0-based index of first heads
    first_heads_index = sample_geometric_buffer(buffer_len, constant_time)

    # if no events occurred, return early
    if first_heads_index is None:
        return False

    # find number of zeroes in binary rep. of prob
    leading_zeroes = T::EXPONENT_BIAS - 1 - prob.raw_exponent()

    # case 1: index into the leading zeroes
    if first_heads_index < leading_zeros:
        return False

    # case 2: index into implicit bit directly to left of mantissa
    if first_heads_index == leading_zeroes:
        return prob.raw_exponent() != 0

    # case 3: index into out-of-bounds/implicitly-zero bits
    if first_heads_index > leading_zeroes + MANTISSA_BITS:
        return False

    # case 4: index into mantissa
    mask = (1 << (T::MANTISSA_BITS + leading_zeroes - first_heads_index))
    return (prob.to_bits() & mask) != 0
```

Postcondition

The postcondition is supplied by 0.1.

1.2 Proof

Proof. To show the correctness of `sample_bernoulli` we observe first that the base-2 representation of `prob` is of the form

`leading_zeroes || implicit_bit || mantissa || trailing_zeroes`

and is represented *exactly* as a normal floating-point number. The IEEE-754 standard represents a normal floating-point number using an exponent E, and a mantissa m, using a base-2 analog of scientific notation.

Definition 1.1 (Floating-Point Number)**.** A (k,ℓ)-bit floating-point number z is represented as

$$z = (-1)^s \cdot (B.M) \cdot (2^E)$$

where

- s is used to represent the *sign* of z
- B is the implicit bit; 1 for normal floating-point numbers and 0 for subnormal floating point numbers
- $M \in \{0,1\}^k$ is a k-bit string representing the part of the mantissa to the right of the radix point, i.e.,

$$1.M = \sum_{i=1}^{k} M_i 2^{-i}$$

- $E \in \mathbb{Z}$ represents the *exponent* of z. When ℓ bits are allocated to representing E, then $E \in [-(2^{\ell-1} - 2), 2^{\ell-1}] \cap \mathbb{Z}$. Note that the range of E is $2^\ell - 2$ rather than 2^ℓ as the remaining to numbers are used to represent special floating point values. When $E = -(2^{\ell-1} - 2)$, then the floating point number is considered *subnormal.*

We now use the technique for arbitrarily biasing a coin in 2 expected tosses as a building block. Recall that we can represent the probability `prob` as `prob` $= \sum_{i=0}^{\infty} \frac{a_i}{2^{i+1}}$ for $a_i \in \{0,1\}$, where a_i is the zero-indexed i-th significant bit in the binary expansion of `prob`. Then let $I \sim Geom(0.5)$ and observe that the random variable a_I is an exact Bernoulli sample with probability `prob` since $P(a_I = 1) = \sum_{i=0}^{\infty} P(a_i = 1 | I = i) P(I = i) = \sum_{i=1}^{\infty} a_i \cdot \frac{1}{2^{i+1}} =$ `prob`. It is therefore sufficient to show that for any (k,ℓ)-bit float `prob` $= \sum_{i=0}^{\infty} \frac{a_i}{2^{i+1}}$, `sample_bernoulli` returns the value a_I with $I \sim Geom(0.5)$.

First, we observe that by line 3, if `prob` $= 1.0$ then `sample_bernoulli` returns `true` which is correct by definition of a Bernoulli random variable. Otherwise, the variable `max_coin_flips` is computed to be the value `T::EXPONENT_BIAS` $+$ `T::MANTISSA_BITS` which equals $2^{\ell-1} - 1 + k$ for any (k,ℓ)-bit float. Since `prob` has finite precision, there is some j for which $a_i = 0$ for all $i > j$. For all (k,ℓ)-bit floating-point numbers, $j \leq 2^{\ell-1} - 1 + k$ by definition. Then `sample_bernoulli` calls `sample_geometric_buffer` with a buffer of length $\lceil \frac{\texttt{max\_coin\_flips}}{8} \rceil$ bytes (as shown in lines 8 and 11) which returns `None` if and only if $I > 8 \cdot \lceil \frac{2^{\ell-1}-1+k}{8} \rceil$, where $I \sim Geom(0.5)$ (by Theorem 2.1). In this case, since $I > j$ this index appears in the `trailing_zeroes` part of the binary expansion of `prob` and should always return `false`, i.e., $a_I = 0$ for all $I > j$. We can therefore restrict our attention to when `sample_geometric_buffer` returns an index $I \leq$ `max_coin_flips` and show that `sample_bernoulli` always returns a_I.

Assuming that `sample_geometric_buffer` returns some $I < j$, `sample_bernoulli` computes the number of leading zeroes in the binary expansion of `prob` to be `leading_zeroes` $=$ `T::EXPONENT_BIAS` $- 1 -$ `raw_exponent(prob)`, where `raw_exponent(prob)` is the value stored in the ℓ bits of the exponent. This value is correct by the specification of a (k,ℓ)-bit float. `sample_bernoulli` then matches on the value `first_heads_index` corresponding to $I \sim \mathit{Geom}(0.5)$ returned by the function `sample_geometric_buffer`:

Case 1 (`first_heads_index` < `leading_zeroes`).
This corresponds to `sample_geometric_buffer` returning a value I such that a_I indexes into the `leading_zeroes` part of the `prob` variable's binary expansion. Therefore, for any $I <$ `leading_zeroes`, it follows that $a_I = 0$ and we should return `false`. In this case, `sample_bernoulli` returns `false`.

Case 2 (`first_heads_index` == `leading_zeroes`).
This corresponds to `sample_geometric_buffer` returning a value I such that a_I indexes into the `implicit_bit` part of the `prob` variable's binary expansion. When `prob` is a normal floating point value, i.e., $E \neq -(2^{\ell-1}-2)$ then the implicit bit $a_I = 1$. Otherwise, when `prob` is a subnormal floating point value, i.e., $E = -(2^{\ell-1}-2)$, the implicit bit $a_I = 0$. Since `raw_exponent(prob)` corresponds to the exponent E for any (k, ℓ)-bit floating point number `prob`, `sample_bernoulli` returns `true` when `raw_exponent(prob)` $\neq 0$ and `false` otherwise.

Case 3 (`leading_zeroes`+`T::MANTISSA_BITS` $< I$). This corresponds to the case where `sample_geometric_buffer` returns a value I where $I > j$, but $I <$ `max_coin_flips` and therefore a_I indexes into the trailing zeroes. In this case, `sample_bernoulli` returns `false` since $a_I = 0$ for all bits in the `trailing_zeroes` part of `prob`'s binary expansion.

Case 4 (`leading_zeroes` < `first_heads_index` < `leading_zeroes` + `T::MANTISSA_BITS`).
This corresponds to `sample_geometric_buffer` returning a value I such that a_I indexes into the `mantissa` part of the `prob` variable's binary expansion. In this case, `sample_bernoulli` left-shifts the value `1` by (`MANTISSA_BITS + leading_zeroes - first_heads_index`) digits, the index into the mantissa corresponding to the digit a_I in the binary representation of `prob`. Since the operation between the left-shifted `1` and the binary representation of `prob` at that position is a bitwise AND, if the bit in question is 1 (matching the left-shifted `1`), `sample_bernoulli` will return `true`. Otherwise, `sample_bernoulli` will return `false`.

Therefore, for any value of `prob`, the function `sample_bernoulli` either raises an exception or returns the value `true` with probability exactly `prob`. □

2 `impl` for Rational Probability

This corresponds to `impl SampleBernoulli<Rational> for bool` in Rust.

2.1 Hoare Triple

Preconditions

- **User-specified types:**
  - Variable `prob` must be of type `Rational`
  - Variable `constant_time` must be of type `bool`

Pseudocode

```
# returns a single bit with some probability of success
def sample_bernoulli(prob: Rational, constant_time : bool) -> bool:
    if constant_time:
        raise NotImplementedError("constant-time uniform sampling of rationals is not implemented")

    let (numer, denom) = prob.into_numer_denom();
    return numer > Integer.sample_uniform_int_below(denom)
```

Postcondition

The postcondition is supplied by 0.1.

2.2 Proof

This proof has not been written.