# Privacy Proofs for OpenDP: Impute Constant Transformation

Grace Tian

Summer 2021

## Contents



## 1 Algorithm Implementation

### 1.1 Code in Rust

The current OpenDP library contains the `make_impute_constant` function implementing the impute constant function. This is defined in lines 62-75 of the file `impute.rs` in the Git repository (https://github.com/opendp/opendp/blob/21-impute/rust/opendp/src/trans/impute.rs#L62-L75).

```
58  /// A [`Transformation`] that imputes elementwise with a constant value.
59  /// Maps a Vec<Option<T>> -> Vec<T> if input domain is AllDomain<Option<T>>,
60  ///     or Vec<T> -> Vec<T> if input domain is NullableDomain<AllDomain<T>>
61  /// Type argument DA is "Domain of the Atom"; the domain type inside VectorDomain.
62  pub fn make_impute_constant<DA, M>(
63      constant: DA::NonNull
64  ) -> Fallible<Transformation<VectorDomain<DA>, VectorDomain<AllDomain<DA::NonNull>>, M, M>>
65      where DA: ImputableDomain,
66            DA::NonNull: 'static + Clone,
67            DA::Carrier: 'static,
68            M: DatasetMetric {
69      if DA::is_null(&constant) { return fallible!(MakeTransformation, "Constant may not be null.") }
70
71      make_row_by_row(
72          DA::new(),
73          AllDomain::new(),
74          move |v| DA::impute_constant(v, &constant).clone())
75  }
```

### 1.2 Pseudo Code in Python

#### Preconditions

To ensure the correctness of the output, we require the following preconditions:

- **User-specified types:**
  - Variable `constant` must be of type `DA::NonNull`
  - Type `DA` must have traits `ImputableDomain`
  - `DA::NonNull` has traits `Clone`

### Postconditions

- Either a valid `Transformation` is returned or an error is returned.

(grace) Not sure if I need to include the error check for whether the constant is nonnull in line 69 code. Is it already checked?

```
def make_impute_constant(constant : DA::NonNull):
    input_domain = VectorDomain(DA);
    output_domain = VectorDomain(AllDomain(DA::NonNull));

    def Relation(d_in: u32, d_out: u32) -> bool:
        return d_out >= d_in*1

    def function(data: Vec(DA)) -> Vec(DA::NonNull):
        def impute_constant(x: DA) -> DA::NonNull:
            return constant if x.is_null else x
        return list(map(impute_constant, data))

    return Transformation(input_domain, output_domain, function(data?),
    input_metric, output_metric, stability_relation=Relation)
```

(grace) Will need to change pseudocode so that it returns the result of a make row by row transformation (which the code does) instead of a Transformation directly. Make sure to ask.

## 1.3 Proof

**Theorem 1.1.** *For every setting of the input parameters* `constant` *to* `make_impute_constant` *such that the given preconditions hold, the transformation returned by* `make_impute_constant` *has the following properties:*

1. (Appropriate output domain). *If vector $v$ is in the* `input_domain`*, then* `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 pair of elements $v, w$ in* `input_domain` *and for every pair (*`d_in`*,* `d_out`*), where* `d_in` *is of the associated type for* `input_metric` *and* `d_out` *is the associated type for* `output_metric`*, if $v, w$ are $d_{in}$-close under* `input_metric` *and* `Relation(d_in, d_out)` $=$ `True`*, then* `function`$(v)$*,* `function`$(w)$ *are $d_{out}$-close under* `output_metric`*.*

*Proof.* 1. **(Appropriate output domain).** In the case of `make_impute_constant`, this corresponds to showing that for every vector $v$ of elements of type `DA`, `function`$(v)$ is a vector of elements of type `DA::NonNull`.

The `function`$(v)$ has type `Vec(DA)` follows from the assumption that element $v$ is in `input_domain` and from the type signature of `function` in line 8 of the pseudocode (Section 1.2), which takes in an element of type `Vec(DA)` and returns an element of type `Vec(DA::NonNull)`. 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.

Lastly, we ensure that return type must be `NonNull` (grace) Attribute? Property? Carrier `DA::NonNull` because we check to make sure the `constant` being imputed in the function must be null. (grace) Not sure if this needs to be explicit in the pseudo code with the error check, or if I can just cite some trait property. This check already exists because in the input of the `make_impute_constant`, `constant` is required to have type of `DA::NonNull`.

2. **(Domain-metric compatibility).** The Symmetric distance is both the `input_metric` and `output_metric`.Symmetric distance is compatible with `VectorDomain(T)` for any generic type `T`, as stated in "List of definitions used in the pseudocode". The theorem holds because for `make_impute_constant`, the input domain is `VectorDomain(DA)` for generic type `TI` and the output domain is `VectorDomain(AllDomain(DA::NonNull))`.

3. **(Stability guarantee).** We know that $d_{in} \leq d_{out}$ because `Relation`$(d_{in},\ d_{out})$ `= True`. Since the vectors $v, w$ are $d_{in}$-close, then $d_{Sym}(v, w) \leq d_{in}$.

The function transformation just replaces the `null` element in vectors $v$ and $w$ with `constant`. Since the null element is also counted toward the symmetric distance of the transformation, the symmetric distance of `function`$(v)$ and `function`$(w)$ stays the same. Therefore the transformation is $d_{out}$ close: $d_{sym}(\texttt{function}(v), \texttt{function}(w)) = d_{sym}(v, w) \leq d_{in} \leq d_{out}$

□