# Privacy Proofs for OpenDP: MakeCount

Connor Wagaman – wagaman@college.harvard.edu

Version as of September 6, 2021 (UTC)

## Contents



Note: This proof document incorporates all suggestions made as of September 6, 2021. There is one question about the proof of lemma 2.3 (this lemma is new); the question appears immediately after the proof of the lemma.

### 0.1 Versions of definitions documents

When looking for definitions for terms that appear in this document, the following versions of the definitions documents should be used.

- **Pseudocode definitions document:** This proof file uses the version of the pseudocode definitions document available as of September 6, 2021, which can be found at this link (archived here).
- **Proof definitions document:** This file uses the version of the proof definitions document available as of September 6, 2021, which can be found at this link (archived here).

# 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/c3b5c3bd9fc50c556362b628 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" (see section 0.1 for links to the document).*

*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 because they have (1) a multiplicative identity element, (2) every value of type `usize` that falls within

the minimum and maximum consecutive integers of type `u32` has an exact representation of type `u32` (the same applies for `i64`), and (3) multiplication and division apply to types `u32` and `i64`, values of type `u32` and `i64` have a partial ordering, and every value of type `usize` can be `inf_cast`ed to a value of type `u32` (meaning that the `inf_cast` will either result in an error or will result in a value of type `u32` that is at least as large as the input value of type `usize`; this also applies for `i64`).

Currently, the `ExactIntCast` and `DistanceConstant` traits are implemented for casting between all numeric types, with the exception of `InfCast` not being implemented for going to or from `usize` and `isize` (and thus `DistanceConstant` not being implemented). Therefore, `usize` and `isize` do not have these traits.

– `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 (type `IntDistance`) that is at least as big. On the other hand, `IntDistance` has trait `InfCast(u64)`, because every value of type `u64` can either be `infcast`ed to a value of type `u32` that is at least as large, or an error can be returned.

  On the other hand, `IntDistance` does not have trait `InfCast(usize)` because `InfCast` is not implemented for going to or from `usize`.

**Question:** Salil had recommended that examples of types that do and don't work be provided. Should these examples and anti-examples be provided? Should I just say, "`u32` has these traits", or should I provide the description for why it has these traits (as I did above)?

**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
```

```
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, `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 1.2.)

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`$(\mathrm{d_{in}}, \texttt{TO}) \leq \mathrm{d_{out}}$. From the pseudocode definitions file linked in section 0.1, 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 1.2).

We now refer to the definition of symmetric distance provided in the proof definitions document (a link to this document is available in section 0.1); the definition is copied here for convenience:

**Definition 2.2** (Symmetric distance)**.** Let $u, v$ be vectors of elements drawn from domain $\mathcal{X}$. Define $h_v(\ell)$ as the multiplicity of element $\ell$ 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

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

Combining the assumptions that `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

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

**Lemma 2.3.** *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 2.2 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` available in the pseudocode definitions document linked in section 0.1, then, $\sum_{z \in \mathcal{X}} h_v(z)$ is equivalent to `len(v)`.

□

**Question:** I am unsure whether the proof of lemma 2.3 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 $d_{out}$-close. By lemma 2.3, we know that `len(v)` $= \sum_{z\in\mathcal{X}} h_v(z)$. Substituting, we have

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

By the triangle inequality,

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

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

$$|\texttt{len(u)} - \texttt{len(v)}| \leq \sum_{z\in\mathcal{X}} |h_{\texttt{u}}(z) - h_{\texttt{v}}(z)|. \tag{5}$$

Combining inequalities 5 and 2, we have

$$|\texttt{len(u)} - \texttt{len(v)}| \leq d_{out}, \tag{6}$$

so `len(u)` and `len(v)` must be $d_{out}$-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 $d_{out}$-close in inequality 6, therefore the outputs will still be $d_{out}$-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 $d_{out}$-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 $d_{out}$-close in inequality 6, therefore the `exact_casted` lengths will also be $d_{out}$-close.

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

$\square$