# Contents

# 1 Introduction

hedgerow is a small library for constructing polymorphic function on constrained sequences of arguments. This is inspired by a concept similar to variadic functions, called *hedges* or *hedge variables*-the "constrained sequences of arguments". This is one of the big differences in the library. All functions are functions on ***hedges***, where each parameter defines a domain and range. A polymorphic function is a single interface to functions on different domains. This is known as *ad hoc* polymorphism, or possibly more familiarly, *overloading* a function. Typically this is implemented over varying types:

```
append(numlistx: number[], numlisty: number[]) {...};

append(strlistx: string[], strlisty: string[]) {...};
```

The first signature expects two `number[]`, while the second expects two `string[]`. Depending on the arguments passed to `append` one of these two implementations may be chosen and invoked. The types, `number[]` and `string[]`, define domains to which the arguments must belong. Trying to pass an argument outside of the domains, may lead to an exception or unexpected behavior.

hedgerow goes beyond the type domain, by generalizing the way in which parameters, domains on arguments, can be defined. A parameter, as defined by the library, is an object with at least one of: {`in`, `requires`, `range`}. Operators, are implementations defined against a list of parameters, called a *region*, which is just a larger domain.

In mathematics, an operator or function, can generally be defined as a mapping of objects from one domain to another:

$$f : X \to Y$$

If $f$ is multivariate, then $X$ is the Cartesian product of the domains of each parameter:

$$\begin{aligned} f :& X \to Y, \text{ where} \\ & X \in (D_1 \times D_2 \times \cdots \times D_n) \end{aligned}$$

$n$ is the *arity* of the function $f$.

`append` above has two possible domains:

$$X \in \texttt{number}[] \times \texttt{number}[]$$
$$\text{OR}$$
$$X \in \texttt{string}[] \times \texttt{string}[]$$

In a context where functions are only allowed to take one (1) variable, e.g. lambda calculus, the argument of a multivariate function is a tuple $(x_1, x_2, ..., x_n)$,

where $x_i \in D_i$. hedgerow allows multivariate functions, but this is useful when thinking about the *region* of an operator vs. the *domain* of a parameter. Because each parameter also has a range, operators are functions on arrays of arguments (hedges). A polymorphic operator is a collection of ordinary operators, each providing an implementation for a specified region.

### 1.0.1 Example: Concatenation

```
import {Operator, PolymorphicOperator} from "hedgerow"

const con = new PolymorphicOperator();

con.implement(
  new Operator(
    new Parameter({
      in: "number",
    requires: n => n >= 0 && !(/[0-9]*\.[0-9]+/.test('' + n)),
    range: -1}),
  (nums: number[], [sep]: string[]) => parseInt(nums.join(sep))));

con.implement(
  new Operator(
  [new Parameter("string"),
   new Parameter({in: "string", range: -1})],
  ([sep]: string[], strs: string[]) => strs.join(sep)));

con.implement(
  new Operator(
  {requires: Array.isArray, range: -1},
  (objs: any[][]) => objs.flat()));

con.implement(
  new Operator(
    {in: Map, range: 2},
  ([l, r]: Map<any, any>[]) =>
    [...l.entries(), ...r.entries()]
      .reduce((map, [k, v]) => map.set(k, v), new Map())));
```

con is a polymorphic concatenation operator. It has implementations for the regions:

```
{
  in: "number",
  requires: n => n >= 0 && !(/[0-9]*\.[0-9]+/.test('' + n)),
  range: -1
}
```

zero or more positive integers followed by a string

- ["string", {in: "string", range: -1}] , string followed by zero or more string
- {requires: Array.isArray, range: -1} , zero or more object required to be an array
- {in: Map, range; 2} , two Map arguments

Depending on the given arguments, con will choose which implementation to invoke.

---

```
con([1,2,3,4]); //1234
con([" "], ["hello", "world"]); //'hello world'
con([[1, 2, 3], [4, 5, 6]]); //[1,2,3,4,5,6]
con([1.1, 2, 3, 4]); //Error no operator for arguments
con(["1", 2, 3, 4]); //Error no operator for arguments
con([{a: 1, b: 2}, {b: 3, c: 4}]); //Error: no operator for arguments

const [map1, map2] = [new Map(), new Map()];

map1.set("a", 1);
map1.set("b", 2);

map2.set("b", 3);
map2.set("c", 4);

con([map1, map2]); //Map(3) { 'a' => 1, 'b' => 3, 'c' => 4 }
```

---

**NB.:** Destructuring array arguments is one way of maintaining named arguments and readability, e.g. ([sep], strs) => strs.join(sep), when implementing functions on array arguments. This is particularly recommended for parameters that expect few arguments.

# 2 Interfaces & Types

The following is a list of type aliases and interfaces used in the library. They are mentioned here as a reference when reading through the documentation, which makes use of them without necessarily explaining them. The library is designed to be implemented against types listed here. "hedgerow/htypes".

---

| | |
|---|---|
| `HedgeFn` | `(...args: any[][]) => any)` |
| `Clause<T>` | `(x: T) => boolean` |
| `Constructor<T>` | `new (...args: any[]) => T` |
| `Maybe<T>` | `T \| null \| undefined` |
| `UnitOrArray<T>` | `T \| T[]` |
| `HedgeParams<N extends number, I extends HedgeFn>` | `Parameters<I>[N] extends (infer E)[] ? E : never` |
| `Hedge<I extends HedgeFn>` | `UnitOrArray< HedgeParams<number, I>[]>[]` |
| `Prototype<N extends number \| HedgeFn, I extends HedgeFn>` | `string \| N extends HedgeFn ? Constructor<ReturnType<N>> : Constructor<HedgeParams<N, I>>` |
| `HedgeDomain<N extends number, I extends HedgeFn>` | `UnitOrArray<Prototype<N, I>> \| HedgeDefinition<N, I>` |
| `HedgeRegion<N extends number, I extends HedgeFn>` | `UnitOrArray<HedgeDomain<N, I>> \| Parameter<N, I>` |
| `ReturnDomain<I extends HedgeFn>` | `UnitOrArray<Prototype<I>> \| Omit<HedgeDefinition, "range">` |

---

```
interface HedgeDefinition<N extends number, I extends HedgeFn> {
  in?: UnitOrArray<Prototype<N, I>>;
  requires?: Clause<HedgeParams<N, I>> | HedgeParams<N, I>[];
  range?: [min: number, max: number] | number
}
```

Another data structure used by the library, but not mentioned above is a `Trie` like structure. `PolymorphicOperator` leverages it to implement polymorphic functions, keying nodes on a `Parameter` in a signature. Terminal nodes are indicated by an implementation value. When a `PolymorphicOperator` instance is invoked, it's underlying Trie is searched, using the `Parameter.accepts` instance method to determine viable paths. This allows searching to fail fast, failing as soon as it can't find a parameter that accepts a given sequence of arguments, searching for an implementation goes into further details.

# 3 Parameters

```
class Parameter<N extends number, I extends HedgeFn>
```

A parameter defines the scope of a sequence of arguments. Each parameter has a `HedgeDefinition`, an object with three (3) optional properties:

- `in`
- `requires`
- `range`

When at least one of `type`, `require`, or `range` is given, any missing properties take their default values. When none of the constraints are given, the parameter is considered ***`void`***. Void parameters are the domain of nullary functions. A region can be a single void parameter or a list of defined parameters. Defining an operator with multiple void or a mix of void and defined parameters will raise an exception.

Notice that parameters can be defined specifically against an argument in an operator signature. `N` is the zero-based index of the parameter in the signature, `I`. This is useful for helping clauses infer the type of the argument passed in. It also restricts `type` to strings or object constructors placed on the argument in the signature.

## 3.1 Type Domain

`in` defines the type domain of the parameter. It is a single or union of object types, and defaults to "object". An object type is either:

- a `string` representing anything that can be returned by `typeof` or the name of an object constructor followed by an optional `!` (only this type) or `:` (only children of this type). This is a special syntax for controlling subtyping (another form of polymorphism) in functions.
- a constructor, (e.g. `Set`, `Map`)
- an array of `string` or constructors
  - this is equivalent to a union of types

## 3.2 Clause

`requires` defines a clause that must be met by each argument in the sequence. Defaults to `undefined` equivalent to `obj => true`. A clause is either

- a function with type `(x:any) => boolean`
- a list of allowed objects

> **NB.:** If the list is one of hashable objects-anything that can be contained in a `Set`, e.g. `string`, `number`, any immutable object- it might be faster to use `(x: T) => set.has(x)`, where `set` is a collection of `T` objects. This is because, the objects allowed by the parameter are not required to be hashable, and the underlying check is a clause that uses `Array.find`.

## 3.3 Range

`range` is a minimum and maximum value. Defaults to `[1, 1]`, `i.e. exactly 1 argument. It can be specified as either`

- a single positive integer or `-1`, indicating a fixed number or variadic number of arguments, respectively
- a tuple with a minimum and maximum, where the minimum is a positive integer and the maximum is `-1` or any integer greater than or equal to the minimum.

## 3.4 Constructing a Parameter

```
constructor(domain: Domain<N, I> = {})
```

A Parameter is constructed from a Domain object. If the domain only has the type property, then the value of ‘type‘ can be passed in as-is. This aligns with the default constraint on arguments being simply the type of the argument.

### 3.4.1 Example: Void, Optional, & Variadic Parameters

```
import {Parameter} from "hedgerow";

const nullary = new Parameter(); //or Parameter.Void

const optional = new Parameter({range: [0, 1]});

const variadic = new Parameter({range: -1});
```

### 3.4.2 Example: Array Parameter

```
const array = new Parameter({require: Array.isArray});
```

### 3.4.3 Example: Numeric Parameters

```
const atMost10NegativeNumbers = new Parameter({
  in: "number",
  requires: n => n < 0
  range: [0, 10]
});

const natural0 = new Parameter({
  in: ["number", "bigint"],
  requires: n => !(/[0-9]*\.[0-9]+/.test('' + n)) && n >= 0
});

type BinaryOp = ([n]: number[], [d]: number[]) => number[];

const denominator = new Parameter<1, BinaryOp>({
  in: "number",
  requires: d => d !== 0 //d has type ‘number’
});
```

### 3.4.4 Example: Subtyped Parameters

```
class Parent { }
class Child extends Parent { }

const [parent, child] = [new Parent(), new Child()];

const parentAndChildren = new Parameter(Parent);
const parentOnly = new Parameter("Parent!");
const childrenOnly = new Parameter("Parent:");
```

## 3.5 Testing Objects against a Parameter

```
accepts(args: UnitOrArray<HedgeParams<N, I>>): boolean;
```

All Parameter implementations have an accepts method for testing whether or not an object is in its domain. Generally objects are passed in as an array of objects with a length within the range of the parameter. But, for the sake of convenience, when testing a single object, it can be passed in as is and will be automatically wrapped in an array. The only exception is when the expected type accepts an array.

### 3.5.1 Example: accepts

```
nullary.accepts([1]) //false
nullary.accepts([]) //true

optional.accepts([]) //true
optional.accepts([1]) //true
optional.accepts([1, 2]) //false

variadic.accepts([]) //true
variadic.accepts([1,2]) //true
variadic.accepts([1,2,3,4,5,6,7]) //true

denominator.accepts(1); //true
denominator.accepts(0); //false
denominator.accepts([0.0001]); //true

atMost10NegativeNumbers.accepts(-1); //true
atMost10NegativeNumbers.accepts(1); //false
atMost10NegativeNumbers.accepts([]); //true
atMost10NegativeNumbers.accepts([-1, -2, -3, -4]); //true
atMost10NegativeNumbers.accepts([-3, -2, -1, 0]); //false

array.accepts({x: 1}); //false
array.accepts("string"); //false
array.accepts([]); //false -> zero arguments
array.accepts([[]]); //true -> empty array
array.accepts([[1, 2,3]]); //true

parentOnly.accepts(parent); //true
parentOnly.accepts(child); //false
parentAndChildren.accepts(parent); //true
parentAndChildren.accepts(child); //true
childrenOnly.accepts(parent); //false
childrenOnly.accepts(child); //true
```

## 3.6 Comparing Parameters

```
is(parameter: Domain<N, I> | Parameter<N, I>): boolean;
```

In order to construct polymorphic operators, parameters support comparison against one another. This is a deep comparison except when comparing clauses that are functions. Comparing two functional clauses is only true when they point to the same object (see the undecidability of the halting problem), even if it is obvious that two functions are equivalent. Parameters can be compared with each other using the `Parameter.is` instance method.

### 3.6.1 Example: `is`

```
optional.is(variadic); //false

nullary.is(Parameter.Void); //true
nullary.is(new Parameter({}); //true

natural0.is(natural0); //true
natural0.is("number"); //false
natural0.is({type: ["bigint", "number"], require: natural0.clause}); //tru

natural0.is({
  type: ["bigint", "number"],
  require: n => !(/[0-9]*\.[0-9]+/.test('' + n)) && n >= 0
}); //false

const num = new Parameter({type: "number"});

num.is("number") //true

const numOrStr = new Parameter({type: ["number", "string"]});

numOrStr.is(["string", "number"]) //true order doesn't matter
numOrStr.is({type: "boolean"}) //false
```

# 4 Operators

```
class Operator<I extends HedgeFn> implements Operator<I>
```

Operators are a combination of a region and an implementation. When an operator is invoked, the arguments are first checked against it's region before being passed to the underlying function.

Operators differ from traditional functions, in that they act on array arguments. For example, an operator expecting a `number` would implemented against a `number[]`. This is because parameters can accept a range of arguments. Operators can be constructed using the `Operator` class.

The region of an operator can be **void** or a list of defined parameter domains, none of which can be void. Operators, otherwise have no rules for the order, type or, amount of parameters of any type, e.g. an operator can support multiple variadic parameters, or have optional parameters proceeded by required parameters. Again, this is an advantage of implementing functions on array arguments.

Operators are also optionally self-validating. The constructor accepts an optional parameter-like constructor defining the domain of the returned value from the implementation.

## 4.1 Constructing an Operator

```
constructor(region: Region<number, I>, f: I, ensures?: ReturnDomain)
```

An operator is constructed using a single parameter or a list of parameters. All operators have at least one parameter in their region. If an operator is constructed with an empty array, this is equivalent to a nullary function. All nullary functions have a region consisting of a single void parameter.

### 4.1.1 Example: Constant function

```
const trueF = new Operator([], () => true);

const falseF = new Operator([Parameter.Void], () => false);

const oneF = new Operator(Parameter.Void, () => 1);
```

#### 4.1.2 Example: sum

```
const sum = new Operator(
  {in: "number", range: -1},
  (nums: number[]) => nums.reduce((x, y) => x + y));
```

This is an example that leverages the arguments being an array. Instead of implementing `sum` as taking an array of numbers to be added together- `f: ([nums]: number[][]) => ...)` - it is implemented as a variadic function accepting `number` values.

#### 4.1.3 Example: filter, reduce

```
const filter = new Operator(
  [{require: Array.isArray}, "function"],
  ([a]: any[][], [f]: Clause[]) => a.filter(f));

const reduce = new Operator(
  [{require: Array.isArray}, "function", {range: [0, 1]}],

 (array: any[][],
  [f]:((a: any, e: any) => any)[],
  [s]: any[]) => array.reduce(f, s));
```

## 4.2 Invoking an Operator

```
of(...args: Hedge<I>): ReturnType<I>;
```

Operators can be invoked using the `Operator.of` instance method, (taken from $f(x)$ being read aloud as "f of x". Operators are not naturally callable.

Notice that the arguments of an operator can be flattened in the call, and bound automatically to its region. This is only recommended when

- the domains of consecutive pairs of parameters in the region are exclusive and/or
- each parameter in the domain accepts exactly one parameter

### 4.2.1 Example: Exclusive Domains

```
const domain1 = ["number", "string", "number"]; //both criteria
const domain2 = ["number", "number"];//second criteria only

const domain3 = [
  {in: "number", requires: n => n < 0, -1}
  {in: "number", requires: n => n >= 0, -1}
]; //first criteria only
```

It is always safe to pass arguments in as hedges, while flattening can lead to unpredictable behavior if exclusivity in the region of the operator cannot be guaranteed.

### 4.2.2 Example: `filter`, `reduce`

```
//Hedged
//[1, 3, 5, 7, 9]
const filtered = filter.of([[1, 2, 3, 4, 5, 6, 7, 8, 9]],
                           [(n: number) => (n & 1) == 1]);

//Flattened -- "number[]" and "function" are exclusive domains
//             "function" and "number" are exclusive domains
//36
const reduced = reduce.of(...filtered,
                          (x: number, y: number) => x + y,
                          11)
```

## 4.3 Validating an Operator

Operators can be constructed with a parameter that validates the result of calling the function before returning it.

### 4.3.1 Example: filterNatural, reduceNatural

```
import {isnatural, optional} from "hedgerow/lib";

const filterNatural = new Operator(
  [{in: "number", requires: isnatural(1), range: -1}, "function"],

  (nums: number[], [f]: Clause<number>[]) => nums.filter(f)));

const reduceNatural = new Operator(
  [{in: "number", require: isnatural(1), range: -1},
   "function",
   optional({in: "number", requires: isnatural(1)})],

  (nums: number[],
   [f]:((a: number, e: number) => number)[],
   [s]: number[]) => nums.reduce(f, s),

  {requires: isnatural(1)});

//[1, 3, 5, 7, 9]
const filtered = filterNatural.of([1, 2, 3, 4, 5, 6, 7, 8, 9],
                                  [(n: number) => (n & 1) == 1]);

//25
const reduced = reduceNatural.of(...filtered,
                           (x: number, y: number) => x + y)
```

Notice the added `requires: isnatural(1)` to `reduceNatural`. This validates that the value returned from `reduceNatural` is greater than or equal to 1.

# 5 Polymorphic Functions

class PolymorphicOperator

A polymorphic function is a single entity to multiple implementations. Depending on the arguments passed in, one of the implementations is chosen, or an exception is raised if no operator can be found.

PolymorphicOperator is implemented as a Trie like structure with Parameter keys and an optional Operator value. Each parameter or node in a PolymorphicOperator is the potential root of another Trie structure. Each level in the structure represents each subsequent parameter in the region of an Operator .

For example, an operator with region ["string", "number"] will be stored at the second level of the the Trie. This design also allows operators with overlapping regions to share a common substructure in the operator, e.g. implementations with regions ["string", "number"] and ["string", "string"] will share a common node in the first level of the PolymorphicOperator. Specifically a node with key "string". That node, points to a child Trie with a "number" and "string" node, each with a value of their respective operator. Finding an operator for given arguments searches for a viable path, failing if no parameters at a given level accepts the hedge.

## 5.1 Constructing a Polymorphic Function

`constructor(...ops:Operator[])`

A polymorphic function is an empty structure optionally initialized with a list of operators.

### 5.1.1 Example: Pattern Matching

In this example we construct a callable polymorphic function based on the given patterns.

```
const isPosInt =
  (n: number) => n >= 0 && !(/[0-9]*\.[0-9]+/.test('' + n));

const fact = new PolymorphicOperator(
  new Operator(
    {in: "number", requires: n => isPosInt(n) && n < 2},
    n => 1),
  new Operator(
    "number",
    ([n]: number[]) => n * fact.of([n - 1]))
);
```

For inputs $0$ and $1$ the first implementation is chosen returning $1$, for all other inputs the second implementation is chosen. The **Polymorphic Functions** section goes into more details about this, but the order the functions are implemented, in this case, does not matter. This is because the implementation of `PolymorphicOperator` is highly opinionated about how it inserts an `Operator`. Generally parameters with clauses have a higher precedence than those without, i.e. adding a `requires` clause to the parameter of the second implementation, would preference it over the first, given their ranges are the same, and it was the most recent implementation.

This function isn't exhaustive. It doesn't handle **only** positive integers. The first parameter only accepts 0 or 1. Adding a `requires` clause to the second implementations' parameter, like `requires: isPosInt` would lead to an *exception* because the second implementation doesn't define a base case, (see previous mention of precedence). And since 0 and 1 pass its requirement, the function will recurse until `[-1]` is passed in, which is out of the region of both operators. If the goal is to restrict this function to only positive numbers, another operator handling negative integers could be added, or the second operator can be constrained to inputs greater than 1.

## 5.2 Implementing Operators in a Polymorphic Function

```
implement = <I extends HedgeFn = HedgeFn>(op: Operator<I>)
```

After constructing a `PolymorphicOperator`, additional operators can be added to the instance using the `implement` method. This invokes an opinionated process that attempts to place more specific parameters before more general ones. In general, parameters in each level of the `PolymorphicOperator` are sorted by `range`, and then whether or not the parameter is constrained. A *degenerate* parameter is one with a `range` where its minimum and maximum are equal. A *closed* parameter is one with a finite `range`. With that, precedence follows from:

1. Void parameter
2. Degenerate parameter with a `requires`
3. Degenerate parameter with no `requires`
4. Closed parameter with a `requires`
5. Closed parameter with no `requires`
6. Variadic parameter with a `requires`
7. Variadic parameter with no `requires`

> **NB.:** There is no notion of subtyping in this ordering. For example an implementation with domain `"object"` and another with `"object:"` have equal precedence. If the intent is for the latter implementation all subtypes of `"object"`, then it should be passed in before the implementation with domain `"object"`.

This precedence is what ultimately decides which operator gets invoked. It's best to avoid ambiguity in general, so arguments always get to their intended operators.

### 5.2.1 Example: PolymorphicOperator Test Function

```
import {op, optional} from "hedgerow/lib";

const opts = op(
  [optional(), optional(), optional()],
  (_, __, ___) => "OPTIONAL OPTIONAL OPTIONAL");

const numnum = op(
  ["number", "number"],
  (_, __) => "NUMBER NUMBER");

const posnumnum op(
  [{in: "number", requires: n => n >= 0}, "number"],
  (_, __) => "POSITIVE NUMBER NUMBER");

const strnum = op(
  ["string", "number"],
  (_, __) => "STRING NUMBER");

const numstr = op(
  ["number", "string"],
  (_, __) => "NUMBER STRING");

const strstr = op(
  ["string", "string"],
  (_, __) => "STRING STRING" );

const f = new PolymorphicOperator(opts, numnum, strnum, numstr, strstr);

f.implement(posnumnum);
```

`opts` has the lowest precedence compared to the others. `posnumnum` has the highest precedence, and the remaining functions have equal precedence. In this operator, `posnumnum` intercepts and handles any arguments valid for `numnum` if the first argument is a positive number.

## 5.3 Invoking a Polymorphic Function

```
of = <R = any>(...args: any[][]): R
```

Like the `Operator`, the `PolymorphicOperator` is invoked using the `of` method. But unlike `Operator`, the arguments **must** be hedged. This is because polymorphic functions have no notion of the available regions. Operators are found based on previously bound arguments.

A special case is held for what is considered a ***void call***. If every hedge is empty the call is equivalent to an empty call. The operator searches for the first implementation that is either *void*, or every parameter is optional. This intentionally makes, for example `f = () => ...` and `opts` from above, ambiguous in their domains.

### 5.3.1 Example: Invoking the PolymorphicOperator Test Function

```
f.of([]) //"OPTIONAL OPTIONAL OPTIONAL"
f.of([], []) //"OPTIONAL OPTIONAL OPTIONAL"
f.of([], [], [], []) //"OPTIONAL OPTIONAL OPTIONAL" !
f.of([1], [2], [3]) //"OPTIONAL OPTIONAL OPTIONAL"
f.of([false]) //"OPTIONAL OPTIONAL OPTIONAL"
f.of([], [], [], [0]) //Error: "no operator for arguments"

f.of([-1], [0]) //"NUMBER NUMBER"
f.of([1], [2])) //"POSITIVE NUMBER NUMBER"
f.of([1], ["string"]) //"NUMBER STRING"
f.of(["string"], [1]) //"STRING NUMBER"
f.of(["string"], ["string"]) //"STRING STRING"
```

# 6 The Support Library

Along with Parameter, Operator, and PolymorphicOperator, the library exposes some support functions. These functions help understand how the library functions at a lower level, by exposing many of the processes used by the classes. For example, calling an operator with *flattened* arguments invokes the bind function which takes a list of flattened arguments and a region, and returns **valid** hedged arguments. If binding fails, an exceptions is raised.

---

```
accepts = <I extends HedgeFn>(
  hedges: HedgeParams<number, I>[][],
  region: Parameter<number, I>[]): boolean
```

---

Checks a given list of arguments against a region.

Parameters

hedges: HedgeParams<number, I>[][]

An array of array arguments

region: Parameter<number, I>[]

An array of parameters

Returns

true if every parameter in the region accepts its given array of arguments, false otherwise.

---

```
ancestry = (obj: any): string[]
```

---

Gets all of the parent types of the object, but does not include the type of the object itself.

Parameters

obj: any

The object to get the types for

Returns

a string[] of parent types if any exist

```
bind = <I extends HedgeFn>(
  args: HedgeParams<number, I>[],
  region: Parameter<number, I>[]): HedgeParams<number, I>[][]
```

Collects an array of arguments into hedges corresponding to a given region.

Parameters

`args: HedgeParams<number, I>[]`

An array of arguments

`region: Parameter<number, I>[]`

An array of parameters

Returns

Hedged parameters corresponding to the given region.

Throws

When the arguments cannot be grouped according to the given region

```
domain = <N extends number, I extends HedgeFn>(
  definition: HedgeDefinition<N, I>): HedgeDefinition<N, I>
```

Standardizes a domain definition by adding in any missing properties as default values and ensuring the range is a valid tuple in the returned definition.

Parameters

`definition: HedgeDefinition<N, I>`

The definition to standardize

Returns

The definition as is if its empty or already in the standard form, otherwise a standard form definition is returned.

```
isclosed = <N extends number, I extends HedgeFn>(
  domain: Parameter<N, I>): boolean
```

Whether or not a parameter has a range that is neither degenerate, nor open.

Parameters

domain: Parameter<N, I>

the parameter to check

Returns

true if the minimum of the range does not equal the maximum, and the maximum $\neq -1$, false otherwise.

---

```
isdegenerate = <N extends number, I extends HedgeFn>(
  domain: Parameter<N, I>): boolean
```

---

Whether or not a parameter has a range where the minimum equals the maximum.

Parameters

`domain: Parameter<N, I>`

the parameter to check

Returns

`true` if the minimum of the range equals the maximum, `false` otherwise.

---

```
isinteger = (n: number): boolean
```

---

Whether or not a given number is an integer

Parameters

`n: number`

the number to check

Returns

`true` if the number is an integer, `false` otherwise.

---

```
isnatural = (base: 0 | 1 = 1): ((n: number) => boolean)
```

---

Returns a function that checks if a number is an integer and greater than or equal
to the given base.

Parameters

`base: 0 | 1`

the base of the natural numbers

Returns

A function that return `true` if the number is greater than or equal to the given base, `false` otherwise.

```
isopen = <N extends number, I extends HedgeFn>(
  domain: Parameter<N, I>): boolean
```

Whether or not a parameter is variadic.

Parameters

`domain: Parameter<N, I>`

the parameter to check

Returns

`true` if the maximum is $-1$, `false` otherwise.

```
isvoid = <N extends number, I extends HedgeFn>(
  domain: Parameter<N, I>): boolean
```

Whether or not a parameter accepts no parameters.

Parameters

`domain: Parameter<N, I>`

the parameter to check

Returns

`true` if the minimum and maximum are 0, `false` otherwise.

```
isinstance = (obj: any, ...types: Prototype[]): boolean
```

Whether or not an object is an instance of one of the given types

Parameters

`obj: any`

the parameter to check

`types: Prototype[]`

a list of types to check the object against

Returns

`true` if the object is an instance of one of the given types, `false` otherwise.

---

`issubtype = (obj: any, ...types: Prototype[]): boolean`

---

Whether or not an object is an instance or subtype of one of the given types

Parameters

`obj: any`

the parameter to check

`types: Prototype[]`

a list of types to check the object against

Returns

`true` if the object is an instance or subtype of one of the given types, `false` otherwise.

---

`lineage = (obj: any): string[]`

---

Gets the type of the object and all of the parent types.

Parameters

`obj: any`

The object to get the types for

Returns

a `string[]` with the type of the object and its parent types if any exist

```
op = <I extends HedgeFn = HedgeFn>(
  region: HedgeRegion<number, I>,
  f: I): HedgeFunction<I>
```

Wraps an `Operator` instance as a callable operator patching to its `of` method.

Has a property `implementation` for accessing the underlying `Operator` instance.

Parameters

`region: Region<number, I>`

An array of parameters defining the domain of the operator

`f: I`

The implementation of the operator

Returns

A callable operator

```
type DomainNoRange = UnitOrArray<Prototype<N, I>>
                   | Omit<HedgeDefinition<N, I> "range">;

optional = <N extends number, I extends HedgeFn>(
  domain: DomainNoRange = "object",
  max: number = 1): Parameter<N, I>
```

Constructs an optional parameter.The parameter will have a range with a minimum equal to 0 and a maximum greater than 0.

Parameters

`domain: DomainNoRange`

the constraints of the parameter without specifying the range.

`max: number`

a number greater than 0

Returns

A parameter with a range where its minimum is 0 and its maximum is greater than 0

```
type DomainNoType = Pick<HedgeDefinition, "requires" | "range">;

pattern = (
  type: UnitOrArray<Prototype>,
  patterns: [DomainNoType, HedgeFn][]): CallableHedgeFunction
```

---

Constructs a callable polymorphic operator from the given patterns. All patterns are applied to parameters of the same type, i.e. all parameters will have the same `in` property.

Parameters

`type: UnitOrArray<Prototype>`

the type domain for all the patterns

`patterns: [DomainNoType, HedgeFn][]`

A list of domain-implementation pairs. The domain specifies any `requires` and `range` properties.

Returns

A callable polymorphic operator reflecting the given pattern.

```
type FnLike = Operator | HedgeFunction;

polyop = (...fns:FnLike[]): CallableHedgeFunction
```

Constructs a callable polymorphic operator from the given implementations.

Parameters

fns: FnLike[]

the operators to initialize the polymorphic function with.

Returns

A callable polymorphic operator initialized with the given operators.

```
region = <N extends number, I extends HedgeFn>(
  region: Region<N, I>): UnitOrArray<Parameter<N, I>>
```

Given a parameter or list of parameters specified in any format, returns the parameter as or a list where each parameter is a Parameter, if it is not one already.

Parameters

region: Region<N, I>

a parameter or list of parameters to transform

Returns

The parameter as or a list where each parameter is a Parameter

```
type DomainNoRange = UnitOrArray<Prototype<N, I>>
                   | Omit<HedgeDefinition<N, I> "range">;

variadic = <N extends number, I extends HedgeFn>(
  type: DomainNoRange = "object",
  min: number = 0): Parameter<N, I>
```

Constructs a variadic parameter.The parameter will have a range with a minimum greater than or equal to 0 and a maximum equal to -1.

Parameters

`domain: DomainNoRange`

the constraints of the parameter without specifying the range.

`max: number`

a number greater than or equal to 0

Returns

A parameter with a range where its minimum is greater than or equal to 0 and its maximum is -1.