Privacy Proofs for OpenDP: MakeCount

Connor Wagaman – wagaman@college.harvard.edu

September 4, 2021

Contents

Note: This proof document incorporates all suggestions made as of Sept. 4, 2021. The only "todos" that remain are those that concern inserting versioned links to the definitions documents. These documents will hopefully be pushed within the next two days, after which I will add the versioned links. There is also one question about the proof of lemma 2.2 (this lemma is new); the question appears immediately after the proof of the lemma.

1 MakeCount

1.1 Version of definitions docs

This document uses the version of the definitions docs published as of September 3, 2021. These documents are available at

TODO: Insert URL to version of definitions docs.

1.2 Implementation of MakeCount in Rust

In OpenDP (Rust), this is called `make_count`. See https://github.com/opendp/opendp/blob/main/rust/opendp/src/trans/count.rs.

This proof is based on the code in https://github.com/opendp/opendp/blob/c3b5c3bd9fc50c556362b628f08c5fddea069b4d/rust/opendp/src/trans/count.rs#L14-L27 from 12 July 2021. (It is from this pull request.) The Rust code can also be seen below.

```rust
pub fn make_count<TIA, TO>(
) -> Fallible<Transformation<VectorDomain<AllDomain<TIA>>, AllDomain<TO>, SymmetricDistance, AbsoluteDistance<TO>>>

    where TO: ExactIntCast<usize> + One + DistanceConstant<IntDistance>, IntDistance: InfCast<TO> {

    Ok(Transformation::new(
        VectorDomain::new_all(),
        AllDomain::new(),
        // think of this as: min(arg.len(), TO::MAX_CONSECUTIVE)
        Function::new(move |arg: &Vec<TIA>|
            TO::exact_int_cast(arg.len()).unwrap_or(TO::MAX_CONSECUTIVE)),
        SymmetricDistance::default(),
        AbsoluteDistance::default(),
        StabilityRelation::new_from_constant(TO::one())))
}
```

1.3 Implementation of MakeCount in Python-style pseudocode, with preconditions

We now use Python-style pseudocode to present a representation of the Rust function.

Recall that functions in the pseudocode are defined in the document "List of definitions used in the pseudocode".

TODO: provide versioned link to the "List of definitions used in the pseudocode"

The use of `code`-style parameters in the preconditions section below (for example, `input_domain`) means that this information should be passed along to the `Transformation` constructor.

Here, we use preconditions to check for traits, and to specify the domains and metrics.

Preconditions

- **User-specified types:** The `make_count` function takes two inputs: a generic input type `TIA` for the `Transformation` (meaning that the input vector to `Transformation` is of type `Vec(TIA)`), and a generic output type `TO` for the `Transformation`.
  - `TO` has traits `One`, `ExactIntCast(usize)`, and `DistanceConstant(IntDistance)`.

    Examples: `u32` and `i64` have these traits. On the other hand, `f32` does not because there are some values `v` of type `usize` that do not have an exact 32-bit floating point representation.
  - `IntDistance` has trait `InfCast(TO)`. (Note that this bullet point is not needed in this proof, but it is needed in the code so a hint can be constructed; otherwise a binary search would be needed to construct the hint.)

    Examples: Recall that `IntDistance` is an alias for the type `u32`. `IntDistance` has trait `InfCast(u32)`, because any unsigned 32-bit integer can be converted to an unsigned 32-bit integer that is at least as big. On the other hand, `IntDistance` does not have trait `InfCast(u64)`, because there are some unsigned 64-bit integers that do not have representations as unsigned 32-bit integers.

Postconditions: a valid `Transformation` must be returned (i.e. if a `Transformation` cannot be returned successfully, a runtime error should be returned)

```
def MakeCount(TIA, TO):

    input_domain = VectorDomain(AllDomain(TIA))
    output_domain = AllDomain(TO)
    input_metric = SymmetricDistance()
    output_metric = AbsoluteDistance(TO)

    # give the Transformation the following properties
    max_value = get_max_consecutive_int(TO)
    def function(data: Vec<TIA>) -> TO:
        try:
            return exact_int_cast(len(data), TO)
        except FailedCast:
            return max_value
    def stability_relation(din:IntDistance, dout:TO) -> bool:
        return 1 * inf_cast(din,TO) <= dout

    # now, return the Transformation
    return Transformation(input_domain,output_domain,function,
   input_metric,output_metric,stability_relation)
```

2 Proofs for the pseudocode

Theorem 2.1. *For every setting of the input parameters `TIA`, `TO` for `MakeCount` such that the given preconditions hold, `MakeCount` raises an exception (at compile time or run time) or returns a valid `Transformation` with the following properties:*

1. (Appropriate output domain). *For every vector `v` in the `input_domain`, `function(v)` is in the `output_domain`.*
2. (Domain-metric compatibility). *The domain `input_domain` matches one of the possible domains listed in the definition of `input_metric`, and likewise `output_domain` matches one of the possible domains listed in the definition of `output_metric`.*
3. (Stability guarantee). *For every input `u`, `v` drawn from the `input_domain` and for every pair $(\mathrm{d_{in}}, \mathrm{d_{out}})$, where $\mathrm{d_{in}}$ is of type `u32` and $\mathrm{d_{out}}$ is of type `TO` (see line 15 of the pseudocode), if `u`, `v` are $\mathrm{d_{in}}$-close under the `input_metric` and `stability_relation(din, dout) = True`, then `function(u)`, `function(v)` are $\mathrm{d_{out}}$-close under the `output_metric`.*

Proof. **(Part 1 – appropriate output domain).** In line 4 of the pseudocode, we have `output_domain = AllDomain(TO)`, so every value of type `TO` is in the `output_domain`, and in line 10 of the Python-style pseudocode, we see that the `function` is always guaranteed to return a value of type `TO`. Because Rust employs "type checking", if the Rust code compiles correctly, then the type correctness follows from the definition of the type signature enforced by Rust. Otherwise, the code raises an exception for incorrect input type. Therefore, since our output domain is any value of type `TO`, we see that `function` has the appropriate output domain `output_domain`.

□

Proof. **(Part 2 – domain-metric compatibility).**

The `input_domain` is `VectorDomain(AllDomain(TIA))`. Because our `input_metric` of `SymmetricDistance` is compatible with any domain of the form `VectorDomain(inner_domain)`, and because `VectorDomain(AllDomain(TIA))` is of this form, we see that it is compatible with our `input_metric` of `SymmetricDistance`.

The `output_domain` is `AllDomain(TO)`. Because our `output_metric` of `AbsoluteDistance(TO)` is compatible with any domain of the form `AllDomain(T)` where `T` has the trait `Sub(Output=T)`, and because `AllDomain(TO)` is of this form and `TO` has the necessary trait, we see that it is compatible with our `output_metric` of `AbsoluteDistance(TO)`.

□

Proof. **(Part 3 – stability relation).** We consider two inputs: a vector `u` of elements of type `TIA`; and a vector `v` of elements of type `TIA`. (This `input_domain` is specified in the pseudocode in section 1.3.)

Assume it is the case that `stability_relation`($\mathrm{d_{in}}, \mathrm{d_{out}}$) = `True`. From the stability relation provided on line 16, this means that `inf_cast`($\mathrm{d_{in}}$, `TO`) $\leq \mathrm{d_{out}}$. From the pseudocode definitions file at

TODO: Add versioned link to pseudocode definitions file

we know that `inf_cast` will cast $\mathrm{d_{in}}$ to a value at least as large as $\mathrm{d_{in}}$, so this assumption that `stability_relation` is `True` also means that $\mathrm{d_{in}} \leq \mathrm{d_{out}}$. Also assume that `v`, `w` are $\mathrm{d_{in}}$-close under the symmetric distance metric (in accordance with the `input_metric` specified in the preconditions in section 1.3).

We now refer to the definition of symmetric distance provided in the proof definitions document;

TODO: Add versioned link to proof defns doc

the definition is copied here for convenience:

Definition 2.1 (Symmetric distance)**.** Let u, v be vectors of elements drawn from domain $\mathcal{X}$. Define $h_v(\ell)$ as the multiplicity of element ℓ in vector v. For example, if v contains five instances of the number "21", then $h_v(21) = 5$.

A definition of the symmetric distance between u and v, then, is

$$d_{\mathrm{Sym}}(u, v) = \sum_{z \in \mathcal{X}} |h_u(z) - h_v(z)|.$$

TODO: Add versioned link to proof defns doc.

Combining the assumptions that `inf_cast`($\mathrm{d_{in}}$, `TO`) $\leq \mathrm{d_{out}}$ and that `v`, `w` are $\mathrm{d_{in}}$-close under the symmetric distance metric means that

$$d_{\mathrm{Sym}}(\mathtt{u}, \mathtt{v}) \leq \mathrm{d_{in}} \leq \mathrm{d_{out}}. \tag{1}$$

Let $\mathcal{X}$ be the domain of all elements of type `TIA`. Therefore, we see that the symmetric distance between `u` and `v` is

$$d_{\mathrm{Sym}}(\mathtt{u}, \mathtt{v}) = \sum_{z \in \mathcal{X}} |h_{\mathtt{u}}(z) - h_{\mathtt{v}}(z)| \leq \mathrm{d_{in}} \leq \mathrm{d_{out}}. \tag{2}$$

Lemma 2.2. *For vector v with each element $\ell \in v$ drawn from domain $\mathcal{X}$, `len(v)` $= \sum_{z \in \mathcal{X}} h_v(z)$.*

Proof. Every element $\ell \in v$ is drawn from domain $\mathcal{X}$, so summing over all $z \in \mathcal{X}$ will sum over every element $\ell \in x$. Recall that definition 2.1 states that $h_v(z)$ will return the number of occurrences of value z in vector v. Therefore, $\sum_{z\in\mathcal{X}} h_v(z)$ is the sum of the number of occurrences of each unique value; this is equivalent to the total number of items in the vector. By the definition of `len`

TODO: link to versioned pseudocode defs

, then, $\sum_{z\in\mathcal{X}} h_v(z)$ is equivalent to `len(v)`. $\square$

Question: I am unsure whether the proof of lemma 2.2 is good. It seems clear to me that `len(v)` $= \sum_{z\in\mathcal{X}} h_v(z)$, so I had trouble knowing what needs to be written and what doesn't need to be written.

We now prove that `len(u)` and `len(v)` are $\mathrm{d_{out}}$-close. By lemma 2.2, we know that `len(v)` $= \sum_{z\in\mathcal{X}} h_v(z)$. Substituting, we have

$$|\mathtt{len(u)} - \mathtt{len(v)}| = |\sum_{z\in\mathcal{X}} h_{\mathtt{u}}(z) - \sum_{z\in\mathcal{X}} h_{\mathtt{v}}(z)| = |\sum_{z\in\mathcal{X}} (h_{\mathtt{u}}(z) - h_{\mathtt{v}}(z))|. \quad (3)$$

By the triangle inequality,

$$|\sum_{z\in\mathcal{X}} (h_{\mathtt{u}}(z) - h_{\mathtt{v}}(z))| \le \sum_{z\in\mathcal{X}} |h_{\mathtt{u}}(z) - h_{\mathtt{v}}(z)|. \quad (4)$$

Therefore, combining equation 3 and inequality 4, we have that

$$|\mathtt{len(u)} - \mathtt{len(v)}| \le \sum_{z\in\mathcal{X}} |h_{\mathtt{u}}(z) - h_{\mathtt{v}}(z)|. \quad (5)$$

Combining inequalities 5 and 2, we have

$$|\mathtt{len(u)} - \mathtt{len(v)}| \le \mathrm{d_{out}}, \quad (6)$$

so `len(u)` and `len(v)` must be $\mathrm{d_{out}}$-close. This, however, does not complete the proof that the stability relation holds because `function(u)` does not return `len(u)`, but either `exact_cast(len(u),TO)` or – in the event `exact_cast` fails – `get_max_consecutive_int(TO)`.

We now consider the two cases that could occur:

1. (Without loss of generality, `exact_cast(len(u),TO)` fails and `exact_cast(len(v),TO)` succeeds). Because `TO` has trait `ExactIntCast(usize)`, if the `exact_cast` fails for `len(u)`, we then know that `len(u)` is greater than `get_max_consecutive_int(TO)`.

Likewise, if the `exact_cast` succeeds for `len(v)`, we then know that `len(v)` is no larger than `get_max_consecutive_int(TO)`. Therefore, because the return value `get_max_consecutive_int(TO)` for `u` is smaller than the true length value `len(u)`, the absolute difference between the output for `u` and the output for `v` will be *smaller* than the absolute distance between `len(u)` and `len(v)`. Since we showed that the `len(u)` and `len(v)` are $\mathrm{d_{out}}$-close in inequality 6, therefore the outputs will still be $\mathrm{d_{out}}$-close.

Note that if `exact_cast` fails for both `len(u)` and `len(v)`, then the output for both `u` and `v` is `get_max_consecutive_int(TO)`, resulting in an absolute distance of 0 between the outputs – the smallest possible absolute distance – so the outputs for `u` and `v` must be $\mathrm{d_{out}}$-close.

2. (Both `exact_cast(len(u),TO)` and `exact_cast(len(v),TO)` succeed). Because `TO` implements `ExactIntCast(usize)`, we know `exact_casts` from `len(u)` to `TO` will be exact. Therefore, the returned values will be `len(u)` and `len(v)`, except the values will now be of type `TO`. Since we showed that the `len(u)` and `len(v)` are $\mathrm{d_{out}}$-close in inequality 6, therefore the `exact_cast`ed lengths will also be $\mathrm{d_{out}}$-close.

Because the outputs will always be $\mathrm{d_{out}}$-close for inputs that follow the conditions specified in part 2 of theorem 2.1, we see that the stability guarantee is proven.

□