Privacy Proofs for OpenDP: MakeCount

Connor Wagaman – wagaman@college.harvard.edu

July 16, 2021

Contents

1 MakeCount

1.1 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.

```
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()))
}
```

1.2 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".

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)`
  - `IntDistance` has trait `InfCast(TO)`
  - **Question:** The final bullet point above 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). Should this precondition be included here or not?

Postconditions: a `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)
```

```
5      input_metric = SymmetricDistance()
6      output_metric = AbsoluteDistance(TO)
7
8      # give the Transformation the following properties
9      max_value = get_max_consecutive_int(TO)
10     def function(data: Vec<TIA>) -> TO:
11         try:
12             return exact_int_cast(len(data), TO)
13         except FailedCast:
14             return max_value
15     def stability_relation(din:u32, dout:TO) -> bool:
16         return 1 * inf_cast(din,TO) <= dout
17
18     # now, return the Transformation
19     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, the* `Transformation` *returned by* `MakeCount` *has 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 section 1.2, we see that any 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`. Therefore, since our output domain is any value of type `TO`, we see that `function` has the appropriate output domain `output_domain`.

Moreover, for some input vector `v` drawn from `input_domain`, `function` either returns `exact_int_cast(len(data), TO)`, which will be of type `TO` by the definition of `exact_int_cast`; or, if the casting fails, it returns `get_max_consecutive_int(TO)` which, from our definition of `get_max_consecutive_int`, will be of type `TO`. Therefore, since our output domain is always some value of type `TO`, we see that `function` has the appropriate output domain `output_domain`. □

Question: Is the second paragraph in the proof above of "Appropriate output domain" necessary?

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 `SymmetricDistance` 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 has the necessary trait, we see that it is compatible with our `output_metric` of `AbsoluteDistance`. □

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 1.2.)

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`$) \leq \mathrm{d_{out}}$. Recall 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 1.2).

We now refer to the definition of symmetric distance provided in the Proof Definitions document; 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 $m_v(\ell)$ as the multiplicity of element ℓ in vector v. For example, if v contains five instances of the number "21", then $m_v(21) = 5$.

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

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

Question: How should I refer readers to a definition located in another document? I know how to use `\label{...}` and `\ref{...}`, but that's only for referring to definitions, sections, etc. located in the same doc.

Question: As a follow-up to the question above, if there's not a good way to refer to definitions in proofs, should important definitions be copied into the proof doc (as above), or should I remove "in-proof" definitions and rely on readers to track down the right definition in the proof definitions document, which is a document that may be continuously updated for the lifetime of the OpenDP project?

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

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

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

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

The `function` used in `MakeCount` sums over a single data type, namely a row. Let `rows` be a one-element domain, where every element of type `TIA` is considered to be the same element of `rows`; and let the single element be called `row`. Also, as in the Pseudocode definitions document, let `len(vec)` be a function that returns the number of rows in vector `vec`.

Therefore, using the notation in definition 2.1, we can write

$$|\texttt{len(u)} - \texttt{len(v)}| = \sum_{z \in \texttt{rows}} |m_{\texttt{u}}(z) - m_{\texttt{v}}(z)| = |m_{\texttt{u}}(\texttt{row}) - m_{\texttt{v}}(\texttt{row})| \tag{3}$$

(note that the summation term is removed in the final term in equation 3 since the domain `rows` consists of the single element `row`).

By the triangle inequality, then, we see that

$$|m_{\texttt{u}}(\texttt{row}) - m_{\texttt{v}}(\texttt{row})| \leq \sum_{z \in \mathcal{X}} |m_{\texttt{u}}(z) - m_{\texttt{v}}(z)|. \tag{4}$$

Combining equations 3 and 4 tells us that $|\texttt{len(u)} - \texttt{len(v)}| = |m_{\texttt{u}}(\texttt{row}) - m_{\texttt{v}}(\texttt{row})| \leq \sum_{z \in \mathcal{X}} |m_{\texttt{u}}(z) - m_{\texttt{v}}(z)|$; combining this with equation 2 tells us that we have

$$|\texttt{len(u)} - \texttt{len(v)}| = |m_{\texttt{u}}(\texttt{row}) - m_{\texttt{v}}(\texttt{row})| \leq \mathrm{d_{out}}, \tag{5}$$

so `len(u)` and `len(v)` must be $\mathrm{d_{out}}$-close. This, however, does not complete the proof 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 equation 5, 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 equation 5, 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.

□