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

1 Types **2**
- 1.1 Notes, todos, questions . . . 2

2 Domains **3**
- 2.1 Notes, todos, questions . . . 3

3 Traits **3**
- 3.1 Math-related definitions . . . 5
- 3.2 Traits that need not appear in the preconditions . . . 5
- 3.3 Notes, todos, questions . . . 5

4 Functions **5**
- 4.1 Functions in the pseudocode language . . . 5
- 4.2 Notes, todos, questions . . . 7

5 Classes **7**
- 5.0.1 Pseudocode with preconditions . . . 7
- 5.0.2 Pseudocode without preconditions . . . 8

6 Metrics **9**
- 6.1 Notes, todos, questions . . . 9

1 Types

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 (`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 1.7 (`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 1.8 (`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 (`IntervalDomain`)**.** `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]`.

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`.[1]

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 2.3 (`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 2.4 (`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]`.

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

Definition 2.5 (`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 Notes, todos, questions

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

3 Traits

Definition 3.1 (`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).

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

Definition 3.2 (`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 3.3 (`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 3.4 (`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 3.5 (`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 3.6 (`InherentNull`). A type `T` has trait if and only if it can hold some value `null`.

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

Definition 3.7 (`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 $\texttt{i} + 1$.

Definition 3.8 (`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 3.9 (`One`). A type `T` has trait `One` if and only if `T` has some multiplicative identity element.

Definition 3.10 (`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 3.11 (`SaturatingAdd`)**.** A type `T` has trait `SaturatingAdd` if it performs addition that saturates at the numeric bounds instead of overflowing.

Definition 3.12 (`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 3.13 (`Zero`)**.** A type `T` has trait `Zero` if and only if `T` has some additive identity element.

3.1 Math-related definitions

(connor) Since these should probably have similar definitions, they are here for now (i.e. not alphabetized) since this is the first version, and it will be easier to make changes if they're all grouped together. They will be brought into the alphabetical list later.

Definition 3.14 (`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 3.15 (`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 3.16 (`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 3.17 (`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`.

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`

3.3 Notes, todos, questions

4 Functions

4.1 Functions in the pseudocode language

Definition 4.1 (`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 4.2 (`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 4.3 (`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 4.4 (`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 4.5 (`get_input_domain`). The function `get_input_domain(function)` returns the input domain of arguments passed to function `function`.

Definition 4.6 (`get_input_metric`). The function `get_input_metric(some_relation)` returns the input metric used by the relation `some_relation`.

Definition 4.7 (`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 $\texttt{i}+1$. The return value is of type `T`.

Definition 4.8 (`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 4.9 (`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 $\texttt{i}-1$. The return value is of type `T`.

Definition 4.10 (`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 4.11 (`get_output_domain`). The function `get_output_domain(function)` returns the output domain of values returned by function `function`.

Definition 4.12 (`get_output_metric`). The function `get_output_metric(some_relation)` returns the output metric used by the relation `some_relation`.

Definition 4.13 (`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 4.14 (`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 4.15 (`is_instance`)**.** The function `is_instance(var,T)` returns `True` if and only if the variable `var` is of type `T`.

Definition 4.16 (`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 4.17 (`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`. See https://doc.rust-lang.org/std/cmp/fn.max.html.

Definition 4.18 (`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.

4.2 Notes, todos, questions

- TODO: Add `data.iter().sum()`
- TODO: Add `sum.saturating_add()`
- TODO: Add `fold`

5 Classes

Definition 5.1 (`Transformation`)**.** We define a `Transformation` in the following way.

Question for reviewers: Which pseudocode style is preferred for this definition? With preconditions (section 5.0.1) or without preconditions (section 5.0.2)?

5.0.1 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
```

5.0.2 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.

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
```

6 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 6.1 (`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:** `T`

Definition 6.2 (`SymmetricDistance`)**.** The definition of *symmetric distance* in the "proof definitions" document tells how the distance between data is calculated.

- **Domains:** `VectorDomain(D)` where `D` is any domain
- **Associated type:** `u32`

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.

6.1 Notes, todos, questions