List of definitions used in the pseudocode

Sílvia Casacuberta, Grace Tian, and Connor Wagaman

June–July 2021

We use the following guideline: if a term appears in the preconditions & pseudocode section of a proof document, then this term is defined in the "List of definitions used in the pseudocode" document. Otherwise, it appears in the "List of definitions used in the proofs" document.

We maintain the terms in alphabetical order within each section. "TODOs" should be included at the end of the corresponding section. On the other hand, "TODOs" which better specify an already-defined term should be included immediately following the definition of that term. Examples should never be part of the definition, but we encourage their use right after the definition of a term.

We also recommend linking to the Rust Standard Library when the term is defined there.

Contents

List of terms that have not yet been added

- The compatibility pairing and subdomains. Compatibility inheritance for subdomains and proof for subdomain traits. But this is awaiting the specific final implementation.
- Clarify why we specify vectors for `SizedDomain`
- Add definition of a valid measurement
- Double-check newly added definitions, and trait vs function (e.g., `SaturatingAdd` and `saturating_add`)
- Incorporate corrections from Prof. Vadhan on GitHub
- Add type trait `Laplace Domain`, and the other ones from base Laplace: `D::Atom`, `SampleLaplace`, `Float`, and `InfCast(D::Atom)`.
- Add function `is_sign_negative()` from Laplace
- Add function `.recip()` from Laplace
- Add function `sample_laplace` (idealized, for now)
- Add `IntDistance`, although Mike said to keep using `u32` in the pseudocode type signing for now.
- Definition of the Rust iterator.
- Do we need `fold`? Maybe always use loops

1 Types

Note 1 (A note on `()` vs. `[]`)**.** Parentheses, `()`, are used to create an instance of a domain. Square brackets, `[]`, are used to describe the type of a domain.

For example, `AllDomain(i8)` is the domain of all values of type `i8`. However, the type of `AllDomain(i8)` – the domain itself, not the elements of the domain – is `AllDomain[i8]`; note the square brackets. This typing style is inspired by the notation used in Python; see https://docs.python.org/3/library/typing.html.

Definition 1.1 (`bool`)**.** The type `bool` represents a value which can only be either `True` or `False`. If a `bool` is casted to an integer, `True` will be `1` and `False` will be `0`.

Definition 1.2 (`::Carrier`)**.** `SomeDomain::Carrier` is the type of a member in `SomeDomain`, where `SomeDomain` is a domain.

For example, `AllDomain(T)::Carrier` is `T`.

Definition 1.3 (`f32`). `f32` is the Rust 32-bit floating point type. See https://doc.rust-lang.org/std/primitive.f32.html.

Definition 1.4 (`f64`). `f64` is the Rust 64-bit floating point type. See https://doc.rust-lang.org/std/primitive.f64.html.

TODO (future – not enough info yet): Add / pointers to "binary64" type defined in IEEE 754-2008.

Definition 1.5 (`IntDistance`). `IntDistance` is equivalent to `u32`.

Definition 1.6 (`::NonNull`). `SomeDomain::NonNull` is (grace) TODO in `SomeDomain`, where `SomeDomain` is a domain.

For example, (grace) TODO.

Definition 1.7 (`u32`). `u32` is the Rust 32-bit unsigned integer type. If v is a value of type `u32`, then we know that $v \in \{0, 1, 2, \ldots, 2^{32} - 1\}$. See https://doc.rust-lang.org/std/primitive.u32.html.

Definition 1.8 (`usize`). `usize` is defined differently on 32-bit and 64-bit machines. This is because the size of this primitive is equal to the number of bytes it takes to reference any location in memory.

- **32-bit machines:** if `v` is a value of type `usize`, then `v` $\in \{0, 1, 2, \ldots, 2^{32} - 1\}$
- **64-bit machines:** if `v` is a value of type `usize`, then `v` $\in \{0, 1, 2, \ldots, 2^{64} - 1\}$

See https://doc.rust-lang.org/std/primitive.usize.html.

Definition 1.9 (`Vec[T]`). The Rust type `Vec[T]` consists of ordered lists of type `T`. For example, if `T = bool`, then values of type `Vec[T]` include `[], [0], [1], [0, 0], ...`. See https://doc.rust-lang.org/std/vec/struct.Vec.html.

1.1 Notes, todos, questions

TODO (future – not enough info yet): Include info on MPFR, and possibly relate it to our existing definitions of floats.

TODO (future – not enough info yet): Define plus, minus, etc. below each type on which they operate. For example, the definition for `u32` should also include a definition of plus on `u32`, multiplication on `u32`, etc.

Question for reviewers: Should we have a general definition for "floats" (and "integers"?), or is it sufficiently understood what a float is in general?

2 Domains

A *data domain* is a representation of the set of values on which a metric or function can operate. For example, if a function accepts inputs from the domain `IntervalDomain(1:u32,17:u32)`, this means that the function can take any input value `v` of type `u32` such that `1 <= v` and `v <= 17`.

Definition 2.1 (`AllDomain`). `AllDomain(T)` is the domain of all values of type `T`. This domain has type `AllDomain[T]`.

For example, `AllDomain(u32)` is the domain of all values of type `u32`.

Definition 2.2 (`ImputableDomain`). Any domain for which the `ImputableDomain` trait is implemented for it, has

- an associated type `NonNull`. `VectorDomain::NonNull` is the datum type after imputation.
- an imputation function `impute_constant` that replaces a null value with a constant or passes a non-null value through
- a function `is_null` to check if a value is null
- a function `null` to construct an instance of the domain

Definition 2.3 (`IntervalDomain`). For any type `T` with trait `TotalOrd`,[1] `IntervalDomain(L:T, U:T)` is the domain of all values `v` of type `T` such that `L <= v` and `v <= U`, for a type `T` that has a total ordering (`T` has trait `TotalOrd`) and for values `L <= U` of type `T`. This domain has type `IntervalDomain[T]`.

An important remark is that the Rust implementation of `IntervalDomain` checks that `L` $\leq$ `U`, and returns an error if `L` $\geq$ `U`. Therefore, any transformation or measurement that uses `IntervalDomain` does not need to re-check this constraint and raise a possible exception for it.

Note that, because both `L` and `U` are of type `T`, there is no need to explicitly pass `T`; the type `T` can be inferred. `IntervalDomain` is defined on any type that implements the trait `TotalOrd`. For example, `IntervalDomain(1:u32, 17:u32)` corresponds to a domain that contains all the `u32` values `v` such that `1 <= v` and `v <= 17`; it has type `IntervalDomain[u32]`.

Definition 2.4 (`InherentNullDomain`). `InherentNullDomain(inner_domain:D)` is the domain of all values of data domain `inner_domain` and `null` values. This domain has type `InherentNullDomain[D]`.

Definition 2.5 (`SizedDomain`). `SizedDomain(inner_domain:D, n:usize)` is the domain of all vectors of length `n` drawn from domain `inner_domain`. This domain has type `SizedDomain[D]`. (silvia) As discussed on July 19, this is not only for vectors

For example, `SizedDomain(VectorDomain(AllDomain(u32)), n)` is the domain of all vectors of length `n` with elements of type `u32`.

Definition 2.6 (`VectorDomain`). `VectorDomain(inner_domain:D)` is the domain of all vectors of elements drawn from domain `inner_domain`. This domain has type `VectorDomain[D]`.

2.1 Subdomains

---

[1] As of June 28, the OpenDP library requires the weaker condition of partial ordering (implements `PartialOrd`) instead.

2.2 Notes, todos, questions

TODO (future – not enough info yet): Add clampable domain (`ClampableDomain`) – waiting until `TotalOrd` is fully implemented in the OpenDP library.

TODO (future – not enough info yet): As of July 20, OpenDP plans to include subdomains (see the Architecture meeting notes for 20/7). We have to include them and prove that they are indeed subdomains. Then in the metric definition it is enough to list the most general domain, since the domain-metric compatibility in inherited. We will add the necessary information here after Mike and Andy have finished the implementation details.

Theorem 2.1 (Domain-metric compatibility inheritance.). *Given a domain* `D`*, for any subdomain* `S` $\subseteq$ `D`*, if* `D` *is compatible with metric* `M` *then* `S` *is compatible with metric* `M`*.*

(silvia) Since it is a theorem, should it be moved to the proof defs doc?

TODO: Need to add definition for what does it mean to be compatible.

3 Traits

Definition 3.1 (`Abs`). A type `T` has trait `Abs` if and only if the absolute value of a value of type `T` can be taken.

Definition 3.2 (`Bounded`). A type `T` has trait `Bounded` if and only if `T` has some upper bound and some lower bound (some smallest possible value and some largest possible value).

Definition 3.3 (`CheckedMul`). A type `T` has trait `CheckedMul` if it performs multiplication that returns "None" if overflowing.

Definition 3.4 (`DistanceConstant`). A type `TO` has trait `DistanceConstant(TI)` if and only if

- `TO` has trait `Mul(Output=TO)` (multiplication can be done with type `TO`)
- `TO` has trait `Div(Output=TO)` (some form of inverse mapping can be done with type `TO`)
- `TO` has trait `PartialOrd` (`TO` has a partial ordering)
- `TO` has trait `InfCast(TI)`

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

Definition 3.5 (`Domain`). A type `T` has trait `Domain` if and only if it can represent a set of values that make up a domain. The `Domain` implementation prescribes a type for members of the domain, as well as a method to check if any instance of that type is a member of that domain.

Definition 3.6 (`ExactIntCast`). A type `TO` has trait `ExactIntCast(TI)` if and only if:

1. It has trait `MaxConsecutiveInt`.
2. Every value of type `TI` can be `exact_int_casted` exactly to a value of type `TO`, as long as the original value of type `TI` is no smaller than `get_min_consecutive_int(TO)` and no larger than `get_max_consecutive_int(TO)`.

A cast error is returned when the value being `exact_int_casted` is greater than `get_max_consecutive_int(TO)` or less than `get_min_consecutive_int(TO)`.

Definition 3.7 (`Float`)**.** Generic trait for floating point numbers.

TODO: The Float trait adds `Div<Self, Output = Self>`. In general, add the dependencies between traits.

Definition 3.8 (`InfCast`)**.** A type `TO` has trait `InfCast(TI)` if and only if every cast from a value of type `TI` to type `TO` will result in a value of type `TO` that is at least as big as the value of type `TI`.

Definition 3.9 (`InherentNull`)**.** A type `T` has trait `InherentNull` if and only if type `T` can hold some value `null`.

As of July 16, 2021, only `f32` and `f64` have the trait `InherentNull`.

Definition 3.10 (`MaxConsecutiveInt`)**.** A type `T` has trait `MaxConsecutiveInt` if and only if there is some maximum nonnegative integer `i` such that all integers from 0 up to `i` (inclusive) can be expressed as a value of type `T`; but such that the next integer that can be expressed by `T` is not $\mathtt{i} + 1$.

Definition 3.11 (`Metric`)**.** A type `T` has trait `Metric` if and only if it can represent a metric for quantifying distances between values in a set. The `Metric` implementation additionally prescribes the type to use for representing distances.

Definition 3.12 (`One`)**.** A type `T` has trait `One` if and only if `T` has some multiplicative identity element.

Definition 3.13 (`OptionNull`)**.** A type `Option<T>` has trait `OptionNull` if and only if `null` can be represented as `Option<T>` (grace) Check this because not sure if it makes sense.

Definition 3.14 (`PartialEq`)**.** A type `T` has trait `PartialEq` if and only if `T` has some (grace) TODO

Definition 3.15 (`PartialOrd`)**.** A type `T` has trait `PartialOrd` if for all elements a, b, c of type `T`, the following properties are satisfied:

1. Reflexivity: $a \leq a$,
2. Antisymmetry: if $a \leq b$ and $b \leq a$ then $a = b$,
3. Transitivity: if $a \leq b$ and $b \leq c$ then $a \leq c$.

Definition 3.16 (`SaturatingAdd`)**.** A type `T` has trait `SaturatingAdd` if it performs addition that saturates at the numeric bounds instead of overflowing.

Definition 3.17 (`TotalOrd`)**.** A type `T` has trait `TotalOrd` if and only if `T` has trait `PartialOrd` and moreover all elements are comparable; that is, for all elements a, b of type `T`, either $a \leq b$ or $b \leq a$.

Definition 3.18 (`Zero`)**.** A type `T` has trait `Zero` if and only if `T` has some additive identity element.

3.1 Math-related definitions

Definition 3.19 (`Add(Output=T)`)**.** A type `T` has trait `Add(Output=T)` if and only if addition can be performed between elements of type `T`, with the result of the addition also being of type `T`.

Definition 3.20 (`Div(Output=T)`)**.** A type `T` has trait `Div(Output=T)` if and only if division can be performed between elements of type `T`, with the result of the division also being of type `T`.

Definition 3.21 (`Mul(Output=T)`)**.** A type `T` has trait `Mul(Output=T)` if and only if multiplication can be performed between elements of type `T`, with the result of the multiplication also being of type `T`.

Definition 3.22 (`Sub(Output=T)`)**.** A type `T` has trait `Sub(Output=T)` if and only if subtraction can be performed between elements of type `T`, with the result of the subtraction also being of type `T`.

Definition 3.23 (`Sum(Output=T)`)**.** A type `T` has trait `Sum(Output=T)` if and only if such type can be created by summing up an iterator. This trait is used to implement the `sum` method on iterators. Types which implement the trait can be generated by the `sum()` method.

3.2 Traits that need not appear in the preconditions

- `'static`. Notes: `'static` is not a type; it is a lifetime name (this is a Rust definition).
- `Clone`
- `Copy`

3.3 Notes, todos, questions

4 Functions

4.1 Functions in the pseudocode language

Definition 4.1 (`abs`)**.** Given an element `var` of type `T`, where `T` must have trait `Abs`, the function `abs` reuturns the absolute value of `var`.

Definition 4.2 (`assert`)**.** The function `assert` is followed by an expression. If `some_expression` evaluates to `False`, then `assert some_expression` results in an error that prevents the code from proceeding further. In Python, this is called `assert`. See https://docs.python.org/3/reference/simple_stmts.html#the-assert-statement.

Definition 4.3 (`can_cast`)**.** The function `can_cast(type1,type2)` returns `True` if and only if no data would be lost by casting from `type1` to `type2`. In other words, it returns `True` if and only if there is an injection from `type1` to `type2`. See https://doc.rust-lang.org/std/convert/trait.TryFrom.html.

For example, `can_cast(u32,u64)` will return `True` because a `u32` can always be expressed as a `u64`; conversely, `can_cast(u64,u32)` will return `False` because a `u64` could be too big to be expressed as a `u32`, and then data would be lost.

Definition 4.4 (`cast`). `cast(val:TI, TO)` converts `val` of type `TI` to the corresponding `val` of type `TO`, and returns `val` of type `TO`. Returns an error if the conversion is unsuccessful.

Definition 4.5 (`checked_mul`). Given two elements `var1, var2` of type `T` with trait `Mul(Output=T)`, `checked.mul(var1, var2)` returns `var1*var2` if the result does not overflow, and else returns "None".

Definition 4.6 (`exact_int_cast`). This function only works for types `TO` that have trait `ExactIntCast(TI)`. For any given `val` such that `val` is between `get_min_consecutive_int(TO)` and `get_max_consecutive_int(TO)`, then `exact_int_cast(val:TI,TO)` returns the an integer value of type `TO` equal to the integer value held by `val` (which was of type `TI`); otherwise, a cast error is returned.

Definition 4.7 (`get_input_domain`). The function `get_input_domain(function)` returns the input domain of arguments passed to function `function`.

Definition 4.8 (`get_input_metric`). The function `get_input_metric(some_relation)` returns the input metric used by the relation `some_relation`.

Definition 4.9 (`get_max_consecutive_int`). This function is only defined on types `T` that have trait `MaxConsecutiveInt`. The function `get_max_value(T)` returns the maximum nonnegative integer `i` such that all integers from 0 up to `i` (inclusive) can be expressed as a value of type `T`; but such that the next integer that can be expressed by `T` is not `i` + 1. The return value is of type `T`.

Definition 4.10 (`get_max_value`). This function is only defined on types `T` that have a total ordering. The function `get_max_value(T)` returns the maximum value that can be expressed by an object of type `T`. The return value is of type `T`.

Definition 4.11 (`get_min_consecutive_int`). This function is only defined on types `T` that have trait `MinConsecutiveInt`. `get_max_value(T)` returns the minimum negative integer `i` such that all integers from 0 down to `i` (inclusive) can be expressed as a value of type `T`; but such that the next integer that can be expressed by `T` is not `i` − 1. The return value is of type `T`.

Definition 4.12 (`get_min_value`). This function is only defined on types `T` that have a total ordering. The function `get_min_value(T)` returns the minimum value that can be expressed by an object of type `T`. The return value is of type `T`.

Definition 4.13 (`get_output_domain`). The function `get_output_domain(function)` returns the output domain of values returned by function `function`.

Definition 4.14 (`get_output_metric`). The function `get_output_metric(some_relation)` returns the output metric used by the relation `some_relation`.

Definition 4.15 (`has_trait`). The function `has_trait(T,(trait1,trait2,...))` is a function that returns `True` if and only if the type `T` implements `trait1`, `trait2`, etc.

Definition 4.16 (`inf_cast`)**.** This function is only defined for casting to types `TO` that have trait `InfCast(TI)`. The function `inf_cast(val:TI, TO)` casts `val` to a value of type `TO` and returns that value. Specifically, `val` will be casted to the value of type `TO` that is closest to `val` and at least as large as `val`. If `inf_cast` is not able to cast `val` to a value of type `TO` at least as large as `val`, then an error is returned instead.

Property: `inf_casted` distances are never less than input distances.

Definition 4.17 (`is_instance`)**.** The function `is_instance(var,T)` returns `True` if and only if the variable `var` is of type `T`.

Definition 4.18 (`is_none`)**.** The function `var.is_none` returns `True` if and only if `var` which is of `float` is equal to "None".

Definition 4.19 (`is_null`)**.** The function `var.is_null` returns `True` if and only if `var` which is of `Option<T>` is not equal to "Null".

Remark 1 (Imputable Domain)**.** To check for nullity, we use `v.is_null()` if v is a float, or `v.is_none()` if v is an `Option<T>`. To abstract these functions, we define the `ImputableDomain` trait to capture both notions of nullity.

Definition 4.20 (`len`)**.** The function `len(vector_name)` returns the number of elements in `vector vector_name`. Output is of type `usize`, so the return value v on 32-bit machines is $v \in \{0, 1, 2, \ldots, 2^{32} - 1\}$; likewise, the return value on 64-bit machines is $v \in \{0, 1, 2, \ldots, 2^{64} - 1\}$. See `https://doc.rust-lang.org/std/vec/struct.Vec.html#method.len`.

Note: we do not call it `length` to avoid notational clashes with, for example, the Bounded Sum code.

Definition 4.21 (`map`)**.** A `map` applies a given function to all the items in an iterable without using an explicit for loop. Hence, `map(f, iter)` is an iterator that maps the values of `iter` with `f`. See ..

Definition 4.22 (`max`)**.** The function `max(var1, var2)` compares `var1` and `var2`, and returns the greater of the two values. When `var1` and `var2` are equivalent, it returns `var2`. The return type of `map` is also an iterator. See `https://doc.rust-lang.org/std/cmp/fn.max.html`.

Definition 4.23 (`min`)**.** The function `min(var1:T, var2:T)` compares `var1` and `var2`, and returns the lesser of the two values. When `var1` and `var2` are equivalent, it returns `var1`. See `https://doc.rust-lang.org/std/cmp/fn.min.html`.

(silvia) Max and min definitions should be consistent (type signing). Why are we not requiring `T` to have total order?

Definition 4.24 (`saturating_add`)**.** Given two elements `var1, var2` of type `T` with trait `Add(Output=T)`, the function `saturating_add(var1, var2)` returns `var1+var2` whenever `var1+var2` $\in$ `[get_min_value(T), get_max_value(T)]`. If `var1+var2` $<$ `get_min_value(T)`, then `saturating_add(var1, var2)` $=$ `get_min_value(T)`. Similarly, in the case where `var1+var2` $>$ `get_max_value(T)`, then `saturating_add(var1, var2)` $=$ `get_max_value(T)`. See `https://doc.rust-lang.org/std/intrinsics/fn.saturating_add.html`.

Definition 4.25 (`sum`)**.** The `sum` function adds the items of an iterable and returns their sum.

4.2 Notes, todos, questions

5 Data structures

Definition 5.1 (`list`)**.** A `list` is a data structure which is a changeable ordered sequence of elements.

Importantly, we remark that in some occasions in our Python-like pseudocodes we will write `list` as an equivalent for the Rust `Vec` in order to maintain a Python-like notation. For this reason, such a `list` will have type `Vec(T)` and be considered an element of `VectorDomain`. We will allow the use of the Rust-like term `Vec` when type signing the functions in the pseudocode and proving the corresponding domain properties in the proof.

6 Classes

Definition 6.1 (`Transformation`)**.** We define a `Transformation` in the following way.

Question for reviewers: Which pseudocode style is preferred for this definition? With preconditions (section 6.1) or without preconditions (section 6.2)?

6.1 Pseudocode with preconditions

- `input_domain` must have trait `Domain`
- `output_domain` must have trait `Domain`
- `function` must operate on inputs from `input_domain`, and it must produce outputs in `output_domain`
- `input_metric` must have trait `Metric`
- `output_metric` must have trait `Metric`
- `stability_relation` must operate on input metrics equal to `input_metric`, and it must operate on output metrics equal to `output_metric`

```
class Transformation:
    def __init__(self, input_domain, output_domain, function, input_metric,
     output_metric, stability_relation):

        self.input_domain = input_domain
        self.output_domain = output_domain

        self.function = function

        self.input_metric = input_metric
        self.output_metric = output_metric

        self.stability_relation = stability_relation
```

6.2 Pseudocode without preconditions

(connor) Mike helped a lot with this definition, so I'm hopeful it's fully correct, or at least very close.

```
class Transformation:
    def __init__(self, input_domain, output_domain, function, input_metric,
     output_metric, stability_relation):

        assert has_trait(input_domain, Domain)
        self.input_domain = input_domain
        assert has_trait(output_domain, Domain)
        self.output_domain = output_domain

        assert get_input_domain(function) == input_domain
        assert get_output_domain(function) == output_domain
        self.function = function

        assert has_trait(input_metric, Metric)
        self.input_metric = input_metric
        assert has_trait(output_metric, Metric)
        self.output_metric = output_metric

        assert get_input_metric(stability_relation) == input_metric
        assert get_output_metric(stability_relation) == output_metric
        self.stability_relation = stability_relation
```

In OpenDP (Rust), this is called `Transformation`. See https://github.com/opendp/opendp/blob/35dbdc73d7d74e049f5101a704d4e036bed365e8/rust/opendp/src/core.rs#L369-L376. When we refer to a *valid transformation* in the proofs, this is the precise definition.

Therefore, there is no need to include the following code snippet in all of the pseudocodes:

```
class Transformation:
    input_domain
    output_domain
    function
    input_metric
    output_metric
    stability_relation
```

TODO: Add the equivalent for Measurements. (Very similar; just change names)

7 Metrics

7.1 Dataset metrics

Metrics are used to measure the distances between data. Metrics have a *domain* on which the metric can measure distance, and an *associated type* that determines the type used to represent the distance between datasets.

Example: `SymmetricDistance` has a domain of `VectorDomain(AllDomain(T))`, which means that `SymmetricDistance` can be used to measure the distance between any objects that are vectors of elements of type `T`. `SymmetricDistance` has an associated type of `u32`, which means that a `u32` value is used to report the distance.

Definition 7.1 (`AbsoluteDistance(T)`). The definition of *absolute distance* in the "proof definitions" document tells how the distance between data is calculated.

- **Domain:** `AllDomain(T)`, where `T` has the trait `Sub(Output=T)`.
- **Associated type:** `Q`.
- ***d*-close:** For any two elements n, m in `AllDomain(T)`, where `T` denotes an arbitrary type with trait `Sub(Output=T)`, and d of generic type `Q`, we say that n, m are d-close under the absolute distance metric (abbreviated as d_{Abs}) whenever

$$d_{Abs}(n, m) = |n - m| \leq d.$$

Definition 7.2 (`SymmetricDistance`). The definition of *symmetric distance* in the "proof definitions" document tells how the distance between data is calculated.

- **Domains:** `VectorDomain(inner_domain)` where `inner_domain` is any domain. (silvia) Said on 19/7: for now, list also `SizedDomain(VectorDomain(inner_domain))`
- **Associated type:** `u32`.
- ***d*-close:** For any two vectors $u, v \in$ `VectorDomain(D)` and any d of type `u32`, we say that u, v are d-close under the symmetric distance metric (abbreviated as d_{Sym}) whenever

$$d_{Sym}(u, v) = |\text{MultiSets}(u)\Delta\text{MultiSets}(v)| \leq d.$$

Note: the associated type of `SymmetricDistance` is hard-coded as `u32`, so when declaring that the metric being used is `SymmetricDistance`, we only need to write `metric = SymmetricDistance();` by contrast, we need to write `AbsoluteDistance(T)` where `T` is the type on which we are taking the absolute distance since the associated type for `AbsoluteDistance` is not hard-coded.

TODO: Change to `IntDistance`.

7.2 Sensitivity metrics

Definition 7.3 (`L1Distance`).

Definition 7.4 (`L2Distance`).

7.3 Notes, todos, questions

Question: Add symmetric difference?

(connor) I don't think we need to. It is not a `\texttt{...}` term.

TODO: Need to learn how to cross-reference TeX files. After Prof. Vadhan's comment on 19/7, we should think of a systematic way to do dependency tracking.

TODO: Add compatibility pairing as described in the July 20 architecture meeting.