Package: anvl (via r-universe)

June 3, 2026

Title Accelerated Array Computing and Automatic Differentiation

Version 0.3.0

Description Accelerated array computing and code transformations for R. Numerical programs operating on multi-dimensional arrays can be just-in-time compiled to optimized executables via 'XLA' -- the same compiler that powers 'JAX' and 'TensorFlow' -- and run on CPU or NVIDIA GPU from the same source. Also provides reverse-mode automatic differentiation, returning the gradient of a function as another R function.

License MIT + file LICENSE

URL https://r-xla.github.io/anvl/, https://github.com/r-xla/anvl

BugReports https://github.com/r-xla/anvl/issues

Depends R (>= 4.2.0)

Imports checkmate, cli, methods, pjrt (>= 0.4.0), rlang, safetensors (>= 0.2.0), stablehlo (>= 0.3.0), tengen (>= 0.2.0), withr, xlamisc (>= 0.3.0)

Suggests knitr, quarto, rmarkdown, quickr (>= 0.3.0), testthat (>= 3.0.0)

Additional_repositories https://r-xla.r-universe.dev

VignetteBuilder knitr

Config/testthat/edition 3

Encoding UTF-8

Roxygen list(markdown = TRUE)

RoxygenNote 7.3.3

Collate 'aaa.R' 'array.R' 'device.R' 'backend.R' 'jit.R' 'primitive.R' 'type-converters.R' 'utils.R' 'primitives.R' 'api.R' 'api-generics.R' 'api-like.R' 'api-rng.R' 'api-subset.R' 'api-utilities.R' 'asserts.R' 'backend-quickr.R' 'backend-xla.R' 'bibentries.R' 'box.R' 'flatten.R' 'graph-passes.R' 'graph.R' 'graph-print.R' 'rules-quickr.R'

'graph-to-quickr.R' 'promotion.R' 'quickr.R' 'reexports.R'
'reverse.R' 'rules-reverse.R' 'rules-stablehlo.R' 'serialize.R'
'stablehlo.R' 'zzz.R'

Config/pak/sysreqs cmake make libprotobuf-dev libuv1-dev protobuf-compiler libprotoc-dev

Repository https://r-xla.r-universe.dev

Date/Publication 2026-06-03 12:04:30 UTC

RemoteUrl https://github.com/r-xla/anvl

RemoteRef v0.3.0

RemoteSha 76f4c30cc07b5fd8f3a99f3e5b937d787f3d196c

Contents

`anvl-package` *anvl: Framework for R code transformations*

Description

Code transformation framework for R.

Third-Party Licenses

The anvl package itself is MIT-licensed. The CUDA backend dynamically loads NVIDIA software which is not bundled with anvl, but downloaded from NVIDIA's official redistributable channels by the CUDA toolkit R package (e.g. cuda12.8) at install time. Its use is governed by the NVIDIA CUDA Toolkit EULA, with the exception of cuDNN, which is covered by the NVIDIA cuDNN SLA, and NCCL, which is covered by its own license. By installing or using the CUDA backend you accept those terms.

Author(s)

Maintainer: Sebastian Fischer <seb.fischer@tutamail.com> (ORCID)

Authors:

- Daniel Falbel <daniel@posit.co> (ORCID)
- Tomasz Kalinowski <tomasz@posit.co>
- Nikolai German <niko.german@gmail.com> (ORCID)

See Also

Useful links:

- https://r-xla.github.io/anvl/
- https://github.com/r-xla/anvl
- Report bugs at https://github.com/r-xla/anvl/issues

.current_descriptor *Get the current graph*

Description

Get the current graph being built (via local_descriptor).

Usage

```
.current_descriptor(silent = FALSE)
```

Arguments

silent	(logical(1)) Whether to return NULL if no graph is currently being built (as opposed to aborting).

Value

A GraphDescriptor object.

`[.AnvlArray` *Subset an Array*

Description

Extracts a subset from an array. You can also use the `[` operator. Supports R-style indexing including scalar indices (which drop dimensions), ranges (`a:b`), and `array(c(...))` for selecting multiple elements along a dimension.

Usage

```
## S3 method for class 'AnvlArray'
x[...]

nv_subset(operand, ...)
```

Arguments

`x`	(`arrayish`) Same as `operand`; this is the name used by the base R S3 generic.
`...`	Subset specifications, one per dimension. Omitted trailing dimensions select all elements. See `vignette("subsetting")` for details.
`operand`	(`arrayish`) Operand.

Value

`arrayish`

See Also

`nv_subset_assign()` for updating subsets, `vignette("subsetting")` for a comprehensive guide.

Examples

```
x <- nv_matrix(1:12, nrow = 3)
x
# Select row 2
x[2, ]

# Select rows 1 to 2, all columns
x[1:2, ]
```

`[<-.AnvlArray` *Update Subset*

Description

Updates elements of an array at specified positions, returning a new array. You can also use the `[<-` operator.

Usage

```
## S3 replacement method for class 'AnvlArray'
x[...] <- value

nv_subset_assign(operand, ..., value)
```

Arguments

`x`	(arrayish) Same as `operand`; this is the name used by the base R S3 generic.
`...`	Subset specifications, one per dimension. See `vignette("subsetting")` for details.
`value`	(arrayish) Replacement values. Scalars are broadcast to the subset shape. Non-scalar values must match the subset shape.
`operand`	(arrayish) Operand.

Value

arrayish
A new array with the same shape as `operand` and the subset replaced.

See Also

`nv_subset()`, `vignette("subsetting")` for a comprehensive guide.

Examples

```
x <- nv_matrix(1:12, nrow = 3)
# Set row 1 to zeros
x[1, ] <- nv_scalar(0L)
x
```

abstract_properties *Abstract Properties*

Description

Calls the extractor after converting the input to an AbstractArray.

Usage

```
shape_abstract(x)

ndims_abstract(x)

dtype_abstract(x)

ambiguous_abstract(x)
```

Arguments

x	(arrayish)

ambiguous *Get Ambiguity of an Array*

Description

Returns whether the array's dtype is ambiguous.

Usage

```
ambiguous(x, ...)
```

Arguments

x	An array object
...	Additional arguments (unused)

Value

logical(1) - TRUE if the dtype is ambiguous, FALSE otherwise

AnvlArray *AnvlArray*

Description

The main array object. Its type is determined by a data type and a shape.

Usage

```
nv_array(
  data,
  dtype = NULL,
  device = NULL,
  shape = NULL,
  ambiguous = NULL,
  backend = NULL,
  byrow = FALSE
)

nv_scalar(data, dtype = NULL, device = NULL, ambiguous = NULL, backend = NULL)

nv_matrix(
  data,
  nrow = NULL,
  ncol = NULL,
  dtype = NULL,
  device = NULL,
  ambiguous = NULL,
  backend = NULL,
  byrow = FALSE
)

nv_empty(dtype, shape, device = NULL, ambiguous = FALSE)

nv_array_like(
  like,
  data,
  dtype = NULL,
  device = NULL,
  shape = NULL,
  ambiguous = NULL,
  backend = NULL
)

nv_scalar_like(
  like,
  data,
```

```
  dtype = NULL,
  device = NULL,
  ambiguous = NULL,
  backend = NULL
)

nv_empty_like(
  like,
  dtype = NULL,
  shape = NULL,
  device = NULL,
  ambiguous = NULL
)
```

Arguments

data	(any) `integer()`, `double()`, or `logical()` scalar, vector, or array.
dtype	(`NULL` | `character(1)` | `tengen::DataType`) One of bool, i8, i16, i32, i64, ui8, ui16, ui32, ui64, f32, f64 or a `tengen::DataType`. The default (`NULL`) uses the current backend's default dtype: `f32` for numeric data on `"xla"`, `f64` for numeric data on `"quickr"`, `i32` for integer data, and `bool` for logical data.
device	(`NULL` | `character(1)` | `PJRTDevice`) The device for the array (`"cpu"`, `"cuda"`). Default is to use the CPU for new arrays. This can be changed by setting the `PJRT_PLATFORM` environment variable.
shape	(`NULL` | `integer()`) The output shape of the array. The default (`NULL`) is to infer it from the data if possible. Note that `nv_array` interprets length 1 vectors as having shape `(1)`. To create a "scalar" with dimension `()`, use `nv_scalar` or explicitly specify `shape = c()`.
ambiguous	(`NULL` | `logical(1)`) Whether the dtype should be marked as ambiguous. Defaults to `FALSE` for new arrays.
backend	(`NULL` | `character(1)`) Backend to use (`"xla"` or `"quickr"`). Defaults to `default_backend()`. Must not be specified inside `jit()`.
byrow	(`logical(1)`) When constructing from an R object and the result has at least two dimensions, fill the array in row-major order rather than the default column-major order, mirroring `base::matrix()`'s `byrow`. Only allowed when `data` is an R object — passing an existing `AnvlArray` together with `byrow = TRUE` is an error.
nrow	(`NULL` | `integer(1)`) Number of rows. Inferred from `ncol` and the data length if `NULL`. Defaults to `1` when `data` is a scalar.

ncol (NULL | integer(1))
Number of columns. Inferred from nrow and the data length if NULL. Defaults to 1 when data is a scalar.

like (AnvlArray)
An existing array. Any of dtype, device, shape, ambiguous, and backend that are NULL (the default) are taken from like.

Value

(AnvlArray)

Extractors

The following generic functions can be used to extract information from an AnvlArray:

- dtype(): Get the data type of the array.
- shape(): Get the shape (dimensions) of the array.
- ndims(): Get the number of dimensions.
- device(): Get the device of the array.
- platform(): Get the platform (e.g. "cpu", "cuda").
- ambiguous(): Get whether the dtype is ambiguous.

Serialization

Arrays can be serialized to and from the safetensors format:

- nv_save() / nv_read(): Save/load arrays to/from a file.
- nv_serialize() / nv_unserialize(): Serialize/deserialize arrays to/from raw vectors.

See Also

nv_fill, nv_iota, nv_seq, as_array, nv_serialize

Examples

```
# A 1-d array (vector) with shape (4). Default type for integers is `i32`
nv_array(1:4)

# Specify a dtype
nv_array(c(1.5, 2.5, 3.5), dtype = "f64")

# A 2x3 matrix
nv_array(1:6, shape = c(2L, 3L))

# A 2x3 matrix filled by row, like `matrix(1:6, 2, 3, byrow = TRUE)`.
nv_array(1:6, shape = c(2L, 3L), byrow = TRUE)

# A scalar array.
nv_scalar(3.14)
```

```
# A 0x3 array
nv_empty("f32", shape = c(0L, 3L))

# --- Extractors ---
x <- nv_array(1:6, shape = c(2L, 3L))
dtype(x)
shape(x)
ndims(x)
device(x)
platform(x)
ambiguous(x)

# --- Transforming arrays with jit ---
add_one <- jit(function(x) x + 1)
add_one(nv_array(1:4))

# --- Eager mode (calling operations directly) ---
nv_add(nv_array(1:3), nv_array(4:6))
```

`AnvlBackendQuickr` *Quickr backend*

Description

Constructs the quickr backend, which stores array data as plain R arrays and compiles jitted functions to R code via the quickr package.

Usage

```
AnvlBackendQuickr()
```

Details

To use it, the `"quickr"` package needs to be installed.

Registered automatically under the name `"quickr"` when the package is loaded; call `local_backend("quickr")` or `with_backend("quickr", ...)` to use it. Requires the quickr package to be installed.

Value

An `AnvlBackend` object with subclass `"AnvlBackendQuickr"`.

Data representation

An `AnvlArray` with `backend = "quickr"` is, under the hood, a plain R vector or array (`numeric`, `integer`, or `logical`) stored in the `$data` field. `as_array()` returns the underlying vector/array directly without copying, and `nv_array()` simply wraps an R vector/array. As a consequence, there is no separate notion of a device: data always lives in R's memory and computation always runs on the CPU.

Status

This backend is **experimental** and has a number of limitations:

- Compilation (tracing + quickr lowering) is somewhat slow, so it is best suited to long-running or repeatedly-called functions where the one-time compilation cost is amortized.
- Only a subset of the primitives that the XLA backend supports are currently lowered to quickr code. See `vignette("primitives")` for an overview.
- Only the data types `f64`, `i32`, and `bool` are supported.
- Only CPU execution is supported.

Quickr JIT arguments

- `unwrap` (`logical(1)`, default `FALSE`): if `TRUE`, the compiled function returns plain R arrays instead of `AnvlArray`s. Useful when the jitted function's output is consumed by non-anvl R code and the extra wrapping would only get stripped again.

See Also

`AnvlBackend()`, `AnvlBackendXla()`, `local_backend()`, `jit()`.

`AnvlBackendXla` *XLA backend*

Description

Constructs the XLA backend, which stores array data in PJRT buffers (via `pjrt::pjrt_buffer()`) and compiles jitted functions to XLA executables via `stablehlo()` and `pjrt::pjrt_compile()`. This is the default backend.

Usage

```
AnvlBackendXla()
```

Value

An `AnvlBackend` object with subclass `"AnvlBackendXla"`.

Data representation

An `AnvlArray` with `backend = "xla"` wraps a `pjrt::pjrt_buffer()` stored in the `$data` field. The buffer owns the memory holding the tensor values and may live on any device supported by PJRT (CPU, CUDA, Metal, ...). Calling `as_array()` transfers the buffer contents back to an R array; calling `nv_array()` on an R object uploads it to the requested device.

Each `AnvlArray` therefore has an associated device, queryable via `device()`. A device is a `pjrt::as_pjrt_device()` object (e.g. the platform `"cpu"` or `"cuda"`, optionally with an index such as `"cuda:1"`). When `device` is `NULL` in `nv_array()` or the `jit()` wrapper, the device defaults to the `PJRT_PLATFORM` environment variable (falling back to `"cpu"`), or is inferred from the existing inputs of a jitted call. Operations require all inputs to live on the same device.

XLA JIT arguments

- `donate` (`character()`, default `character()`): names of arguments whose underlying buffers may be donated to (i.e., reused/consumed by) the compiled XLA executable. Donated buffers must not be used again by the caller after the call; this can reduce memory usage and copies for large inputs. Must not overlap with `static`.

See Also

`AnvlBackend()`, `AnvlBackendQuickr()`, `local_backend()`, `jit()`.

AnvlBox *AnvlBox*

Description

Virtual S3 base class for `GraphBox`.

See Also

GraphBox

AnvlGraph *Graph of Primitive Calls*

Description

Computational graph consisting exclusively of primitive calls. This is a mutable class.

Usage

```
AnvlGraph(
  calls = list(),
  in_tree = NULL,
  out_tree = NULL,
  inputs = list(),
  outputs = list(),
  constants = list(),
  is_static_flat = NULL,
  static_args_flat = NULL
)
```

Arguments

calls	(list(PrimitiveCall)) The primitive calls that make up the graph.
in_tree	(NULL | Node) The tree of inputs. May contain leaves for both array inputs and static (non-array) arguments. Only the array leaves correspond to entries in inputs; use is_static_flat to distinguish them.
out_tree	(NULL | Node) The tree of outputs.
inputs	(list(GraphValue)) The inputs to the graph (array arguments only).
outputs	(list(GraphValue)) The outputs of the graph.
constants	(list(GraphValue)) The constants of the graph.
is_static_flat	(NULL | logical()) Boolean mask indicating which flat positions in in_tree are static (non-array) args. NULL when all args are array inputs.
static_args_flat	(NULL | list()) Flattened traced values for the static arguments indicated by is_static_flat.

Value

(AnvlGraph)

AnvlPrimitive *AnvlPrimitive*

Description

Primitive interpretation rule. Note that [[and [[<- access the interpretation rules. To access other fields, use $ and $<-.

A primitive is considered higher-order if it contains subgraphs.

Usage

```
AnvlPrimitive(name, subgraphs = character())
```

Arguments

name	(character()) The name of the primitive.
subgraphs	(character()) Names of parameters that are subgraphs. Only used if higher_order = TRUE.

Value

(AnvlPrimitive)

---

arr *Create an R array*

Description

Create an R array without having to wrap data in c()

Usage

```
arr(..., shape = NULL)
```

Arguments

...	(any) Values of new array.
shape	(NULL | integer()) Shape of new array. If NULL (default), uses length of elements to create a 1D array.

Examples

```
arr(1, 2, 3)
arr(1, 2, 3, 4, shape = c(2, 2))
```

---

arrayish *Array-like Objects*

Description

A arrayish value is any object that can be input to a primitive such as prim_add.

During runtime of a JIT-compiled function, these are AnvlArray objects.

The following types are arrayish (during tracing):

- AnvlArray: a concrete array holding data on a device.
- GraphBox: a boxed abstract array representing a value in a graph.
- Length-1 vectors: numeric(1) and logical(1)
- R arrays of types: numeric and logical.

Use is_arrayish() to check whether a value is arrayish.

Usage

```
is_arrayish(x, convert_ok = TRUE)
```

Arguments

x	(any) Object to check.
convert_ok	(logical(1)) Whether to accept numeric(1) and logical(1) and R arrays of type numeric and logical.

Value

logical(1)

See Also

AnvlArray, GraphBox

Examples

```
# AnvlArrays are arrayish
is_arrayish(nv_array(1:4))

# Scalar R literals are arrayish by default
is_arrayish(1.5)
# R arrays are arrayish by default
is_arrayish(array(1.5))

# R arrays
is_arrayish(array(1:4), convert_ok = TRUE)
is_arrayish(array(1:4), convert_ok = FALSE)

# Length 1 vectors
is_arrayish(1.5, convert_ok = FALSE)
is_arrayish(1.5, convert_ok = TRUE)
```

---

as-AnvlArray *Coerce AnvlArray to an R Vector*

Description

Convert an AnvlArray to a bare R vector. The array's shape is discarded; the result is always a flat vector. Each method requires a compatible dtype:

- as.double() / as.numeric(): float or (signed/unsigned) integer dtypes.
- as.integer(): signed or unsigned integer dtypes.

- `as.logical()`: `bool`.
- `as.vector()`: any dtype; the R type is chosen by the dtype, or forced via `mode` (e.g. `"integer"`, `"double"`, `"logical"`, `"list"`).

Use `as_array()` to obtain an R array that preserves the shape, or `nv_convert()` to change the dtype of an `AnvlArray` before coercing.

Usage

```
## S3 method for class 'AnvlArray'
as.double(x, ...)

## S3 method for class 'AnvlArray'
as.integer(x, ...)

## S3 method for class 'AnvlArray'
as.logical(x, ...)

## S3 method for class 'AnvlArray'
as.vector(x, mode = "any")
```

Arguments

`x`	(`AnvlArray`) Array to coerce.
`...`	Unused.
`mode`	(`character(1)`) For `as.vector()` only. See `base::as.vector()`. Defaults to `"any"`, meaning the natural R type for the array's dtype.

Value

An R vector of the corresponding type (`double`, `integer`, or `logical`).

Examples

```
x <- nv_array(c(1.5, 2.5, 3.5, 4.5), shape = c(2L, 2L))
as.numeric(x)
as.integer(nv_array(1:6, shape = c(2L, 3L)))
as.logical(nv_array(c(TRUE, FALSE), dtype = "bool"))
as.vector(x)
```

as_anvl_array *Convert to AnvlArray*

Description

Use this to canonicalize inputs at the start of a function so it works both with eager executing and in combination with `jit()`. Use `as_anvl_array()` for a single input and `as_anvl_arrays()` for multiple inputs. The latter will also ensure all arrays are from the same backend and live on the same device.

Usage

```
as_anvl_array(x, device = NULL)

as_anvl_arrays(...)
```

Arguments

x	(arrayish) Input to standardize.
device	(NULL | device) Target device. If x is an AnvlArray on a different device, an error is raised.
...	(arrayish) Inputs to align.

Details

During tracing, boxes are returned as is. During eager mode, R literals and arrays are converted to AnvlArrays on the specified device. For AnvlArray inputs, we check that they live on provided device (if specified).

Value

(AnvlArray for `as_anvl_array()`, list of AnvlArrays for `as_anvl_arrays()`).

Examples

```
as_anvl_array(1L)
as_anvl_arrays(nv_array(1:3), 1L)
```

as_array *Convert to an R array*

Description

Transfers array data to R and returns it as an R `array`. Only in the case of scalars is the result a vector of length 1, as R `arrays` cannot have 0 dimensions.

Usage

```
as_array(x, ...)
```

Arguments

`x`	(`arrayish`) An array-like object.
`...`	Additional arguments passed to methods (unused).

Details

This is implemented via the generic `tengen::as_array()`.

Value

An R `array` or `vector` of length 1.

Examples

```
x <- nv_array(1:4, dtype = "f32")
as_array(x)
y <- nv_scalar(1L)
# R arrays can't have 0 dimensions:
as_array(y)
```

as_dtype *Convert to a DataType*

Description

Coerces a value to a `DataType`. Accepts data type strings (e.g. `"f32"`, `"i64"`, `"bool"`) or existing `DataType` objects (they are returned unchanged).

Usage

```
as_dtype(x)
```

Arguments

x	A character string or DataType to convert.

Details

This is implemented via the generic tengen::as_dtype().

Value

A DataType object.

See Also

is_dtype(), tengen::as_dtype(), tengen::DataType

Examples

```
as_dtype("f32")
as_dtype("i32")
```

---

as_raw *Convert an array to a raw vector*

Description

Returns the underlying bytes of an array as a raw vector.

Usage

```
as_raw(x, ...)
```

Arguments

x	(arrayish) An array-like object.
...	Additional arguments passed to method: • row_major (logical(1)) Whether to write the bytes in row-major order.

Details

This is implemented via the generic tengen::as_raw().

Value

A raw vector.

Examples

```
x <- nv_array(1:4, shape = c(2, 2), dtype = "f32")
as_raw(x, row_major = TRUE)
as_raw(x, row_major = FALSE)
```

---

at2vt *Convert AbstractArray to ValueType*

Description

Convert an `AbstractArray` to a `ValueType`.

Usage

```
at2vt(x)
```

Arguments

`x`	(`AbstractArray`)

Value

(`ValueType`)

---

await *Block until an async operation completes*

Description

Block until the array's underlying computation has finished, and return the object invisibly. Useful for benchmarking, where the dispatch of an asynchronous operation should not be confused with its execution.

Usage

```
await(x, ...)
```

Arguments

`x`	(`AnvlArray` or other awaitable) An object with an `await()` method.
`...`	Additional arguments passed to methods (unused).

Details

Implemented via the generic `pjrt::await()`. For backends without asynchronous execution (e.g. `"quickr"`), this is a no-op.

Value

`x`, invisibly.

See Also

`pjrt::await()`, `map_tree()` (to await a tree of outputs)

Examples

```
x <- nv_array(1:4, dtype = "f32")
await(x)

# Await all leaves of a (possibly nested) list of arrays.
map_tree(list(x, list(y = x)), await)
```

---

backend *Get Backend of an Array*

Description

Get Backend of an Array

Usage

```
backend(x, ...)
```

Arguments

`x`	An array object
`...`	Additional arguments (unused)

Value

`character(1)` - the backend name

build_tree *Build Tree*

Description

Captures the nesting structure of an object as a tree of Nodes. Each leaf in the input becomes a LeafNode with an integer index corresponding to its position in the flat list produced by flatten(). Lists become ListNodes that record child nodes and names. The resulting tree can be passed to unflatten() to reconstruct the original structure from a flat list.

Usage

```
build_tree(x, counter = NULL)
```

Arguments

x	(any) Object whose structure to capture. Lists are recursed into; everything else is a leaf.
counter	(NULL | environment) Internal counter for assigning leaf indices. Mostly used internally and otherwise left as NULL (default.)

Value

A Node (LeafNode for scalars, ListNode for lists).

See Also

flatten(), unflatten(), tree_size(), reindex_tree()

Examples

```
x <- list(a = 1, b = list(c = 2, d = 3))
tree <- build_tree(x)
tree_size(tree)

flat <- flatten(x)
unflatten(tree, flat)
```

common_dtype *Type Promotion Rules*

Description

Computes the common dtype for a set of abstract types, respecting whether a type is ambiguous or not. A type is ambiguous if it comes from a literal (like 1 or 1.0) or was promoted to an ambiguous type. Promoting to an ambiguous type can happen in scenarios like `x + 1.2`, where `x` is a bool or an int.

Usage

```
common_dtype(
  lhs_dtype,
  rhs_dtype,
  lhs_ambiguous = FALSE,
  rhs_ambiguous = FALSE
)
```

Arguments

`lhs_dtype`	(`tengen::DataType`) The left-hand side type.
`rhs_dtype`	(`tengen::DataType`) The right-hand side type.
`lhs_ambiguous`	(`logical(1)`) Whether the left-hand side type is ambiguous.
`rhs_ambiguous`	(`logical(1)`) Whether the right-hand side type is ambiguous.

Value

(`list(dtype = [tengen::DataType], ambiguous = logical(1)‘)

ConcreteArray *Concrete Array Class*

Description

An `AbstractArray` that also holds a reference to the actual array data. Usually represents a closed-over constant in a program. Inherits from `AbstractArray`.

Usage

```
ConcreteArray(data)
```

Arguments

data	(AnvlArray) The actual array data.

Lowering

When lowering to XLA, these become inputs to the executable instead of embedding them into programs as constants. This is to avoid increasing compilation time and bloating the size of the executable.

Examples

```
y <- nv_array(c(0.5, 0.6))
x <- ConcreteArray(y)
x
ambiguous(x)
shape(x)
ndims(x)
dtype(x)

# How it appears during tracing
graph <- trace_fn(function() y, list())
graph
graph$outputs[[1]]$aval
```

---

current_platform *Current Lowering Target Platform*

Description

Returns the target platform currently set by an enclosing stablehlo() call (e.g. "cpu", "cuda"). Platform-aware lowering rules call this to branch on the target — e.g. SVD switches to a layout-flip variant when targeting CUDA with m < n because cuSOLVER's gesvd requires m >= n. Returns NULL outside of a lowering call.

local_platform() sets the current platform for the duration of the calling scope, restoring the previous value via withr::defer() when the scope exits. Useful in tests and for manually exercising platform-aware lowering rules outside of a stablehlo() call.

Usage

```
current_platform()

local_platform(platform, envir = parent.frame())
```

Arguments

platform	(character(1) | NULL) Target platform name (e.g. "cpu", "cuda"), or NULL to clear it.
envir	(environment) Environment whose exit triggers restoration of the previous platform.

Value

current_platform() returns NULL or character(1). local_platform() invisibly returns the previous platform.

See Also

stablehlo()

---

default_backend *Get the default backend*

Description

Returns the current default backend from getOption("anvl.default_backend", "xla").

Usage

```
default_backend()
```

Value

character(1) — the backend name (e.g. "xla", "quickr").

See Also

local_backend()

default_device *Get the default device*

Description

Returns a device object for the default backend and platform. For the `"xla"` backend, the platform is determined by the `PJRT_PLATFORM` environment variable (defaulting to `"cpu"`). Other backends (e.g. `"quickr"`) only support CPU. The backend defaults to `default_backend()`.

Usage

```
default_device(backend = NULL)
```

Arguments

backend	(`NULL` | `character(1)`) Backend. Defaults to `default_backend()` when `NULL`.

Value

A backend-specific device object.

See Also

`nv_device()`, `default_backend()`

device *Get the device of an array*

Description

Returns the device on which an array is allocated.

Usage

```
device(x, ...)
```

Arguments

x	(`arrayish`) An array-like object.
...	Additional arguments passed to methods (unused).

Details

This is implemented via the generic `tengen::device()`.

Value

PJRTDevice

Examples

```
x <- nv_array(1:4, dtype = "f32")
device(x)
```

---

device_arg *Select JIT device from a function argument*

Description

Pass the result to jit()'s device argument to indicate that the device should be read from a formal argument of the function being compiled. At call time, the value of that argument is used to derive the backend via backend() dispatch and is forwarded to the backend-specific JIT as the compilation device.

This is intended for functions that have no dynamic array inputs from which the backend could otherwise be detected (e.g. array constructors like prim_fill() or prim_iota()).

Usage

```
device_arg(argname)
```

Arguments

argname	(character(1)) Name of a formal argument of the function passed to jit().

Value

(AnvlDeviceArg)
An object recognized by jit().

See Also

jit(), backend()

Examples

```
f <- function(x) nv_scalar(1, device = x)
g <- jit(f, backend = "auto", device = device_arg("x"))
g(nv_device("cpu", "xla"))
```

dtype *Get the data type of an array*

Description

Returns the data type of an array (e.g. `f32`, `i64`).

Usage

```
dtype(x, ...)
```

Arguments

`x`	(`arrayish`) An array-like object.
`...`	Additional arguments passed to methods (unused).

Details

This is implemented via the generic `tengen::dtype()`.

Value

A `DataType`.

See Also

`tengen::dtype()`

Examples

```
x <- nv_array(1:4, dtype = "f32")
dtype(x)
```

eq_type *Compare AbstractArray Types*

Description

Compare two abstract arrays for type equality.

Usage

```
eq_type(e1, e2, ambiguity)

neq_type(e1, e2, ambiguity)
```

Arguments

e1	(AbstractArray) First array to compare.
e2	(AbstractArray) Second array to compare.
ambiguity	(logical(1)) Whether to consider the ambiguous field when comparing. If TRUE, arrays with different ambiguity are not equal. If FALSE, only dtype and shape are compared.

Value

logical(1) - TRUE if the arrays are equal, FALSE otherwise.

Examples

```
a <- nv_aval("f32", c(2L, 3L))
b <- nv_aval("f32", c(2L, 3L))

# Same dtype and shape
eq_type(a, b, ambiguity = FALSE)

# Different dtype
eq_type(a, nv_aval("i32", c(2L, 3L)), ambiguity = FALSE)

# Different shape
eq_type(a, nv_aval("f32", c(3L, 2L)), ambiguity = FALSE)

# ambiguity parameter controls whether ambiguous field is compared
c <- nv_aval("f32", c(2L, 3L), ambiguous = TRUE)
eq_type(a, c, ambiguity = FALSE)
eq_type(a, c, ambiguity = TRUE)

# neq_type is the negation of eq_type
neq_type(a, b, ambiguity = FALSE)
```

---

filter_list_node *Filter List Node*

Description

Subsets a ListNode to keep only the children whose names match names, then reindexes the leaf nodes so they map to contiguous positions in a flat list. If all names are kept the original tree is returned unchanged.

Usage

```
filter_list_node(tree, names)
```

Arguments

tree	(ListNode) A named list node as returned by build_tree().
names	(character) Names of children to keep.

Value

A ListNode containing only the selected children with reindexed leaves.

See Also

build_tree(), reindex_tree(), unflatten()

Examples

```
x <- list(a = 1, b = 2, c = 3)
tree <- build_tree(x)
sub <- filter_list_node(tree, c("a", "c"))
tree_size(sub)

unflatten(sub, x[c("a", "c")])
```

---

flatten *Flatten*

Description

Recursively flattens a nested list into a single flat list containing only the leaf values, preserving left-to-right order.

Currently only lists are flattened and all other objects are treated as leaves.

Use build_tree() to capture the nesting structure so it can be restored with unflatten().

Usage

```
flatten(x)
```

Arguments

x	(any) Object to flatten.

Value

list() containing the flattened values.

See Also

build_tree(), unflatten(), tree_size()

Examples

```
x <- list(a = 1, b = list(c = 2, d = 3))
flatten(x)

flatten(list(1:3, "hello"))
```

---

gradient *Gradient*

Description

Returns a new function that computes the gradient of `f` via reverse-mode automatic differentiation. `f` must return a single float scalar. The returned function has the same signature as `f` and returns the gradients in the same structure as the inputs (or the subset selected by `wrt`).

Usage

```
gradient(f, wrt = NULL)
```

Arguments

`f`	(`function`) Function to differentiate. Arguments can be arrayish (`AnvlArray`) or static (non-array) values. Must return a single scalar float array.
`wrt`	(`character` | `integer` | `NULL`) Names or positions of the arguments to compute the gradient with respect to. Only arrayish (float array) arguments can be included; static arguments must not appear in `wrt`. If `NULL` (the default), the gradient is computed with respect to all arguments (which must all be arrayish in that case).

Value

`function`

See Also

`value_and_gradient()` to get both the output and gradients, `transform_gradient()` for the low-level graph transformation.

Examples

```
f <- function(x, y) sum(x * y)
g <- jit(gradient(f))
g(nv_array(c(1, 2), dtype = "f32"), nv_array(c(3, 4), dtype = "f32"))

# Differentiate with respect to a single argument
g_x <- jit(gradient(f, wrt = "x"))
g_x(nv_array(c(1, 2), dtype = "f32"), nv_array(c(3, 4), dtype = "f32"))

# Static (non-array) arguments are passed through but cannot be in wrt
f2 <- function(x, power) sum(x^power)
g2 <- jit(gradient(f2, wrt = "x"), static = "power")
g2(nv_array(c(1, 2, 3), dtype = "f32"), power = 2L)
```

---

graph_desc_add *Add a Primitive Call to a Graph Descriptor*

Description

Add a primitive call to a graph descriptor. Inside a primitive body created with `new_primitive()`, pass the lexically-bound `self` as the primitive argument.

Usage

```
graph_desc_add(primitive, args, params = list(), infer_fn, desc = NULL)
```

Arguments

`primitive`	(`AnvlPrimitive` | `JitPrimitive`) The primitive the call is for. A `JitPrimitive` is accepted and unwrapped to its underlying `AnvlPrimitive` metadata.
`args`	(`list` of `GraphNode`) The arguments to the primitive.
`params`	(`list`) The parameters to the primitive.
`infer_fn`	(`function`) The inference function to use. Must output a list of `AbstractArray`s.
`desc`	(`GraphDescriptor` | `NULL`) The graph descriptor to add the primitive call to. Uses the current descriptor if `NULL`.

Value

(`list` of `GraphBox`)

graph_to_quickr_r_function

Convert an AnvlGraph to a plain R function

Description

Lowers a supported subset of `AnvlGraph` objects to a plain R function (no compilation) suitable for `quickr::quick()`. The returned function expects plain R scalars/vectors/arrays and returns plain R values/arrays.

Usage

```
graph_to_quickr_r_function(graph)
```

Arguments

`graph`	(`AnvlGraph`) Graph to convert.

Details

Most users will prefer `jit()` with `backend = "quickr"`. This function is the lower-level graph API.

Value

`(function)`

See Also

`jit()` with `options(anvl.backend = "quickr")` for tracing and compiling a regular R function in one step.

GraphBox

Graph Box

Description

An `AnvlBox` subclass that wraps a `GraphNode` during graph construction (tracing). When a function is traced via `trace_fn()`, each intermediate array value is represented as a `GraphBox`. It also contains an associated `GraphDescriptor` in which the node "lives".

Usage

```
GraphBox(gnode, desc)
```

Arguments

gnode	(GraphNode) The graph node – either a GraphValue or a GraphLiteral.
desc	(GraphDescriptor) The descriptor of the graph being built.

Value

(GraphBox)

Extractors

- dtype()
- shape()
- ndims()
- ambiguous()

See Also

AnvlBox, trace_fn(), jit()

GraphDescriptor *Graph Descriptor*

Description

Descriptor of an AnvlGraph. This is a mutable class.

Usage

```
GraphDescriptor(
  calls = list(),
  tensor_to_gval = NULL,
  gval_to_box = NULL,
  constants = list(),
  in_tree = NULL,
  out_tree = NULL,
  inputs = list(),
  outputs = list(),
  is_static_flat = NULL,
  static_args_flat = NULL,
  devices = character()
)
```

Arguments

`calls`	(`list(PrimitiveCall)`) The primitive calls that make up the graph.
`tensor_to_gval`	(`hashtab`) Mapping: `AnvlArray` -> `GraphValue`
`gval_to_box`	(`hashtab`) Mapping: `GraphValue` -> `GraphBox`
`constants`	(`list(GraphValue)`) The constants of the graph.
`in_tree`	(`NULL | Node`) The tree of inputs. May contain leaves for both array inputs and static (non-array) arguments. Only the array leaves correspond to entries in `inputs`; use `is_static_flat` to distinguish them.
`out_tree`	(`NULL | Node`) The tree of outputs.
`inputs`	(`list(GraphValue)`) The inputs to the graph (array arguments only).
`outputs`	(`list(GraphValue)`) The outputs of the graph.
`is_static_flat`	(`NULL	logical()`) Boolean mask indicating which flat positions in `in_tree` are static (non-array) args. `NULL` when all args are array inputs.
`static_args_flat`	(`NULL	list()`) Flattened traced values for the static arguments indicated by `is_static_flat`.
`devices`	(`character()`) Device platforms encountered during tracing (e.g. `"cpu"`, `"cuda"`). Populated automatically as arrays are registered.

Value

(`GraphDescriptor`)

`GraphLiteral` *Graph Literal*

Description

Literal in an `AnvlGraph`. This is a mutable class.

Usage

```
GraphLiteral(aval)
```

Arguments

aval	(LiteralArray) The value of the literal.

Value

(GraphLiteral)

GraphNode *Graph Node*

Description

Virtual base class for nodes in an AnvlGraph. Is either a GraphValue or a GraphLiteral. Cannot be instantiated directly - use GraphValue() or GraphLiteral() instead.

GraphValue *Graph Value*

Description

Value in an AnvlGraph. This is a mutable class.

Usage

```
GraphValue(aval)
```

Arguments

aval	(AbstractArray) The abstract value of the variable.

Value

(GraphValue)

IotaArray *Iota Array Class*

Description

An AbstractArray representing an integer sequence. Usually created by nv_iota() / nv_seq(), which both call prim_iota() internally. Inherits from AbstractArray.

Usage

```
IotaArray(shape, dtype, dimension, start = 1L, ambiguous = FALSE)
```

Arguments

shape	(stablehlo::Shape | integer()) The shape of the array.
dtype	(tengen::DataType) The data type.
dimension	(integer(1)) The dimension along which values increase.
start	(integer(1)) The starting value.
ambiguous	(logical(1)) Whether the type is ambiguous. Ambiguous types usually arise from R literals (e.g., 1L, 1.0) and follow special promotion rules. See the vignette("type-promotion") for more details.

Lowering

When lowering to stableHLO, these become iota operations that generate the integer sequence so they do not need to actually hold the data in the executable, similar to ALTREPs in R. It lowers to stablehlo::hlo_iota(), optionally shifting the starting value via stablehlo::hlo_add().

Examples

```
x <- IotaArray(shape = 4L, dtype = "i32", dimension = 1L)
x
ambiguous(x)
shape(x)
ndims(x)
dtype(x)
# How it appears during tracing:
graph <- trace_fn(function() nv_iota(dim = 1L, dtype = "i32", shape = 4L), list())
graph
graph$outputs[[1]]$aval
```

is_device *Test whether an object is a device*

Description

Test whether an object is a device

Usage

```
is_device(x)
```

Arguments

x	An object to test.

Value

logical(1)

is_dtype *Check if an object is a DataType*

Description

Tests whether x is a DataType object.

Usage

```
is_dtype(x)
```

Arguments

x	An object to test.

Value

TRUE or FALSE.

See Also

as_dtype(), tengen::is_dtype()

Examples

```
is_dtype("f32")
is_dtype(as_dtype("f32"))
```

`jit` *JIT compile a function*

Description

Wraps a function so that it is traced and compiled on first call. Subsequent calls with the same input structure, shapes, and dtypes hit an LRU cache and skip recompilation. Unlike `xla()`, the compiled executable is not created eagerly but lazily on the first invocation.

Usage

```
jit(
  f,
  static = character(),
  cache_size = 100L,
  backend = NULL,
  device = NULL,
  ...
)
```

Arguments

`f`	(`function`) Function to compile. Must accept and return `AnvlArray`s (and/or static arguments).
`static`	(`character()` | `integer()`) Names or positions of parameters of `f` that are *not* arrays. Static values are embedded as constants in the compiled program; a new compilation is triggered whenever a static value changes. For example useful when you want R control flow in your function.
`cache_size`	(`integer(1)`) Maximum number of compiled executables to keep in the LRU cache.
`backend`	(`NULL` | `character(1)`) Compilation backend (e.g. `"xla"`, `"quickr"`). The special value `"auto"` defers backend selection to call-time. `NULL` (default) respects `device` and otherwise falls back to `default_backend()`.
`device`	(`NULL` | `character(1)` | `nv_device` | `device_arg()`) Target device. When a concrete device is specified, all arrays are moved to it. The default (`NULL`) infers the device at call time, falling back to `default_device()`. In order to use dynamic device selection with the `"auto"` backend (e.g. for functions without dynamic inputs such as constant creation), set `device = device_arg("<arg>")`.
`...`	Backend-specific options. Passing an option that is not supported by the selected backend raises an error. See the **XLA JIT arguments** and **Quickr JIT arguments** sections below for the options accepted by each backend.

Value

A `JitFunction` (a `function` with the same formals as `f`). The returned wrapper expects `AnvlArray` inputs and returns `AnvlArray` values.

Device and Backend selection

There are various ways to specify which device and which backend to use.

Concrete backend: In the case where we fix a concrete backend (backend is not `"auto"`), the device can be inferred or set explicitly. Setting the device explicitly allows you to enforce that the function always uses the specified device, e.g. `"cuda:0"`. If the `device` argument is set, all encountered arrays are copied to it.

If the device is not specified (`NULL`; default) the device will be inferred from the input arrays and the constants within the program. If conflicting devices are found, an error is thrown. If no array with a device is found, we fall back to the default device.

Auto backend: When setting `backend = "auto"`, the backend will be inferred from the array inputs and otherwise fall back to the default backend. If you want to `jit()` a function without array inputs but make it work with different devices, set `device = device_arg("<argname>")` where `<argname>` is the name of the argument specifying the device. Note that this is only necessary with the `"auto"` backend. When using a concrete backend, you can just specify the device via a static argument.

XLA JIT arguments

- `donate` (`character()`, default `character()`): names of arguments whose underlying buffers may be donated to (i.e., reused/consumed by) the compiled XLA executable. Donated buffers must not be used again by the caller after the call; this can reduce memory usage and copies for large inputs. Must not overlap with `static`.

Quickr JIT arguments

- `unwrap` (`logical(1)`, default `FALSE`): if `TRUE`, the compiled function returns plain R arrays instead of `AnvlArray`s. Useful when the jitted function's output is consumed by non-anvl R code and the extra wrapping would only get stripped again.

See Also

`xla()` for ahead-of-time compilation, `jit_eval()` for evaluating an expression once.

Examples

```
f <- jit(function(x, y) x + y)
f(nv_array(1), nv_array(2))

# Static arguments enable data-dependent control flow
g <- jit(function(x, flag) {
  if (flag) x + 1 else x * 2
}, static = "flag")
g(nv_array(3), TRUE)
g(nv_array(3), FALSE)
```

```
with_backend("quickr", {
  h <- jit(function(x, y) x + y)
  h(nv_array(1), nv_array(2))
})
```

`jit_eval` *JIT-compile and evaluate an expression*

Description

Convenience wrapper that JIT-compiles and immediately evaluates a single expression. Equivalent to wrapping `expr` in an anonymous function, calling `jit()` on it, and invoking the result. Useful if you want to evaluate an expression once.

Usage

```
jit_eval(expr, ...)
```

Arguments

`expr`	(NSE) Expression to compile and evaluate.
`...`	Backend-specific options forwarded to `jit()` (e.g. `device` for the `"xla"` backend, `unwrap` for the `"quickr"` backend).

Value

(any)
Result of the compiled and evaluated expression.

Examples

```
x <- nv_array(c(1, 2, 3), dtype = "f32")
jit_eval(x + x)
```

`LiteralArray` *Literal Array Class*

Description

An `AbstractArray` where all elements have the same constant value. This either arises when using literals in traced code (e.g. `x + 1`) or when using `nv_fill()` to create a constant.

Usage

```
LiteralArray(data, shape, dtype = default_dtype(data), ambiguous)
```

Arguments

`data`	(`double(1)` | `integer(1)` | `logical(1)` | `AnvlArray`) The scalar value or scalarish AnvlArray (contains 1 element).
`shape`	(`stablehlo::Shape` | `integer()`) The shape of the array.
`dtype`	(`tengen::DataType`) The data type. Defaults to the current backend's default floating dtype, `i32` for integer, and `bool` for logical.
`ambiguous`	(`logical(1)`) Whether the type is ambiguous. Ambiguous types usually arise from R literals (e.g., `1L`, `1.0`) and follow special promotion rules. See the `vignette("type-promotion")` for more details.

Type Ambiguity

When arising from R literals, the resulting `LiteralArray` is ambiguous because no type information was available. See the `vignette("type-promotion")` for more details.

Lowering

`LiteralArrays` become constants inlined into the stableHLO program. I.e., they lower to `stablehlo::hlo_tensor()`.

Examples

```
x <- LiteralArray(1L, shape = integer(), ambiguous = TRUE)
x
ambiguous(x)
shape(x)
ndims(x)
dtype(x)
# How it appears during tracing:
# 1. via R literals
graph <- trace_fn(function() 1, list())
graph
```

```
graph$outputs[[1]]$aval
# 2. via nv_fill()
graph <- trace_fn(function() nv_fill(2L, shape = c(2, 2)), list())
graph
graph$outputs[[1]]$aval
```

`local_backend` *Temporarily set the default backend*

Description

Sets the `anvl.default_backend` option for the duration of the calling scope. This affects `nv_array()`, `nv_scalar()`, and `jit()`.

Usage

```
local_backend(backend, envir = parent.frame())
```

Arguments

`backend`	(`character(1)`) Backend to use (`"xla"` or `"quickr"`).
`envir`	The environment to scope the change to.

Value

The previous value of the option (invisibly).

`local_descriptor` *Create a graph*

Description

Creates a new `GraphDescriptor` which is afterwards accessible via `.current_descriptor()`. The graph is automatically removed when exiting the current scope. After the graph is either cleaned up automatically (by exiting the scope) or finalized, the previously built graph is restored, i.e., accessible via `.current_descriptor()`.

Usage

```
local_descriptor(..., envir = parent.frame())
```

Arguments

`...`	(any) Additional arguments to pass to the `GraphDescriptor` constructor.
`envir`	(`environment`) Environment where exit handler will be registered for cleaning up the `GraphDescriptor` if it was not returned yet.

Value

A `GraphDescriptor` object.

`map_tree` *Map Over a Tree*

Description

Apply a function to each leaf of a (possibly nested) list, preserving the tree structure. Equivalent to flattening `.x` with `flatten()`, applying `.f` to each leaf, and reassembling with `unflatten()`.

Usage

```
map_tree(.x, .f, ...)
```

Arguments

`.x`	(any) A leaf or a (nested) list of leaves.
`.f`	(`function`) Function to apply to each leaf of `.x`.
`...`	Additional arguments passed to `.f`.

Value

An object with the same nesting structure as `.x`, where each leaf is the result of `.f(leaf, ...)`.

See Also

`flatten()`, `build_tree()`, `unflatten()`, `await()`

Examples

```
map_tree(list(a = 1, b = list(c = 2, d = 3)), \(x) x + 1)
```

ndims *Get the number of dimensions of an array*

Description

Returns the number of dimensions (sometimes also refered to as rank) of an array. Equivalent to `length(shape(x))`.

Usage

```
ndims(x)
```

Arguments

x	(arrayish) An array-like object.

Value

`integer(1)`

See Also

`tengen::ndims()`

Examples

```
x <- nv_array(1:4, dtype = "f32")
ndims(x)
```

new_primitive *Create a Primitive*

Description

Builds an `AnvlPrimitive` metadata object, wraps `fn` with `jit()`, attaches the metadata via `attr(., "primitive")`, prepends class `"JitPrimitive"`, and (by default) registers the result under `name` in the primitive registry.

The backend is always `"auto"` and cannot be configured.

Usage

```
new_primitive(
  name,
  fn,
  subgraphs = character(),
  static = character(),
  device = NULL,
  register = TRUE
)
```

Arguments

name	(character(1)) Primitive name.
fn	(function) Body of the primitive. Its formals become the formals of the returned JIT-compiled callable. Inside fn, the primitive is accessible via the lexically-bound symbol self (an AnvlPrimitive); pass it as the first argument to graph_desc_add().
subgraphs	(character()) Names of parameters that are subgraphs (for higher-order primitives).
static	(character() | integer()) Passed to jit().
device	(NULL | character(1) | device_arg()) Passed to jit(). Useful for primitives with no array inputs (e.g. prim_fill) where the device must come from an explicit argument.
register	(logical(1)) If TRUE (default), register the result under name in the primitive registry.

Value

A callable of class c("JitPrimitive", "JitFunction").

---

nv_abs *Absolute Value*

Description

Element-wise absolute value. You can also use abs().

Usage

```
nv_abs(operand)
```

Arguments

operand	(arrayish) Operand.

Value

arrayish
Has the same shape and data type as the input.

See Also

prim_abs() for the underlying primitive.

Examples

```
x <- nv_array(c(-1, 2, -3))
abs(x)
```

---

nv_acos *Arc Cosine*

Description

Element-wise inverse cosine. You can also use acos().

Usage

```
nv_acos(operand)
```

Arguments

operand	(arrayish) Operand.

Value

arrayish
Has the same shape and data type as the input.

See Also

prim_acos() for the underlying primitive.

Examples

```
x <- nv_array(c(-1, 0, 1))
acos(x)
```

nv_acosh *Inverse Hyperbolic Cosine*

Description

Element-wise inverse hyperbolic cosine. You can also use `acosh()`.

Usage

```
nv_acosh(operand)
```

Arguments

`operand`	(`arrayish`) Operand.

Value

`arrayish`
Has the same shape and data type as the input.

See Also

`prim_acosh()` for the underlying primitive.

Examples

```
x <- nv_array(c(1, 2, 10))
acosh(x)
```

nv_add *Addition*

Description

Adds two arrays element-wise. You can also use the + operator.

Usage

```
nv_add(lhs, rhs)
```

Arguments

`lhs`, `rhs`	(`arrayish`) Left and right operand. Operands are promoted to a common data type. Scalars are broadcast to the shape of the other operand.

Value

arrayish
Has the same shape and the promoted common data type of the inputs.

See Also

prim_add() for the underlying primitive.

Examples

```
x <- nv_array(c(1, 2, 3))
y <- nv_array(c(4, 5, 6))
x + y
```

---

nv_and *Logical And*

Description

Element-wise logical AND. You can also use the & operator.

Usage

```
nv_and(lhs, rhs)
```

Arguments

lhs, rhs	(arrayish) Left and right operand. Operands are promoted to a common data type. Scalars are broadcast to the shape of the other operand.

Value

arrayish
Has the same shape and the promoted common data type of the inputs.

See Also

prim_and() for the underlying primitive.

Examples

```
x <- nv_array(c(TRUE, FALSE, TRUE))
y <- nv_array(c(TRUE, TRUE, FALSE))
x & y
```

nv_argmax *Index of the Maximum*

Description

Returns the index of the maximum value along a dimension. Ties are broken by returning the smallest index.

Usage

```
nv_argmax(operand, dim = NULL, drop = TRUE, nan_rm = FALSE)
```

Arguments

operand	(arrayish) Operand.
dim	(integer(1) | NULL) Dimension along which to find the index. If NULL (default), uses the last dimension.
drop	(logical(1)) If TRUE (default) the reduced dimension is removed; if FALSE it is kept with size 1.
nan_rm	(logical(1)) How to handle NaN values in floating-point inputs. If FALSE (default), NaN propagates. If TRUE, NaN values are skipped.

Value

arrayish of dtype i32
Same shape as operand with dim removed (or set to 1 if drop = FALSE).

NaN handling

With nan_rm = FALSE (default), if any entry along the reduced axis is NaN, the returned index points at the first such NaN. With nan_rm = TRUE, NaN entries are skipped.

See Also

nv_argmin(), nv_reduce_max().

Examples

```
nv_argmax(nv_array(c(3, 1, 4, 1, 5, 9, 2, 6)))
nv_argmax(nv_matrix(c(3, 1, 5, 2, 4, 0), nrow = 2, byrow = TRUE),
  dim = 2L
)
nv_argmax(nv_array(c(1, NaN, 3)))
```

```
nv_argmax(nv_array(c(1, NaN, 3)), nan_rm = TRUE)
```

---

nv_argmin *Index of the Minimum*

Description

Returns the index of the minimum value along a dimension. Ties are broken by returning the smallest index.

Usage

```
nv_argmin(operand, dim = NULL, drop = TRUE, nan_rm = FALSE)
```

Arguments

operand	(arrayish) Operand.
dim	(integer(1) | NULL) Dimension along which to find the index. If NULL (default), uses the last dimension.
drop	(logical(1)) If TRUE (default) the reduced dimension is removed; if FALSE it is kept with size 1.
nan_rm	(logical(1)) How to handle NaN values in floating-point inputs. If FALSE (default), NaN propagates. If TRUE, NaN values are skipped.

Value

arrayish of dtype i32
Same shape as operand with dim removed (or set to 1 if drop = FALSE).

NaN handling

With nan_rm = FALSE (default), if any entry along the reduced axis is NaN, the returned index points at the first such NaN. With nan_rm = TRUE, NaN entries are skipped.

See Also

nv_argmax(), nv_reduce_min().

Examples

```
nv_argmin(nv_array(c(3, 1, 4, 1, 5, 9, 2, 6)))
nv_argmin(nv_array(c(2, NaN, 1, 3)))
nv_argmin(nv_array(c(2, NaN, 1, 3)), nan_rm = TRUE)
```

nv_argsort *Argsort*

Description

Returns the indices that would sort the array along a dimension.

Usage

```
nv_argsort(operand, dim = NULL, decreasing = FALSE, stable = FALSE)
```

Arguments

operand	(arrayish) Operand.
dim	(integer(1) | NULL) Dimension along which to compute the sort permutation. If NULL (default), uses the last dimension.
decreasing	(logical(1)) If TRUE, returns indices that produce a decreasing sort. Default FALSE.
stable	(logical(1)) If TRUE, the sort is stable: indices for equal values keep their original relative order. Default FALSE.

Value

arrayish of dtype i32
Same shape as operand. For a size-0 axis, the output is an empty i32 array of the same shape (a valid empty permutation). as_array(operand)[as_array(nv_argsort(operand))] reproduces the sorted array (for 1-D inputs).

NaN handling

NaN values sort to the **end** (ascending) or **beginning** (descending), regardless of sign. +0 and -0 compare equal.

See Also

nv_sort(), prim_sort().

Examples

```
x <- nv_array(c(3, 1, 4, 1, 5))
nv_argsort(x)
```

nv_asin *Arc Sine*

Description

Element-wise inverse sine. You can also use `asin()`.

Usage

```
nv_asin(operand)
```

Arguments

operand	(arrayish) Operand.

Value

arrayish
Has the same shape and data type as the input.

See Also

`prim_asin()` for the underlying primitive.

Examples

```
x <- nv_array(c(-1, 0, 1))
asin(x)
```

nv_asinh *Inverse Hyperbolic Sine*

Description

Element-wise inverse hyperbolic sine. You can also use `asinh()`.

Usage

```
nv_asinh(operand)
```

Arguments

operand	(arrayish) Operand.

Value

arrayish
Has the same shape and data type as the input.

See Also

prim_asinh() for the underlying primitive.

Examples

```
x <- nv_array(c(-1, 0, 1))
asinh(x)
```

---

nv_atan *Arc Tangent*

Description

Element-wise inverse tangent. You can also use atan().

Usage

```
nv_atan(operand)
```

Arguments

operand	(arrayish) Operand.

Value

arrayish
Has the same shape and data type as the input.

See Also

prim_atan() for the underlying primitive.

Examples

```
x <- nv_array(c(-1, 0, 1))
atan(x)
```

nv_atan2 *Arctangent 2*

Description

Element-wise two-argument arctangent, i.e. the angle (in radians) between the positive x-axis and the point `(rhs, lhs)`.

Usage

```
nv_atan2(lhs, rhs)
```

Arguments

`lhs`, `rhs`	(`arrayish`) Left and right operand. Operands are promoted to a common data type. Scalars are broadcast to the shape of the other operand.

Value

`arrayish`
Has the same shape and the promoted common data type of the inputs.

See Also

`prim_atan2()` for the underlying primitive.

Examples

```
y <- nv_array(c(1, 0, -1))
x <- nv_array(c(0, 1, 0))
nv_atan2(y, x)
```

nv_atanh *Inverse Hyperbolic Tangent*

Description

Element-wise inverse hyperbolic tangent. You can also use `atanh()`.

Usage

```
nv_atanh(operand)
```

Arguments

operand	(arrayish) Operand.

Value

arrayish
Has the same shape and data type as the input.

See Also

prim_atanh() for the underlying primitive.

Examples

```
x <- nv_array(c(-0.5, 0, 0.5))
atanh(x)
```

---

nv_aval *Abstract Array Class*

Description

Representation of an abstract array type. During tracing, it is wrapped in a GraphNode held by a GraphBox. In the lowered AnvlGraph it is also part of GraphNodes representing the values in the program.

The base class represents an *unknown* value, but child classes exist for:

- closed-over constants: ConcreteArray
- scalar arrays arising from R literals: LiteralArray
- sequence patterns: IotaArray

To convert a arrayish value to an abstract array, use to_abstract().

Usage

```
nv_aval(dtype, shape, ambiguous = FALSE)

AbstractArray(dtype, shape, ambiguous = FALSE)
```

Arguments

dtype	(tengen::DataType | character(1)) The data type of the array.
shape	(stablehlo::Shape | integer()) The shape of the array. Can be provided as an integer vector.
ambiguous	(logical(1)) Whether the type is ambiguous. Ambiguous types usually arise from R literals (e.g., 1L, 1.0) and follow special promotion rules. See the vignette("type-promotion") for more details.

Extractors

The following extractors are available on AbstractArray objects:

- dtype(): Get the data type of the array.
- shape(): Get the shape (dimensions) of the array.
- ambiguous(): Get whether the dtype is ambiguous.
- ndims(): Get the number of dimensions.

See Also

LiteralArray, ConcreteArray, IotaArray, GraphValue, to_abstract(), GraphBox

Examples

```
# -- Creating abstract arrays --
a <- AbstractArray("f32", c(2L, 3L))
a
dtype(a)
shape(a)
ambiguous(a)

# Shorthand
nv_aval("f32", c(2L, 3L))

# How AbstractArrays appear in an AnvlGraph
graph <- trace_fn(function(x) x + 1, list(x = nv_aval("i32", 4L)))
graph
graph$inputs[[1]]$aval
```

nv_bind *Combine arrays by rows or columns*

Description

Combine arrays along the row (nv_rbind) or column (nv_cbind) dimension. Arguments are first promoted to a common data type (see nv_promote_to_common()).

Each input is then handled according to its rank:

- 0-D: broadcast to match the non-stacked dimensions of the other inputs.
- 1-D: treated as a single row/column.
- Other: used as-is.

Usage

```
nv_rbind(...)

nv_cbind(...)

## S3 method for class 'AnvlArray'
rbind(..., deparse.level = 1)

## S3 method for class 'AnvlArray'
cbind(..., deparse.level = 1)
```

Arguments

...	(arrayish) Arrays to combine. Inputs are promoted to a common data type.
deparse.level	Ignored. Kept for compatibility with base::rbind() and base::cbind().

Value

arrayish

Differences from base R

base::rbind() and base::cbind() applied to an array() of rank > 2 flatten the trailing dimensions into the column axis (so a c(2, 3, 4) array becomes a 2 x 12 matrix). nv_rbind and nv_cbind instead preserve all non-stacked dimensions: combining two c(2, 3, 4) arrays with nv_rbind produces a c(4, 3, 4) array, and with nv_cbind a c(2, 6, 4) array.

See Also

nv_concatenate()

Examples

```
# Vectors as rows / columns
nv_rbind(nv_array(1:3), nv_array(4:6))
nv_cbind(nv_array(1:3), nv_array(4:6))

# Scalar broadcasting
nv_rbind(nv_matrix(1:6, nrow = 2), nv_scalar(0))

# Rank-3 arrays preserve trailing dimensions
a <- nv_array(1:24, shape = c(2, 3, 4))
shape(nv_rbind(a, a)) # c(4, 3, 4)
```

---

nv_bitcast_convert *Bitcast Conversion*

Description

Reinterprets the bits of an array as a different data type without modifying the underlying data. If the target type is narrower, an extra trailing dimension is added; if wider, the last dimension is consumed.

Usage

```
nv_bitcast_convert(operand, dtype)
```

Arguments

operand	(arrayish) Operand.
dtype	(character(1) | tengen::DataType) Target data type.

Value

arrayish
Has the given dtype.

See Also

prim_bitcast_convert() for the underlying primitive, nv_convert() for value-preserving type conversion.

Examples

```
x <- nv_array(1L)
prim_bitcast_convert(x, dtype = "i8")
```

`nv_broadcast_arrays` *Broadcast Arrays to a Common Shape*

Description

Broadcasts arrays to a common shape using NumPy-style broadcasting rules.

Usage

```
nv_broadcast_arrays(...)
```

Arguments

`...`	(`arrayish`) Arrays to broadcast.

Value

(`list()` of `arrayish`)
List of arrays, all with the same shape.

Broadcasting Rules

1. If the arrays have different numbers of dimensions, prepend size-1 dimensions to the shorter shape.
2. For each dimension: if the sizes match, keep them; if one is 1, expand it to the other's size; otherwise raise an error.

See Also

`nv_broadcast_scalars()`, `nv_broadcast_to()`

Examples

```
x <- nv_matrix(1:6, nrow = 2)
y <- nv_array(c(10, 20, 30))
nv_broadcast_arrays(x, y)
```

nv_broadcast_scalars *Broadcast Scalars to Common Shape*

Description

Broadcast scalar arrays to match the shape of non-scalar arrays. All non-scalar arrays must have the same shape.

Usage

```
nv_broadcast_scalars(...)
```

Arguments

`...`	(arrayish) Arrays to broadcast. Scalars will be broadcast to the common non-scalar shape.

Value

(list() of arrayish)
List of broadcasted arrays.

Examples

```
x <- nv_array(c(1, 2, 3))
# scalar 1 is broadcast to shape [3]
nv_broadcast_scalars(x, nv_scalar(1))
```

nv_broadcast_to *Broadcast to Shape*

Description

Broadcasts an array to a target shape using NumPy-style broadcasting rules.

Usage

```
nv_broadcast_to(operand, shape)
```

Arguments

`operand`	(arrayish) Operand.
`shape`	(integer()) Target shape. Each existing dimension must either match or be 1.

Value

arrayish
Has the given shape and the same data type as operand.

See Also

nv_broadcast_arrays(), nv_broadcast_scalars(), prim_broadcast_in_dim() for the underlying primitive.

Examples

```
x <- nv_array(c(1, 2, 3))
nv_broadcast_to(x, shape = c(2, 3))
```

---

nv_cbrt *Cube Root*

Description

Element-wise cube root.

Usage

```
nv_cbrt(operand)
```

Arguments

operand	(arrayish) Operand.

Value

arrayish
Has the same shape and data type as the input.

See Also

prim_cbrt() for the underlying primitive.

Examples

```
x <- nv_array(c(1, 8, 27))
nv_cbrt(x)
```

nv_ceiling *Ceiling*

Description

Element-wise ceiling (round toward positive infinity). You can also use `ceiling()`.

Usage

```
nv_ceiling(operand)
```

Arguments

operand	(arrayish) Operand.

Value

arrayish
Has the same shape and data type as the input.

See Also

`prim_ceil()` for the underlying primitive.

Examples

```
x <- nv_array(c(1.2, 2.7, -1.5))
ceiling(x)
```

nv_chol *Cholesky Decomposition*

Description

Computes the Cholesky decomposition of a symmetric positive-definite matrix. Supports batched inputs: dimensions before the last two are batch dimensions.

Usage

```
nv_chol(operand, lower = FALSE)

## S3 method for class 'AnvlArray'
chol(x, ..., lower = FALSE)
```

Arguments

operand	(arrayish) Symmetric positive-definite matrix with at least 2 dimensions. The last two dimensions form the square matrix; any leading dimensions are batch dimensions.
lower	(logical(1)) If TRUE, return the lower-triangular factor.
x	(arrayish) Same as operand; this is the name used by the base R S3 generic.
...	No additional arguments.

Value

arrayish
Triangular matrix with the same shape and data type as the input.

See Also

nv_solve(), prim_chol()

Examples

```
a <- nv_matrix(c(4, 2, 2, 3), nrow = 2, dtype = "f32")
nv_chol(a)
```

---

nv_clamp *Clamp*

Description

Element-wise clamp: max(min_val, min(operand, max_val)). Converts min_val and max_val to the data type of operand.

Usage

```
nv_clamp(min_val, operand, max_val)
```

Arguments

min_val, max_val	(arrayish) Minimum and maximum values (scalar or same shape as operand).
operand	(arrayish) Operand.

Details

The underlying stableHLO function already broadcasts scalars, so no need to broadcast manually.

Value

arrayish
Has the same shape and data type as the input.

See Also

prim_clamp() for the underlying primitive.

Examples

```
x <- nv_array(c(-1, 0.5, 2))
nv_clamp(nv_scalar(0), x, nv_scalar(1))
```

---

nv_concatenate *Concatenate*

Description

Concatenates arrays along a dimension. Operands are promoted to a common data type and scalars are broadcast before concatenation.

Usage

```
nv_concatenate(..., dimension = NULL)
```

Arguments

...	(arrayish) Arrays to concatenate. Must have the same shape except along dimension.
dimension	(integer(1) | NULL) Dimension along which to concatenate. If NULL (default), assumes all inputs are at most 1-D and concatenates along dimension 1.

Value

arrayish
Has the common data type and a shape matching the inputs in all dimensions except dimension, which is the sum of input sizes.

See Also

prim_concatenate() for the underlying primitive.

Examples

```
x <- nv_array(c(1, 2, 3))
y <- nv_array(c(4, 5, 6))
nv_concatenate(x, y)
```

`nv_convert` *Convert Data Type*

Description

Converts the elements of an array to a different data type. Returns the input unchanged if it already has the target type.

Usage

```
nv_convert(operand, dtype)
```

Arguments

`operand`	(`arrayish`) Operand.
`dtype`	(`character(1)` | `tengen::DataType`) Data type.

Value

`arrayish`
Has the given `dtype` and the same shape as `operand`.

See Also

`prim_convert()` for the underlying primitive.

Examples

```
x <- nv_array(c(1L, 2L, 3L))
nv_convert(x, dtype = "f32")
```

nv_cos *Cosine*

Description

Element-wise cosine. You can also use `cos()`.

Usage

```
nv_cos(operand)
```

Arguments

operand	(arrayish) Operand.

Value

arrayish
Has the same shape and data type as the input.

See Also

prim_cos() for the underlying primitive.

Examples

```
x <- nv_array(c(0, pi / 2, pi))
cos(x)
```

nv_cosh *Hyperbolic Cosine*

Description

Element-wise hyperbolic cosine. You can also use `cosh()`.

Usage

```
nv_cosh(operand)
```

Arguments

operand	(arrayish) Operand.

Value

arrayish
Has the same shape and data type as the input.

See Also

prim_cosh() for the underlying primitive.

Examples

```
x <- nv_array(c(-1, 0, 1))
cosh(x)
```

---

nv_crossprod *Cross Product (Matrix)*

Description

Computes t(lhs) %*% rhs. If rhs is missing, computes t(lhs) %*% lhs.

Usage

```
nv_crossprod(lhs, rhs = NULL)

## S3 method for class 'AnvlArray'
crossprod(x, y = NULL, ...)
```

Arguments

lhs	(arrayish) An array with at least 2 dimensions.
rhs	(arrayish | NULL) Optional second array. If NULL, uses lhs.
x, y	Same as lhs and rhs; the names used by the base R S3 generic.
...	No additional arguments.

Value

arrayish

See Also

nv_tcrossprod(), nv_matmul()

Examples

```
x <- nv_matrix(1:6, nrow = 3, dtype = "f32")
nv_crossprod(x)
```

---

nv_cummax *Cumulative Maximum*

Description

Running maximum, optionally along a single dimension.

Usage

```
nv_cummax(operand, dim = NULL, with_indices = FALSE, nan_rm = FALSE)
```

Arguments

operand	(arrayish) Operand.
dim	(integer(1) | NULL) Dimension along which to accumulate. If NULL (default), the input is first flattened to a 1-D array, like base::cummax().
with_indices	(logical(1)) If FALSE (default), returns the running-maximum array. If TRUE, returns list(values = ..., indices = ...) where indices is the 1-based index of the last occurrence of the running maximum at each position (dtype i32, matching torch). When dim = NULL, indices refer to the flattened input.
nan_rm	(logical(1)) How to handle NaN values in floating-point inputs. If FALSE (default), NaN propagates forward from its first occurrence. If TRUE, NaN is treated as the identity element of the cumulative op (0 for sum, 1 for prod, -Inf / +Inf for max / min) and contributes nothing to the running value.

Value

arrayish (when with_indices = FALSE) or named list of two arrays (when with_indices = TRUE).

Relation to base R

Both nv_cummax() (with dim = NULL) and base::cummax() flatten a multi-dimensional input to 1-D before accumulating, but the flatten order differs: anvl arrays are row-major (C order), so the flattened sequence iterates the last dim fastest, whereas base R uses column-major (Fortran) order. The two agree on 1-D inputs.

See Also

prim_cummax() for the underlying primitive.

Examples

```
x <- nv_matrix(c(3, 1, 4, 1, 5, 9), nrow = 2)
nv_cummax(x)
nv_cummax(x, dim = 1L)
nv_cummax(x, dim = 1L, with_indices = TRUE)
nv_cummax(nv_array(c(1, NaN, 3)))                # NaN propagates
nv_cummax(nv_array(c(1, NaN, 3)), nan_rm = TRUE) # NaN skipped
```

---

nv_cummin *Cumulative Minimum*

Description

Running minimum, optionally along a single dimension.

Usage

```
nv_cummin(operand, dim = NULL, with_indices = FALSE, nan_rm = FALSE)
```

Arguments

operand	(arrayish) Operand.
dim	(integer(1) | NULL) Dimension along which to accumulate. If NULL (default), the input is first flattened to a 1-D array, like base::cummin().
with_indices	(logical(1)) If FALSE (default), returns the running-minimum array. If TRUE, returns list(values = ..., indices = ...) where indices is the 1-based index of the last occurrence of the running minimum at each position (dtype i32, matching torch). When dim = NULL, indices refer to the flattened input.
nan_rm	(logical(1)) How to handle NaN values in floating-point inputs. If FALSE (default), NaN propagates forward from its first occurrence. If TRUE, NaN is treated as the identity element of the cumulative op (0 for sum, 1 for prod, -Inf / +Inf for max / min) and contributes nothing to the running value.

Value

arrayish (when with_indices = FALSE) or named list of two arrays (when with_indices = TRUE).

Relation to base R

Both `nv_cummin()` (with `dim = NULL`) and `base::cummin()` flatten a multi-dimensional input to 1-D before accumulating, but the flatten order differs: anvl arrays are row-major (C order), so the flattened sequence iterates the last dim fastest, whereas base R uses column-major (Fortran) order. The two agree on 1-D inputs.

See Also

`prim_cummin()` for the underlying primitive.

Examples

```
x <- nv_matrix(c(3, 1, 4, 1, 5, 9), nrow = 2)
nv_cummin(x)
nv_cummin(x, dim = 1L)
nv_cummin(x, dim = 1L, with_indices = TRUE)
nv_cummin(nv_array(c(3, NaN, 1)))                # NaN propagates
nv_cummin(nv_array(c(3, NaN, 1)), nan_rm = TRUE) # NaN skipped
```

`nv_cumprod` *Cumulative Product*

Description

Cumulative product, optionally along a single dimension.

Usage

```
nv_cumprod(operand, dim = NULL, nan_rm = FALSE)
```

Arguments

`operand`	(`arrayish`) Operand.
`dim`	(`integer(1) | NULL`) Dimension along which to accumulate. If `NULL` (default), the input is first flattened to a 1-D array, like `base::cumprod()`.
`nan_rm`	(`logical(1)`) How to handle `NaN` values in floating-point inputs. If `FALSE` (default), `NaN` propagates forward from its first occurrence. If `TRUE`, `NaN` is treated as the identity element of the cumulative op (`0` for sum, `1` for prod, `-Inf` / `+Inf` for max / min) and contributes nothing to the running value.

Value

`arrayish`
Has the same shape and data type as the input.

Relation to base R

Both `nv_cumprod()` (with `dim = NULL`) and `base::cumprod()` flatten a multi-dimensional input to 1-D before accumulating, but the flatten order differs: anvl arrays are row-major (C order), so the flattened sequence iterates the last dim fastest, whereas base R uses column-major (Fortran) order. The two agree on 1-D inputs.

See Also

`prim_cumprod()` for the underlying primitive.

Examples

```
x <- nv_matrix(1:6, nrow = 2)
nv_cumprod(x)              # row-major flatten, then accumulate
nv_cumprod(x, dim = 1L)    # accumulate along rows
nv_cumprod(nv_array(c(2, NaN, 3)))                # NaN propagates
nv_cumprod(nv_array(c(2, NaN, 3)), nan_rm = TRUE) # NaN treated as 1
```

---

nv_cumsum *Cumulative Sum*

Description

Cumulative sum, optionally along a single dimension.

Usage

```
nv_cumsum(operand, dim = NULL, nan_rm = FALSE)
```

Arguments

operand	(`arrayish`) Operand.
dim	(`integer(1)` | `NULL`) Dimension along which to accumulate. If `NULL` (default), the input is first flattened to a 1-D array, like `base::cumsum()`.
nan_rm	(`logical(1)`) How to handle `NaN` values in floating-point inputs. If `FALSE` (default), `NaN` propagates forward from its first occurrence. If `TRUE`, `NaN` is treated as the identity element of the cumulative op (`0` for sum, `1` for prod, `-Inf` / `+Inf` for max / min) and contributes nothing to the running value.

Value

`arrayish`
Has the same shape and data type as the input.

Relation to base R

Both nv_cumsum() (with dim = NULL) and base::cumsum() flatten a multi-dimensional input to 1-D before accumulating, but the flatten order differs: anvl arrays are row-major (C order), so the flattened sequence iterates the last dim fastest, whereas base R uses column-major (Fortran) order. The two agree on 1-D inputs.

See Also

prim_cumsum() for the underlying primitive.

Examples

```
x <- nv_matrix(1:6, nrow = 2)
nv_cumsum(x)              # row-major flatten, then accumulate
nv_cumsum(x, dim = 1L)    # accumulate along rows
nv_cumsum(nv_array(c(1, NaN, 3)))                # NaN propagates
nv_cumsum(nv_array(c(1, NaN, 3)), nan_rm = TRUE) # NaN treated as 0
```

---

nv_det *Determinant*

Description

Computes the determinant of a square matrix via nv_determinant().

Usage

```
nv_det(operand)
```

Arguments

operand	(arrayish) Square matrix of floating-point data type.

Value

Scalar arrayish with the same dtype as operand.

See Also

nv_determinant(), nv_solve(), prim_lu()

Examples

```
a <- nv_matrix(c(4, 3, 6, 3), nrow = 2, dtype = "f64")
nv_det(a)
```

`nv_determinant` *Determinant in modulus/sign form*

Description

Computes the determinant of a square matrix in the modulus / sign decomposition matching base R's `base::determinant()`. For the plain scalar determinant, use `nv_det()`.

Usage

```
nv_determinant(operand, logarithm = TRUE)

## S3 method for class 'AnvlArray'
determinant(x, logarithm = TRUE, ...)
```

Arguments

`operand`	(`arrayish`) Square matrix of floating-point data type.
`logarithm`	(`logical(1)`) If `TRUE` (default), return the log of the absolute determinant.
`x`	(`arrayish`) Same as `operand`; this is the name used by the base R S3 generic.
`...`	No additional arguments.

Details

For computing the determinant, we use:

$$PA = LU$$

$$\det(L) = 1$$

$$\det(A) = \det(U)/\det(P) = \text{sign}(P^{-1}) \prod_i U_{ii} = \text{sign}(P) \prod_i U_{ii}$$

Matching base R's `det_ge_real`, the magnitude is computed in log space when `logarithm = TRUE` ($\sum_i \log|U_{ii}|$) and as a direct product when `logarithm = FALSE` ($\prod_i |U_{ii}|$).

Value

Named `list` with elements `modulus` and `sign`, both scalar `arrayish` with the same dtype as `operand`. The full determinant is `sign * exp(modulus)` (with `logarithm = TRUE`) or `sign * modulus` (with `logarithm = FALSE`).

See Also

`nv_det()`, `nv_solve()`, `prim_lu()`

Examples

```
a <- nv_matrix(c(4, 3, 6, 3), nrow = 2, dtype = "f64")
nv_determinant(a)
nv_determinant(a, logarithm = FALSE)
```

---

nv_device *Create a Device*

Description

Constructs a backend-specific device object.

A device identifies a compute resources, such as CPU, or a specific GPU. It is relevant for data allocation (e.g. via `nv_array()`) but also compilation (jit).

Usage

```
nv_device(x, backend = NULL)
```

Arguments

x	(`character(1)` | device object) Identifier for the device (e.g. `"cpu"`, `"cuda"`, `"cuda:<n>"`), or an existing device object (returned as-is).
backend	(`NULL` | `character(1)`) The backend for which to create the device. Defaults to `default_backend()` when `NULL`.

Value

A backend-specific device object (e.g. `PJRTDevice` for `"xla"`, `quickr_device` for `"quickr"`).

See Also

`backend()`, `AnvlBackend()`.

Examples

```
# Create CPU device for xla backend:
nv_device("cpu", "xla")
# Create CPU device for quickr backend:
nv_device("cpu", "quickr")
# Pass through an existing device:
dev <- nv_device("cpu")
identical(nv_device(dev), dev)
```

nv_diag *Diagonal Matrix*

Description

Creates a diagonal matrix from a 1-D array.

Usage

```
nv_diag(operand)
```

Arguments

operand	(arrayish) A 1-D array of length n whose elements become the diagonal entries.

Value

arrayish
An n x n matrix with x on the diagonal and zeros elsewhere.

Examples

```
nv_diag(nv_array(c(1, 2, 3)))
```

nv_digamma *Digamma*

Description

Element-wise digamma function (logarithmic derivative of the gamma function). You can also use digamma().

Usage

```
nv_digamma(operand)
```

Arguments

operand	(arrayish) Operand.

Value

arrayish
Has the same shape and data type as the input.

See Also

prim_digamma() for the underlying primitive.

Examples

```
x <- nv_array(c(0.5, 1, 2, 5))
digamma(x)
```

---

nv_div *Division*

Description

Divides two arrays element-wise. You can also use the / operator.

Usage

```
nv_div(lhs, rhs)
```

Arguments

lhs, rhs	(arrayish) Left and right operand. Operands are promoted to a common data type. Scalars are broadcast to the shape of the other operand.

Value

arrayish
Has the same shape and the promoted common data type of the inputs.

See Also

prim_div() for the underlying primitive.

Examples

```
x <- nv_array(c(10, 20, 30))
y <- nv_array(c(2, 5, 10))
x / y
```

nv_eigh *Symmetric Eigendecomposition*

Description

Computes the eigendecomposition of a symmetric matrix `operand` of shape `(n, n)`:

$$A = \text{vectors}\,\text{diag}(\text{values})\,\text{vectors}^\top.$$

Only the lower triangle of `operand` is read. The columns of `vectors` are the (orthonormal) eigenvectors and `values` is the length-`n` vector of (real) eigenvalues in ascending order. Output names and order match `base::eigen()`.

Usage

```
nv_eigh(operand)
```

Arguments

`operand`	(`arrayish`) Symmetric square matrix of floating-point data type.

Value

Named `list` with elements `values` (length `n`) and `vectors` (shape `(n, n)`). Both have the same dtype as the input.

See Also

`prim_eigh()`, `base::eigen()`

Examples

```
x <- nv_matrix(c(2, 1, 1, 2), nrow = 2, dtype = "f64")
nv_eigh(x)
```

nv_eq *Equal*

Description

Element-wise equality comparison. You can also use the == operator.

Usage

```
nv_eq(lhs, rhs)
```

Arguments

lhs, rhs	(arrayish) Left and right operand. Operands are promoted to a common data type. Scalars are broadcast to the shape of the other operand.

Value

arrayish
Has the same shape as the inputs and boolean data type.

See Also

prim_eq() for the underlying primitive.

Examples

```
x <- nv_array(c(1, 2, 3))
y <- nv_array(c(1, 3, 2))
x == y
```

---

nv_erf *Error Function*

Description

Element-wise error function erf(x) = (2 / sqrt(pi)) * integral_0^x exp(-t^2) dt.

Usage

```
nv_erf(operand)
```

Arguments

operand	(arrayish) Operand.

Value

arrayish
Has the same shape and data type as the input.

See Also

prim_erf() for the underlying primitive.

Examples

```
x <- nv_array(c(-1, 0, 1))
nv_erf(x)
```

---

nv_erf_inv *Inverse Error Function*

Description

Element-wise inverse error function (the inverse of `erf` on `(-1, 1)`).

Usage

```
nv_erf_inv(operand)
```

Arguments

operand	(arrayish) Operand.

Value

arrayish
Has the same shape and data type as the input.

See Also

`prim_erf_inv()` for the underlying primitive.

Examples

```
x <- nv_array(c(-0.5, 0, 0.5))
nv_erf_inv(x)
```

nv_erfc *Complementary Error Function*

Description

Element-wise complementary error function `erfc(x) = 1 - erf(x)`.

Usage

```
nv_erfc(operand)
```

Arguments

operand	(arrayish) Operand.

Value

arrayish
Has the same shape and data type as the input.

See Also

`prim_erfc()` for the underlying primitive.

Examples

```
x <- nv_array(c(-1, 0, 1))
nv_erfc(x)
```

nv_exp *Exponential*

Description

Element-wise exponential. You can also use `exp()`.

Usage

```
nv_exp(operand)
```

Arguments

operand	(arrayish) Operand.

Value

arrayish
Has the same shape and data type as the input.

See Also

prim_exp() for the underlying primitive.

Examples

```
x <- nv_array(c(0, 1, 2))
exp(x)
```

---

nv_expm1 *Exponential Minus One*

Description

Element-wise exp(x) - 1, more accurate for small x.

Usage

```
nv_expm1(operand)
```

Arguments

operand	(arrayish) Operand.

Value

arrayish
Has the same shape and data type as the input.

See Also

prim_expm1() for the underlying primitive.

Examples

```
x <- nv_array(c(0, 0.001, 1))
nv_expm1(x)
```

nv_extract_diag *Extract Diagonal*

Description

Extracts the diagonal elements from a 2-D array.

Usage

```
nv_extract_diag(operand)
```

Arguments

operand	(arrayish) Operand.

Value

arrayish
A 1-D array of length `min(nrow, ncol)` containing the diagonal elements.

See Also

`nv_diag()` for creating a diagonal matrix, `nv_trace()`

Examples

```
x <- nv_array(1:9, shape = c(3, 3))
nv_extract_diag(x)
```

nv_eye *Identity Matrix*

Description

Creates an `n x n` identity matrix.

`nv_eye_like()` is a variant where `dtype` and `device` default to those of `like`.

Usage

```
nv_eye(n, dtype = "f32", device = NULL)

nv_eye_like(like, n, dtype = NULL, device = NULL)
```

Arguments

n	(integer(1)) Size of the identity matrix.
dtype	(character(1) | tengen::DataType) Data type.
device	(character(1) | PJRTDevice | quickr_device | NULL) Device for data to live on.
like	(arrayish) Existing array whose attributes are used as defaults (only for nv_eye_like()).

Value

arrayish
An n x n identity matrix.

See Also

nv_diag() for general diagonal matrices.

Examples

```
nv_eye(3L)
x <- nv_fill(0, shape = c(3, 3), dtype = "f64")
nv_eye_like(x, 3L)
```

---

nv_fill *Fill Constant*

Description

Creates an array filled with a scalar value. More memory-efficient than nv_array(value, shape = shape) for large arrays.

nv_fill_like() is a variant where dtype, shape, ambiguous, and device default to those of like.

Usage

```
nv_fill(value, shape, dtype = NULL, ambiguous = FALSE, device = NULL)

nv_fill_like(
  like,
  value,
  shape = NULL,
  dtype = NULL,
  ambiguous = NULL,
  device = NULL
)
```

Arguments

value	(numeric(1)) Scalar value to fill the array with.
shape	(integer()) Shape of the output array.
dtype	(character(1) | NULL) Data type.
ambiguous	(logical(1)) Whether the type is ambiguous. Ambiguous types usually arise from R literals (e.g., 1L, 1.0) and follow special promotion rules. See the vignette("type-promotion") for more details.
device	(character(1) | PJRTDevice | quickr_device | NULL) Device for data to live on.
like	(AnvlArray) Existing array whose attributes are used as defaults (only for nv_fill_like()).

Value

arrayish
Has the given shape and dtype.

See Also

prim_fill() for the underlying primitive.

Examples

```
nv_fill(0, shape = c(2, 3))
x <- nv_matrix(1:6, nrow = 2)
nv_fill_like(x, 0)
```

---

nv_flatten *Flatte*

Description

Flattens an N-dimensional array into a 1-dimensional array. Fails with scalar inputs.

Usage

```
nv_flatten(operand)
```

Arguments

operand	(arrayish) Operand.

Value

(arrayish)
1-D array.

Examples

```
nv_flatten(matrix(1:4, nrow = 2))
```

---

nv_floor *Floor*

Description

Element-wise floor (round toward negative infinity). You can also use `floor()`.

Usage

```
nv_floor(operand)
```

Arguments

operand	(arrayish) Operand.

Value

arrayish
Has the same shape and data type as the input.

See Also

`prim_floor()` for the underlying primitive.

Examples

```
x <- nv_array(c(1.2, 2.7, -1.5))
floor(x)
```

nv_ge *Greater Than or Equal*

Description

Element-wise greater than or equal comparison. You can also use the >= operator.

Usage

```
nv_ge(lhs, rhs)
```

Arguments

lhs, rhs	(arrayish) Left and right operand. Operands are promoted to a common data type. Scalars are broadcast to the shape of the other operand.

Value

arrayish
Has the same shape as the inputs and boolean data type.

See Also

prim_ge() for the underlying primitive.

Examples

```
x <- nv_array(c(1, 2, 3))
y <- nv_array(c(3, 2, 1))
x >= y
```

nv_gt *Greater Than*

Description

Element-wise greater than comparison. You can also use the > operator.

Usage

```
nv_gt(lhs, rhs)
```

Arguments

lhs, rhs	(arrayish) Left and right operand. Operands are promoted to a common data type. Scalars are broadcast to the shape of the other operand.

Value

arrayish
Has the same shape as the inputs and boolean data type.

See Also

prim_gt() for the underlying primitive.

Examples

```
x <- nv_array(c(1, 2, 3))
y <- nv_array(c(3, 2, 1))
x > y
```

---

nv_if *Conditional Branching*

Description

Conditional execution of two branches. Unlike nv_ifelse(), which selects elements, this executes only one of the two branches depending on a scalar predicate.

Usage

```
nv_if(pred, true, false)
```

Arguments

pred	(arrayish of boolean type, scalar) Predicate.
true	(function()) Zero-argument function for the true branch.
false	(function()) Zero-argument function for the false branch. Must return outputs with the same shapes as the true branch.

Value

Result of the executed branch.

See Also

prim_if() for the underlying primitive, nv_ifelse() for element-wise selection.

Examples

```
nv_if(nv_scalar(TRUE), \() nv_scalar(1), \() nv_scalar(2))
```

---

nv_ifelse *Conditional Element Selection*

Description

Selects elements from true_value or false_value based on pred, analogous to R's ifelse().

Usage

```
nv_ifelse(pred, true_value, false_value)
```

Arguments

pred	(arrayish of boolean type) Predicate array. Must be scalar or have the same shape as the non-scalar arguments.
true_value, false_value	(arrayish) Values to return where pred is TRUE / FALSE. true_value and false_value are promoted to a common data type. Scalars (including pred) are broadcast to the shape of the non-scalar arguments.

Value

arrayish
Has the common data type of true_value and false_value and the shape of the non-scalar arguments.

See Also

prim_ifelse() for the underlying primitive.

Examples

```
pred <- nv_array(c(TRUE, FALSE, TRUE))
nv_ifelse(pred, nv_array(c(1, 2, 3)), nv_array(c(4, 5, 6)))
# scalar branches are broadcast and promoted to a common dtype
nv_ifelse(pred, nv_scalar(1L), nv_scalar(0.5))
```

nv_inv *Matrix Inverse*

Description

Computes `operand^-1`, the inverse of a square non-singular matrix, by solving `operand %*% x = I`.

For most use cases prefer `nv_solve()` directly: forming the explicit inverse is both slower and less numerically stable than solving against a right-hand side.

Usage

```
nv_inv(operand)
```

Arguments

`operand`	(`arrayish`) Square non-singular matrix.

Value

`arrayish`
The inverse, same shape and dtype as `operand`.

See Also

`nv_solve()`

Examples

```
a <- nv_matrix(c(4, 3, 6, 3), nrow = 2, dtype = "f64")
nv_inv(a)
```

nv_iota *Iota*

Description

Creates an array with values increasing along the specified dimension, starting from `start`.

`nv_iota_like()` is a variant where `dtype`, `shape`, `ambiguous`, and `device` default to those of `like`.

Usage

```
nv_iota(dim, dtype, shape, start = 1L, ambiguous = FALSE, device = NULL)

nv_iota_like(
  like,
  dim,
  shape = NULL,
  start = 1L,
  dtype = NULL,
  ambiguous = NULL,
  device = NULL
)
```

Arguments

dim	(integer(1)) Dimension along which values increase.
dtype	(character(1) | tengen::DataType) Data type.
shape	(integer()) Shape.
start	(integer(1)) Starting value (default 1).
ambiguous	(logical(1)) Whether the type is ambiguous. Ambiguous types usually arise from R literals (e.g., 1L, 1.0) and follow special promotion rules. See the vignette("type-promotion") for more details.
device	(character(1) | PJRTDevice | quickr_device | NULL) Device for data to live on.
like	(AnvlArray) Existing array whose attributes are used as defaults (only for nv_iota_like()).

Value

arrayish
Has the given dtype and shape.

See Also

nv_seq() for a simpler 1-D sequence, prim_iota() for the underlying primitive.

Examples

```
nv_iota(dim = 1L, dtype = "i32", shape = 5L)
x <- nv_fill(0L, shape = c(2, 3))
nv_iota_like(x, dim = 1L)
```

nv_is_finite *Is Finite*

Description

Element-wise check if values are finite (not `Inf`, `-Inf`, or `NaN`).

Usage

```
nv_is_finite(operand)

## S3 method for class 'AnvlArray'
is.finite(x)
```

Arguments

operand	(arrayish) Operand.
x	(arrayish) Same as operand; this is the name used by the base R S3 generic.

Value

arrayish
Has the same shape as the input and boolean data type.

See Also

`prim_is_finite()` for the underlying primitive.

Examples

```
x <- nv_array(c(1, Inf, NaN, -Inf, 0))
nv_is_finite(x)
```

nv_is_infinite *Is Infinite*

Description

Element-wise check if values are infinite (`Inf` or `-Inf`). You can also use `is.infinite()`.

Usage

```
nv_is_infinite(operand)

## S3 method for class 'AnvlArray'
is.infinite(x)
```

Arguments

operand	(arrayish) Operand.
x	(arrayish) Same as operand; this is the name used by the base R S3 generic.

Value

arrayish
Has the same shape as the input and boolean data type.

See Also

nv_is_finite(), nv_is_nan()

Examples

```
x <- nv_array(c(1, NaN, Inf, -Inf, 0))
nv_is_infinite(x)
```

---

nv_is_nan *Is NaN*

Description

Element-wise check if values are NaN. You can also use `is.nan()`.

Usage

```
nv_is_nan(operand)

## S3 method for class 'AnvlArray'
is.nan(x)
```

Arguments

operand	(arrayish) Operand.
x	(arrayish) Same as operand; this is the name used by the base R S3 generic.

Value

arrayish
Has the same shape as the input and boolean data type.

See Also

nv_is_finite(), nv_is_infinite()

Examples

```
x <- nv_array(c(1, NaN, Inf, -Inf, 0))
nv_is_nan(x)
```

---

nv_le *Less Than or Equal*

Description

Element-wise less than or equal comparison. You can also use the <= operator.

Usage

```
nv_le(lhs, rhs)
```

Arguments

lhs, rhs	(arrayish) Left and right operand. Operands are promoted to a common data type. Scalars are broadcast to the shape of the other operand.

Value

arrayish
Has the same shape as the inputs and boolean data type.

See Also

prim_le() for the underlying primitive.

Examples

```
x <- nv_array(c(1, 2, 3))
y <- nv_array(c(3, 2, 1))
x <= y
```

nv_lgamma *Log-Gamma*

Description

Element-wise natural logarithm of the absolute value of the gamma function. You can also use `lgamma()`.

Usage

```
nv_lgamma(operand)
```

Arguments

operand	(arrayish) Operand.

Value

arrayish
Has the same shape and data type as the input.

See Also

`prim_lgamma()` for the underlying primitive.

Examples

```
x <- nv_array(c(0.5, 1, 2, 5))
lgamma(x)
```

nv_log *Natural Logarithm*

Description

Element-wise natural logarithm. You can also use `log()`.

Usage

```
nv_log(operand)
```

Arguments

operand	(arrayish) Operand.

Value

arrayish
Has the same shape and data type as the input.

See Also

prim_log() for the underlying primitive.

Examples

```
x <- nv_array(c(1, 2.718, 7.389))
log(x)
```

---

nv_log10 *Base-10 Logarithm*

Description

Element-wise base-10 logarithm. You can also use log10().

Usage

```
nv_log10(operand)
```

Arguments

operand	(arrayish) Operand.

Value

arrayish
Has the same shape and data type as the input.

See Also

nv_log(), nv_log2()

Examples

```
x <- nv_array(c(1, 10, 100, 1000))
nv_log10(x)
```

nv_log1p *Log Plus One*

Description

Element-wise `log(1 + x)`, more accurate for small `x`.

Usage

```
nv_log1p(operand)
```

Arguments

`operand`	(`arrayish`) Operand.

Value

`arrayish`
Has the same shape and data type as the input.

See Also

`prim_log1p()` for the underlying primitive.

Examples

```
x <- nv_array(c(0, 0.001, 1))
nv_log1p(x)
```

nv_log2 *Base-2 Logarithm*

Description

Element-wise base-2 logarithm. You can also use `log2()`.

Usage

```
nv_log2(operand)
```

Arguments

`operand`	(`arrayish`) Operand.

Value

arrayish
Has the same shape and data type as the input.

See Also

nv_log(), nv_log10()

Examples

```
x <- nv_array(c(1, 2, 4, 8))
nv_log2(x)
```

---

nv_logistic *Logistic (Sigmoid)*

Description

Element-wise logistic sigmoid: `1 / (1 + exp(-x))`.

Usage

```
nv_logistic(operand)
```

Arguments

operand	(arrayish) Operand.

Value

arrayish
Has the same shape and data type as the input.

See Also

prim_logistic() for the underlying primitive.

Examples

```
x <- nv_array(c(-2, 0, 2))
nv_logistic(x)
```

nv_lt *Less Than*

Description

Element-wise less than comparison. You can also use the < operator.

Usage

```
nv_lt(lhs, rhs)
```

Arguments

lhs, rhs	(arrayish) Left and right operand. Operands are promoted to a common data type. Scalars are broadcast to the shape of the other operand.

Value

arrayish
Has the same shape as the inputs and boolean data type.

See Also

prim_lt() for the underlying primitive.

Examples

```
x <- nv_array(c(1, 2, 3))
y <- nv_array(c(3, 2, 1))
x < y
```

nv_lu *LU Decomposition*

Description

Computes the partial-pivoted LU decomposition of a matrix operand:

$$PA = LU,$$

where P is a permutation matrix, L is unit lower triangular, and U is upper triangular.

This function returns L and U as separate matrices. Use prim_lu() to get them in packed LU form.

Usage

```
nv_lu(operand)
```

Arguments

operand	(arrayish) Matrix of data type floating-point with exactly 2 dimensions.

Value

Named list:

- L – unit lower-triangular factor of shape (m, k), where (m, n) = shape(operand) and k = min(m, n).
- U – upper-triangular factor of shape (k, n).
- pivots – length k, dtype i32. LAPACK-style sequential 1-based row swaps as returned by getrf.
- permutation – length m, dtype i32. A 1-based permutation vector representing P.

See Also

prim_lu()

Examples

```
x <- nv_matrix(c(4, 3, 6, 3), nrow = 2, dtype = "f64")
nv_lu(x)
```

---

nv_matmul *Matrix Multiplication*

Description

Matrix multiplication of two arrays. You can also use the %*% operator. Supports batched matrix multiplication when inputs have more than 2 dimensions.

Usage

```
nv_matmul(lhs, rhs)
```

Arguments

lhs, rhs	(arrayish) Arrays with at least 2 dimensions. Operands are promoted to a common data type.

Value

arrayish

Shapes

- lhs: (b1, ..., bk, m, n)
- rhs: (b1, ..., bk, n, p)
- output: (b1, ..., bk, m, p)

See Also

prim_dot_general() for the underlying primitive.

Examples

```
x <- nv_matrix(1:6, nrow = 2)
y <- nv_matrix(1:6, nrow = 3)
x %*% y
```

---

nv_max *Maximum*

Description

Element-wise maximum of two arrays.

Usage

```
nv_max(lhs, rhs)
```

Arguments

lhs, rhs	(arrayish) Left and right operand. Operands are promoted to a common data type. Scalars are broadcast to the shape of the other operand.

Value

arrayish
Has the same shape and the promoted common data type of the inputs.

See Also

prim_max() for the underlying primitive.

Examples

```
x <- nv_array(c(1, 5, 3))
y <- nv_array(c(4, 2, 6))
nv_max(x, y)
```

---

nv_mean *Mean*

Description

Computes the arithmetic mean along the specified dimensions. You can also use `mean()`.

Usage

```
nv_mean(operand, dims = NULL, drop = TRUE, nan_rm = FALSE)

## S3 method for class 'AnvlArray'
mean(x, trim = 0, na.rm = FALSE, ..., dims = NULL, drop = TRUE)
```

Arguments

`operand`	(arrayish) Operand.
`dims`	(`integer()` | `NULL`) Dimensions to reduce. If `NULL` (default), reduces over all dimensions, returning a scalar.
`drop`	(`logical(1)`) Whether to drop reduced dimensions.
`nan_rm`	(`logical(1)`) How to handle `NaN` values in floating-point inputs. If `FALSE` (default), `NaN` propagates. If `TRUE`, `NaN` values are skipped.
`x`	(arrayish) Same as `operand`; this is the name used by the base R S3 generic.
`trim`	Currently not supported.
`na.rm`	Forwarded to `nv_mean()`'s `nan_rm` argument.
`...`	No additional arguments.

Value

arrayish
Has the same data type as the input. When `drop = TRUE`, the reduced dimensions are removed. When `drop = FALSE`, the reduced dimensions are set to 1.

See Also

nv_reduce_sum()

Examples

```
x <- nv_matrix(1:6, nrow = 2)
nv_mean(x)            # all dims -> scalar
nv_mean(x, dims = 1L)
nv_mean(nv_array(c(1, NaN, 3)))
nv_mean(nv_array(c(1, NaN, 3)), nan_rm = TRUE)
```

---

nv_median *Median*

Description

Computes the median along a dimension. Equivalent to `nv_quantile(operand, 0.5, dim, interpolation)`; for an even-length axis with the default `"linear"` interpolation, the average of the two middle values is returned, matching base R's `median()`.

You can also use `median()` directly on an `AnvlArray` or `AnvlBox`; extra arguments (e.g. `interpolation`) are forwarded via `...`.

Usage

```
nv_median(operand, dim = NULL, interpolation = "linear", nan_rm = FALSE)

## S3 method for class 'AnvlArray'
median(x, na.rm = FALSE, ..., dim = NULL, interpolation = "linear")
```

Arguments

`operand`	(arrayish) Operand.
`dim`	(integer(1) | NULL) Dimension along which to compute the median. If `NULL` (default), uses the last dimension.
`interpolation`	(character(1)) Forwarded to nv_quantile(). One of `"linear"` (default), `"lower"`, `"higher"`, `"nearest"`, `"midpoint"`.
`nan_rm`	(logical(1)) Forwarded to nv_quantile(). See its documentation for details.
`x`	(arrayish) Same as `operand`; this is the name used by the base R S3 generic.
`na.rm`	Forwarded to nv_median()'s `nan_rm` argument.
`...`	No additional arguments.

Value

arrayish
Same shape as operand with dim removed.

See Also

nv_quantile(), nv_sort(), prim_sort().

Examples

```
nv_median(nv_array(c(3, 1, 4, 1, 5, 9, 2, 6)))
median(nv_array(c(3, 1, 4, 1, 5, 9, 2, 6)))
nv_median(nv_matrix(c(3, 1, 5, 2, 4, 0), nrow = 2, byrow = TRUE),
  dim = 2L
)
# forwards through the S3 generic via `...`
median(nv_array(c(1, 2, 3, 4)), interpolation = "lower")
nv_median(nv_array(c(1, NaN, 3, 5)))
nv_median(nv_array(c(1, NaN, 3, 5)), nan_rm = TRUE)
```

---

nv_min *Minimum*

Description

Element-wise minimum of two arrays.

Usage

```
nv_min(lhs, rhs)
```

Arguments

lhs, rhs	(arrayish) Left and right operand. Operands are promoted to a common data type. Scalars are broadcast to the shape of the other operand.

Value

arrayish
Has the same shape and the promoted common data type of the inputs.

See Also

prim_min() for the underlying primitive.

Examples

```
x <- nv_array(c(1, 5, 3))
y <- nv_array(c(4, 2, 6))
nv_min(x, y)
```

---

nv_mod *Modulo (Flooring Remainder)*

Description

Element-wise flooring remainder of division. The sign of the result equals the sign of `rhs`, matching base R's `%%` operator.

Usage

```
nv_mod(lhs, rhs)
```

Arguments

`lhs`, `rhs`	(`arrayish`) Left and right operand. Operands are promoted to a common data type. Scalars are broadcast to the shape of the other operand.

Value

`arrayish`
Has the same shape and the promoted common data type of the inputs.

See Also

`nv_remainder()` for truncating remainder, `prim_remainder()` for the underlying primitive.

Examples

```
x <- nv_array(c(1L, -1L))
y <- nv_array(c(-3L, 3L))
nv_mod(x, y)
as.vector(x) %% as.vector(y)
```

nv_mul *Multiplication*

Description

Multiplies two arrays element-wise. You can also use the `*` operator.

Usage

```
nv_mul(lhs, rhs)
```

Arguments

`lhs`, `rhs`	(`arrayish`) Left and right operand. Operands are promoted to a common data type. Scalars are broadcast to the shape of the other operand.

Value

`arrayish`
Has the same shape and the promoted common data type of the inputs.

See Also

`prim_mul()` for the underlying primitive.

Examples

```
x <- nv_array(c(1, 2, 3))
y <- nv_array(c(4, 5, 6))
x * y
```

nv_ne *Not Equal*

Description

Element-wise inequality comparison. You can also use the `!=` operator.

Usage

```
nv_ne(lhs, rhs)
```

Arguments

lhs, rhs	(arrayish) Left and right operand. Operands are promoted to a common data type. Scalars are broadcast to the shape of the other operand.

Value

arrayish
Has the same shape as the inputs and boolean data type.

See Also

prim_ne() for the underlying primitive.

Examples

```
x <- nv_array(c(1, 2, 3))
y <- nv_array(c(1, 3, 2))
x != y
```

---

nv_negate *Negation*

Description

Negates an array element-wise. You can also use the unary - operator.

Usage

```
nv_negate(operand)
```

Arguments

operand	(arrayish) Operand.

Value

arrayish
Has the same shape and data type as the input.

See Also

prim_negate() for the underlying primitive.

Examples

```
x <- nv_array(c(1, -2, 3))
-x
```

---

nv_not *Logical Not*

Description

Element-wise logical NOT. You can also use the ! operator.

Usage

```
nv_not(operand)
```

Arguments

operand	(arrayish) Operand.

Value

arrayish
Has the same shape and data type as the input.

See Also

prim_not() for the underlying primitive.

Examples

```
x <- nv_array(c(TRUE, FALSE, TRUE))
!x
```

nv_or *Logical Or*

Description

Element-wise logical OR. You can also use the | operator.

Usage

```
nv_or(lhs, rhs)
```

Arguments

lhs, rhs	(arrayish) Left and right operand. Operands are promoted to a common data type. Scalars are broadcast to the shape of the other operand.

Value

arrayish
Has the same shape and the promoted common data type of the inputs.

See Also

prim_or() for the underlying primitive.

Examples

```
x <- nv_array(c(TRUE, FALSE, TRUE))
y <- nv_array(c(TRUE, TRUE, FALSE))
x | y
```

nv_outer *Outer Product*

Description

Computes the outer product of two 1-D arrays.

Usage

```
nv_outer(lhs, rhs)
```

Arguments

lhs, rhs	(arrayish) 1-D arrays.

Value

arrayish
A 2-D array of shape `(length(lhs), length(rhs))`.

Examples

```
x <- nv_array(c(1, 2, 3))
y <- nv_array(c(4, 5))
nv_outer(x, y)
```

---

nv_pad *Pad*

Description

Pads an array with a given value at the edges and optionally between elements.

Usage

```
nv_pad(
  operand,
  padding_value,
  edge_padding_low,
  edge_padding_high,
  interior_padding = NULL
)
```

Arguments

operand	(arrayish) Operand.
padding_value	(arrayish) Scalar value to use for padding. Must have the same dtype as operand.
edge_padding_low	(integer()) Amount of padding to add at the start of each dimension.
edge_padding_high	(integer()) Amount of padding to add at the end of each dimension.
interior_padding	(integer() | NULL) Amount of padding to add between elements in each dimension. If NULL (default), no interior padding is applied.

Value

arrayish
Has the same data type as operand.

See Also

prim_pad() for the underlying primitive.

Examples

```
x <- nv_array(c(1, 2, 3))
nv_pad(x, nv_scalar(0), edge_padding_low = 2L, edge_padding_high = 1L)
```

---

nv_polygamma *Polygamma*

Description

Element-wise polygamma function: the (n+1)-th derivative of the log-gamma function. The order n is broadcast against x (so nv_polygamma(1, x) works for any x). For n = 0 this is the digamma function; for n = 1, trigamma() dispatches here.

Inputs are promoted to a common floating data type and scalar arguments are broadcast to the shape of the non-scalar arguments.

Usage

```
nv_polygamma(n, x)
```

Arguments

n, x	(arrayish) Floating-point arrayish values. After promotion and broadcasting, n and x must have the same shape; n typically holds non-negative integer values.

Value

arrayish
Has the same shape and the promoted common data type of the inputs.

See Also

prim_polygamma() for the underlying primitive.

Examples

```
x <- nv_array(c(0.5, 1, 2, 5))
nv_polygamma(1, x) # trigamma
```

nv_popcnt *Population Count*

Description

Element-wise population count (number of set bits).

Usage

```
nv_popcnt(operand)
```

Arguments

operand	(arrayish) Operand.

Value

arrayish
Has the same shape and data type as the input.

See Also

prim_popcnt() for the underlying primitive.

Examples

```
x <- nv_array(c(7L, 3L, 15L))
nv_popcnt(x)
```

nv_pow *Power*

Description

Raises lhs to the power of rhs element-wise. You can also use the ^ operator.

Usage

```
nv_pow(lhs, rhs)
```

Arguments

lhs, rhs	(arrayish) Left and right operand. Operands are promoted to a common data type. Scalars are broadcast to the shape of the other operand.

Value

arrayish
Has the same shape and the promoted common data type of the inputs.

See Also

prim_pow() for the underlying primitive.

Examples

```
x <- nv_array(c(2, 3, 4))
y <- nv_array(c(3, 2, 1))
x ^ y
```

---

nv_print *Print Array*

Description

Prints an array value to the console during JIT execution and returns the input unchanged. Useful for debugging.

Usage

```
nv_print(operand)
```

Arguments

operand	(arrayish) Operand.

Value

arrayish
Returns operand unchanged.

See Also

prim_print() for the underlying primitive.

Examples

```
x <- nv_array(c(1, 2, 3))
nv_print(x)
```

nv_promote_to_common *Promote Arrays to a Common Dtype*

Description

Promote arrays to a common data type, see common_dtype for more details.

Usage

```
nv_promote_to_common(...)
```

Arguments

...	(arrayish) Arrays to promote.

Value

(list() of arrayish)

Examples

```
x <- nv_array(1L)
y <- nv_array(1.5)
# integer is promoted to float
nv_promote_to_common(x, y)
```

nv_qr *QR Decomposition*

Description

Computes the reduced QR decomposition of a matrix operand:

$$A = QR,$$

where Q has orthonormal columns ($Q^\top Q = I$) and R is upper triangular. For an $m \times n$ input with $k = \min(m, n)$, Q has shape $m \times k$ and R has shape $k \times n$.

Usage

```
nv_qr(operand)

## S3 method for class 'AnvlArray'
qr(x, ...)
```

Arguments

operand	(arrayish) Matrix of data type floating-point with exactly 2 dimensions.
x	(arrayish) Same as operand; this is the name used by the base R S3 generic.
...	No additional arguments.

Value

Named list with elements Q (shape (m, k)) and R (shape (k, n)), where (m, n) = shape(operand) and k = min(m, n). Both have the same data type as operand.

See Also

prim_qr()

Examples

```
x <- nv_matrix(c(1, 2, 3, 4, 5, 6), nrow = 3, dtype = "f32")
nv_qr(x)
```

---

nv_quantile *Quantile*

Description

Computes the probs quantile(s) of an array along a dimension.

probs follows the same scalar-vs-array convention as nv_select()'s index:

- a length-1 numeric (e.g. 0.5) treats probs as scalar — the output has dim removed, like a reduction;
- a 1-D R array (e.g. array(c(0.25, 0.5, 0.75))) prepends a leading dimension of size length(probs).

Plain length-K (K > 1) vectors are rejected; wrap with array() to make the array intent explicit.

Usage

```
nv_quantile(
  operand,
  probs,
  dim = NULL,
  interpolation = "linear",
  nan_rm = FALSE
)
```

Arguments

operand	(arrayish) Operand.
probs	(numeric(1) | 1-D array) One or more probabilities in [0, 1]. Either a length-1 numeric (scalar; dim is dropped) or a 1-D array (a leading dim of size length(probs) is prepended). Plain length-K (K > 1) vectors are rejected — wrap with array().
dim	(integer(1) | NULL) Dimension along which to compute the quantile. If NULL (default), uses the last dimension.
interpolation	(character(1)) One of "linear" (default), "lower", "higher", "nearest", "midpoint". See "Interpolation modes".
nan_rm	(logical(1)) How to handle NaN values in floating-point inputs. If FALSE (default), NaN propagates. If TRUE, NaN values are skipped.

Value

arrayish
For scalar probs: same shape as x with dim removed. For array probs: a **leading** dimension of size length(probs) is prepended.

Interpolation modes

Let h = (n - 1) * q be the 0-based fractional index for an axis of length n and probability q, with lo = floor(h), hi = ceil(h), frac = h - lo. Then:

- "linear" (default): (1 - frac) * sorted[lo] + frac * sorted[hi].
- "lower": sorted[lo] — the lower bracket of linear.
- "higher": sorted[hi] — the upper bracket of linear.
- "nearest": sorted[lo] if frac < 0.5 else sorted[hi].
- "midpoint": (sorted[lo] + sorted[hi]) / 2.

See Also

nv_median(), nv_sort().

Examples

```
x <- nv_array(c(3, 1, 4, 1, 5, 9, 2, 6))
nv_quantile(x, 0.5) # = nv_median(x)
nv_quantile(x, array(c(0.25, 0.5, 0.75)))
nv_quantile(x, 0.5, interpolation = "lower")
nv_quantile(nv_array(c(1, NaN, 3, 5)), 0.5)
nv_quantile(nv_array(c(1, NaN, 3, 5)), 0.5, nan_rm = TRUE)
```

nv_rbinom *Sample from a Binomial Distribution*

Description

Samples from a binomial distribution with n trials and success probability p. When n = 1 (the default), this is a Bernoulli distribution.

Usage

```
nv_rbinom(shape, initial_state, n = 1L, prob = 0.5, dtype = "i32")
```

Arguments

shape	(integer()) Shape.
initial_state	(arrayish) RNG state (ui64[2]).
n	(integer(1)) Number of trials.
prob	(numeric(1)) Probability of success on each trial.
dtype	(character(1) | tengen::DataType) Data type.

Value

(list() of arrayish)
List of two elements: the updated RNG state and the sampled values.

See Also

Other rng: nv_rdunif(), nv_rng_state(), nv_rnorm(), nv_runif()

Examples

```
state <- nv_rng_state(42L)
# Bernoulli samples
result <- nv_rbinom(c(2, 3), state)
result[[2]]
```

nv_rdunif *Sample from a Discrete Uniform Distribution*

Description

Samples integers from 1 to n with equal probability (with replacement), analogous to R's `sample.int(n, size, replace = TRUE)`.

Usage

```
nv_rdunif(shape, initial_state, n, dtype = "i32")
```

Arguments

shape	(integer()) Shape.
initial_state	(arrayish) RNG state (ui64[2]).
n	(integer(1)) Number of categories (samples integers 1 to n).
dtype	(character(1) | tengen::DataType) Data type.

Value

(list() of arrayish)
List of two elements: the updated RNG state and the sampled integers.

See Also

Other rng: nv_rbinom(), nv_rng_state(), nv_rnorm(), nv_runif()

Examples

```
state <- nv_rng_state(42L)
# Roll 6 dice
result <- nv_rdunif(6, state, n = 6L)
result[[2]]
```

nv_read *Read arrays from a file*

Description

Loads arrays from a file in the safetensors format.

Usage

```
nv_read(path, device = NULL, backend = default_backend())
```

Arguments

path	(character(1)) Path to the safetensors file.
device	(NULL | character(1) | PJRTDevice) The device on which to place the loaded arrays ("cpu", "cuda", ...). Default is to use the CPU.
backend	(character(1)) Backend for the loaded arrays. Defaults to default_backend().

Details

This is a convenience wrapper around nv_unserialize() that opens and closes a file connection.

Value

Named list of AnvlArray objects.

See Also

nv_save(), nv_serialize(), nv_unserialize()

Examples

```
x <- nv_matrix(1:6, nrow = 2)
x
path <- tempfile(fileext = ".safetensors")
nv_save(list(x = x), path)
nv_read(path)
```

`nv_reduce_all` *All Reduction*

Description

Performs logical AND along the specified dimensions. Returns `TRUE` only if all elements are `TRUE`.

Usage

```
nv_reduce_all(operand, dims = NULL, drop = TRUE)
```

Arguments

`operand`	(`arrayish`) Operand.
`dims`	(`integer()` | `NULL`) Dimensions to reduce. If `NULL` (default), reduces over all dimensions, returning a scalar.
`drop`	(`logical(1)`) Whether to drop reduced dimensions.

Value

`arrayish`
Boolean array. When `drop = TRUE`, the reduced dimensions are removed. When `drop = FALSE`, the reduced dimensions are set to 1.

See Also

`prim_reduce_all()` for the underlying primitive.

Examples

```
x <- nv_matrix(c(TRUE, FALSE, TRUE, TRUE), nrow = 2)
nv_reduce_all(x)            # all dims -> scalar
nv_reduce_all(x, dims = 1L)
```

nv_reduce_any *Any Reduction*

Description

Performs logical OR along the specified dimensions. Returns TRUE if any element is TRUE.

Usage

```
nv_reduce_any(operand, dims = NULL, drop = TRUE)
```

Arguments

operand	(arrayish) Operand.
dims	(integer() | NULL) Dimensions to reduce. If NULL (default), reduces over all dimensions, returning a scalar.
drop	(logical(1)) Whether to drop reduced dimensions.

Value

arrayish
Boolean array. When drop = TRUE, the reduced dimensions are removed. When drop = FALSE, the reduced dimensions are set to 1.

See Also

prim_reduce_any() for the underlying primitive.

Examples

```
x <- nv_matrix(c(TRUE, FALSE, TRUE, TRUE), nrow = 2)
nv_reduce_any(x)            # all dims -> scalar
nv_reduce_any(x, dims = 1L)
```

nv_reduce_max *Max Reduction*

Description

Finds the maximum of array elements along the specified dimensions.

Usage

```
nv_reduce_max(operand, dims = NULL, drop = TRUE, nan_rm = FALSE)
```

Arguments

operand	(arrayish) Operand.
dims	(integer() | NULL) Dimensions to reduce. If NULL (default), reduces over all dimensions, returning a scalar.
drop	(logical(1)) Whether to drop reduced dimensions.
nan_rm	(logical(1)) How to handle NaN values in floating-point inputs. If FALSE (default), NaN propagates. If TRUE, NaN values are skipped.

Value

arrayish
Has the same data type as the input. When drop = TRUE, the reduced dimensions are removed. When drop = FALSE, the reduced dimensions are set to 1.

See Also

prim_reduce_max() for the underlying primitive.

Examples

```
x <- nv_matrix(1:6, nrow = 2)
nv_reduce_max(x)            # all dims -> scalar
nv_reduce_max(x, dims = 1L)
nv_reduce_max(nv_array(c(1, NaN, 3)))
nv_reduce_max(nv_array(c(1, NaN, 3)), nan_rm = TRUE)
```

nv_reduce_min *Min Reduction*

Description

Finds the minimum of array elements along the specified dimensions.

Usage

```
nv_reduce_min(operand, dims = NULL, drop = TRUE, nan_rm = FALSE)
```

Arguments

operand	(arrayish) Operand.
dims	(integer() | NULL) Dimensions to reduce. If NULL (default), reduces over all dimensions, returning a scalar.
drop	(logical(1)) Whether to drop reduced dimensions.
nan_rm	(logical(1)) How to handle NaN values in floating-point inputs. If FALSE (default), NaN propagates. If TRUE, NaN values are skipped.

Value

arrayish
Has the same data type as the input. When drop = TRUE, the reduced dimensions are removed. When drop = FALSE, the reduced dimensions are set to 1.

See Also

prim_reduce_min() for the underlying primitive.

Examples

```
x <- nv_matrix(1:6, nrow = 2)
nv_reduce_min(x)            # all dims -> scalar
nv_reduce_min(x, dims = 1L)
nv_reduce_min(nv_array(c(1, NaN, 3)))
nv_reduce_min(nv_array(c(1, NaN, 3)), nan_rm = TRUE)
```

nv_reduce_prod *Product Reduction*

Description

Multiplies array elements along the specified dimensions.

Usage

```
nv_reduce_prod(operand, dims = NULL, drop = TRUE, nan_rm = FALSE)
```

Arguments

operand	(arrayish) Operand.
dims	(integer() | NULL) Dimensions to reduce. If NULL (default), reduces over all dimensions, returning a scalar.
drop	(logical(1)) Whether to drop reduced dimensions.
nan_rm	(logical(1)) How to handle NaN values in floating-point inputs. If FALSE (default), NaN propagates. If TRUE, NaN values are skipped.

Value

arrayish
Has the same data type as the input. When drop = TRUE, the reduced dimensions are removed. When drop = FALSE, the reduced dimensions are set to 1.

See Also

prim_reduce_prod() for the underlying primitive.

Examples

```
x <- nv_matrix(1:6, nrow = 2)
nv_reduce_prod(x)            # all dims -> scalar
nv_reduce_prod(x, dims = 1L)
nv_reduce_prod(nv_array(c(2, NaN, 3)))
nv_reduce_prod(nv_array(c(2, NaN, 3)), nan_rm = TRUE)
```

nv_reduce_sum *Sum Reduction*

Description

Sums array elements along the specified dimensions.

Usage

```
nv_reduce_sum(operand, dims = NULL, drop = TRUE, nan_rm = FALSE)
```

Arguments

operand	(arrayish) Operand.
dims	(integer() | NULL) Dimensions to reduce. If NULL (default), reduces over all dimensions, returning a scalar.
drop	(logical(1)) Whether to drop reduced dimensions.
nan_rm	(logical(1)) How to handle NaN values in floating-point inputs. If FALSE (default), NaN propagates. If TRUE, NaN values are skipped.

Value

arrayish
Has the same data type as the input. When drop = TRUE, the reduced dimensions are removed. When drop = FALSE, the reduced dimensions are set to 1.

See Also

prim_reduce_sum() for the underlying primitive.

Examples

```
x <- nv_matrix(1:6, nrow = 2)
nv_reduce_sum(x)            # all dims -> scalar
nv_reduce_sum(x, dims = 1L)
nv_reduce_sum(nv_array(c(1, NaN, 3)))
nv_reduce_sum(nv_array(c(1, NaN, 3)), nan_rm = TRUE)
```

nv_remainder *Remainder (Truncating)*

Description

Element-wise remainder. This differs from base R's %%, use nv_mod()/%% instead.

Usage

```
nv_remainder(lhs, rhs)
```

Arguments

lhs, rhs	(arrayish) Left and right operand. Operands are promoted to a common data type. Scalars are broadcast to the shape of the other operand.

Value

arrayish
Has the same shape and the promoted common data type of the inputs.

See Also

nv_mod() for the flooring remainder, prim_remainder() for the underlying primitive.

Examples

```
x <- nv_array(c(7, 8, 9))
y <- nv_array(c(3, 3, 4))
nv_remainder(x, y)
```

nv_reshape *Reshape*

Description

Reshapes an array to a new shape without changing the underlying data. Returns the input unchanged if it already has the target shape.

Usage

```
nv_reshape(operand, shape)
```

Arguments

operand	(arrayish) Operand.
shape	(integer()) Target shape. Must have the same number of elements as operand.

Details

Note that row-major order is used, which differs from R's column-major order.

Value

arrayish
Has the given shape and the same data type as operand.

See Also

prim_reshape() for the underlying primitive.

Examples

```
x <- nv_array(1:6)
nv_reshape(x, c(2, 3))
```

---

nv_reverse *Reverse*

Description

Reverses the order of elements along specified dimensions.

Usage

```
nv_reverse(operand, dims)
```

Arguments

operand	(arrayish) Operand.
dims	(integer()) Dimensions to reverse.

Value

arrayish
Has the same shape and data type as operand.

See Also

prim_reverse() for the underlying primitive.

Examples

```
x <- nv_array(c(1, 2, 3, 4, 5))
nv_reverse(x, dims = 1L)
```

---

nv_rng_state *Generate RNG State*

Description

Creates an initial RNG state from a seed. This state is required by all random sampling functions and is updated after each call.

Usage

```
nv_rng_state(seed, device = default_device())
```

Arguments

seed	(arrayish) Scalar i32 seed value.
device	(character(1) | PJRTDevice | quickr_device | NULL) Device for data to live on.

Value

nv_array of dtype ui64 and shape (2).

See Also

Other rng: nv_rbinom(), nv_rdunif(), nv_rnorm(), nv_runif()

Examples

```
state <- nv_rng_state(42L)
state
```

nv_rnorm *Sample from a Normal Distribution*

Description

Samples from a normal distribution with mean μ and standard deviation σ using the Box-Muller transform.

Usage

```
nv_rnorm(shape, initial_state, dtype = "f32", mu = 0, sigma = 1)
```

Arguments

shape	(integer()) Shape.
initial_state	(arrayish) RNG state (ui64[2]).
dtype	(character(1) | tengen::DataType) Data type.
mu	(arrayish) Mean.
sigma	(arrayish) Standard deviation. Must be positive, otherwise results are invalid.

Value

(list() of arrayish)
List of two elements: the updated RNG state and the sampled values.

Covariance

To implement a covariance structure use Cholesky decomposition.

See Also

Other rng: nv_rbinom(), nv_rdunif(), nv_rng_state(), nv_runif()

Examples

```
state <- nv_rng_state(42L)
result <- nv_rnorm(c(2, 3), state)
result[[2]]
```

nv_round *Round*

Description

Element-wise rounding. You can also use the `round()` generic.

Usage

```
nv_round(operand, method = "nearest_even")
```

Arguments

`operand`	(`arrayish`) Operand.
`method`	(`character(1)`) Rounding method. Either `"nearest_even"` (default) or `"afz"` (away from zero).

Value

`arrayish`
Has the same shape and data type as the input.

See Also

`prim_round()` for the underlying primitive.

Examples

```
x <- nv_array(c(1.4, 2.5, 3.6))
round(x)
```

nv_rsqrt *Reciprocal Square Root*

Description

Element-wise reciprocal square root, i.e. `1 / sqrt(x)`.

Usage

```
nv_rsqrt(operand)
```

Arguments

operand	(arrayish) Operand.

Value

arrayish
Has the same shape and data type as the input.

See Also

prim_rsqrt() for the underlying primitive.

Examples

```
x <- nv_array(c(1, 4, 9))
nv_rsqrt(x)
```

---

nv_runif *Sample from a Uniform Distribution*

Description

Samples from a uniform distribution in the open interval (lower, upper).

Usage

```
nv_runif(shape, initial_state, dtype = "f32", lower = 0, upper = 1)
```

Arguments

shape	(integer()) Shape.
initial_state	(arrayish) RNG state (ui64[2]).
dtype	(character(1) | tengen::DataType) Data type.
lower, upper	(numeric(1)) Lower and upper bound.

Value

(list() of arrayish)
List of two elements: the updated RNG state and the sampled values.

See Also

Other rng: nv_rbinom(), nv_rdunif(), nv_rng_state(), nv_rnorm()

Examples

```
state <- nv_rng_state(42L)
result <- nv_runif(c(2, 3), state)
result[[2]]
```

---

nv_save *Save arrays to a file*

Description

Saves a named list of arrays to a file in the safetensors format.

Usage

```
nv_save(arrays, path)
```

Arguments

arrays	(named list of AnvlArray) Named list of arrays to save. Names must be unique.
path	(character(1)) File path to write to.

Details

This is a convenience wrapper around nv_serialize() that opens and closes a file connection.

Value

NULL (invisibly).

See Also

nv_read(), nv_serialize(), nv_unserialize()

Examples

```
x <- nv_matrix(1:6, nrow = 2)
x
path <- tempfile(fileext = ".safetensors")
nv_save(list(x = x), path)
nv_read(path)
```

nv_sd *Standard Deviation Reduction*

Description

Computes the standard deviation along the specified dimensions.

Usage

```
nv_sd(operand, dims, drop = TRUE, correction = 1L, nan_rm = FALSE)
```

Arguments

operand	(arrayish) Operand.
dims	(integer() | NULL) Dimensions to reduce. If NULL (default), reduces over all dimensions, returning a scalar.
drop	(logical(1)) Whether to drop reduced dimensions.
correction	(integer(1)) Degrees of freedom correction. Default is 1 (Bessel's correction).
nan_rm	(logical(1)) How to handle NaN values in floating-point inputs. If FALSE (default), NaN propagates. If TRUE, NaN values are skipped.

Details

Uses Bessel's correction by default (correction = 1), matching R's sd(). Set correction = 0 for population standard deviation.

Value

arrayish
Has the same data type as the input. When drop = TRUE, the reduced dimensions are removed. When drop = FALSE, the reduced dimensions are set to 1.

See Also

nv_var(), nv_mean()

Examples

```
x <- nv_array(c(1, 2, 3, 4, 5))
nv_sd(x, dims = 1L)
```

nv_select *Select Elements Along a Dimension*

Description

Picks one or more elements along dimension `dim` of `operand`. Use this instead of `[` or `nv_subset` when the index to select is provided programmatically.

Usage

```
nv_select(operand, dim, index)
```

Arguments

`operand`	(arrayish) Operand.
`dim`	(integer(1)) Dimension to index into.
`index`	(arrayish) Scalar or 1D arrayish input (integer).

Value

arrayish
Same data type as `operand`. `dim` is dropped if `index` was scalar.

See Also

`nv_subset()` for general subsetting, `prim_static_slice()`.

Examples

```
m <- nv_matrix(1:6, nrow = 2)
nv_select(m, dim = 2L, index = 2L)
nv_select(m, dim = 1L, index = 1L)
nv_select(m, dim = 2L, index = array(c(1L, 3L)))
```

nv_seq *Sequence*

Description

Creates a 1-D array with values from `start` to `end` (inclusive).

Without `steps`, behaves like R's `seq(start, end)` producing integer values. With `steps`, produces `steps` evenly spaced values (like `seq(start, end, length.out = steps)`).

`nv_seq_like()` is a variant where `dtype`, `ambiguous`, and `device` default to those of `like`.

Usage

```
nv_seq(
  start,
  end,
  steps = NULL,
  dtype = NULL,
  ambiguous = FALSE,
  device = NULL
)

nv_seq_like(
  like,
  start,
  end,
  steps = NULL,
  dtype = NULL,
  ambiguous = NULL,
  device = NULL
)
```

Arguments

`start`, `end`	(`numeric(1)`) Start and end values. When `steps` is `NULL`, must satisfy `start <= end`.
`steps`	(`integer(1)` or `NULL`) Number of evenly spaced values to generate. Must be at least 1. When `NULL` (default), generates consecutive integer values from `start` to `end`.
`dtype`	(`character(1)`) Data type. Default `"i32"` when `steps` is `NULL`, `"f32"` when `steps` is given. For `nv_seq_like()`, `NULL` uses `dtype(like)`.
`ambiguous`	(`logical(1)`) Whether the type is ambiguous. Ambiguous types usually arise from R literals (e.g., `1L`, `1.0`) and follow special promotion rules. See the `vignette("type-promotion")` for more details.

device (character(1) | PJRTDevice | quickr_device | NULL)
Device for data to live on.

like (AnvlArray)
Existing array whose attributes are used as defaults (only for nv_seq_like()).

Value

arrayish
1-D array of length end - start + 1.

Examples

```
nv_seq(3, 7)
x <- nv_array(c(1, 2, 3), dtype = "f64")
nv_seq_like(x, 1, 5)
```

---

nv_serialize *Serialize arrays to raw bytes*

Description

Serializes a named list of arrays into the safetensors format.

Usage

```
nv_serialize(arrays, con = NULL)
```

Arguments

arrays (named list of AnvlArray)
Named list of arrays to serialize. Names must be unique.

con (NULL | connection)
An optional connection to write to. If NULL (default), a raw vector is returned.

Details

The ambiguity of the arrays is stored in the metadata and preserved in write-read roundtrips.

Value

A raw vector if con is NULL, otherwise NULL (invisibly).

See Also

nv_unserialize(), nv_save(), nv_read()

Examples

```
x <- nv_matrix(1:6, nrow = 2)
x
raw_data <- nv_serialize(list(x = x))
raw_data
nv_unserialize(raw_data)
```

`nv_shift_left` *Shift Left*

Description

Element-wise left bit shift.

Usage

```
nv_shift_left(lhs, rhs)
```

Arguments

`lhs`, `rhs`	(`arrayish`) Left and right operand. Operands are promoted to a common data type. Scalars are broadcast to the shape of the other operand.

Value

`arrayish`
Has the same shape and the promoted common data type of the inputs.

See Also

`prim_shift_left()` for the underlying primitive.

Examples

```
x <- nv_array(c(1L, 2L, 4L))
y <- nv_array(c(1L, 2L, 1L))
nv_shift_left(x, y)
```

nv_shift_right_arithmetic

Arithmetic Shift Right

Description

Element-wise arithmetic right bit shift.

Usage

```
nv_shift_right_arithmetic(lhs, rhs)
```

Arguments

lhs, rhs	(arrayish) Left and right operand. Operands are promoted to a common data type. Scalars are broadcast to the shape of the other operand.

Value

arrayish
Has the same shape and the promoted common data type of the inputs.

See Also

prim_shift_right_arithmetic() for the underlying primitive.

Examples

```
x <- nv_array(c(8L, -16L, 32L))
y <- nv_array(c(1L, 2L, 3L))
nv_shift_right_arithmetic(x, y)
```

nv_shift_right_logical

Logical Shift Right

Description

Element-wise logical right bit shift.

Usage

```
nv_shift_right_logical(lhs, rhs)
```

Arguments

lhs, rhs	(arrayish) Left and right operand. Operands are promoted to a common data type. Scalars are broadcast to the shape of the other operand.

Value

arrayish
Has the same shape and the promoted common data type of the inputs.

See Also

prim_shift_right_logical() for the underlying primitive.

Examples

```
x <- nv_array(c(8L, 16L, 32L))
y <- nv_array(c(1L, 2L, 3L))
nv_shift_right_logical(x, y)
```

---

nv_sign *Sign*

Description

Element-wise sign function. You can also use sign().

Usage

```
nv_sign(operand)
```

Arguments

operand	(arrayish) Operand.

Value

arrayish
Has the same shape and data type as the input.

See Also

prim_sign() for the underlying primitive.

Examples

```
x <- nv_array(c(-3, 0, 5))
sign(x)
```

---

nv_sin *Sine*

Description

Element-wise sine. You can also use `sin()`.

Usage

```
nv_sin(operand)
```

Arguments

operand	(arrayish) Operand.

Value

arrayish
Has the same shape and data type as the input.

See Also

`prim_sin()` for the underlying primitive.

Examples

```
x <- nv_array(c(0, pi / 2, pi))
sin(x)
```

nv_sinh *Hyperbolic Sine*

Description

Element-wise hyperbolic sine. You can also use `sinh()`.

Usage

```
nv_sinh(operand)
```

Arguments

operand	(arrayish) Operand.

Value

arrayish
Has the same shape and data type as the input.

See Also

`prim_sinh()` for the underlying primitive.

Examples

```
x <- nv_array(c(-1, 0, 1))
sinh(x)
```

nv_solve *Solve Linear System*

Description

Solves the linear system `a %*% x = b` for `x`. Uses LU decomposition with partial pivoting internally, so `a` need only be square and non-singular.

Usage

```
nv_solve(a, b)

## S3 method for class 'AnvlArray'
solve(a, b, ...)
```

Arguments

a	(arrayish) Coefficient matrix.
b	(arrayish) Right-hand side. If missing, returns nv_inv() of a.
...	No additional arguments.

Details

$$Ax = b$$

$$PA = LU$$

$$LUx = Pb$$

$$Ly = Pb$$

$$Ux = y$$

Value

arrayish
The solution x such that a %*% x = b.

Shapes

- a: (n, n)
- b: (n,) or (n, k)
- output: same shape as b

See Also

nv_chol(), nv_triangular_solve(), prim_lu()

Examples

```
a <- nv_matrix(c(4, 3, 6, 3), nrow = 2, dtype = "f64")
b <- nv_matrix(c(1, 2), nrow = 2, dtype = "f64")
nv_solve(a, b)
```

nv_sort *Sort*

Description

Sorts an array along a dimension.

You can also use `sort()` directly.

Usage

```
nv_sort(operand, dim = NULL, decreasing = FALSE, stable = FALSE)

## S3 method for class 'AnvlArray'
sort(x, decreasing = FALSE, ..., dim = NULL)
```

Arguments

`operand`	(arrayish) Operand.
`dim`	(`integer(1)` | `NULL`) Dimension along which to sort. If `NULL` (default), uses the last dimension.
`decreasing`	(`logical(1)`) If `TRUE`, sort in decreasing order.
`stable`	(`logical(1)`) If `TRUE`, the sort is stable: equal values keep their original relative order along `dim`. Default `FALSE`. Stability is only observable for floats when `-0` / `+0` or `-NaN` / `+NaN` are mixed (they compare equal under the total order used here); for distinct values the result is identical either way.
`x`	(arrayish) Same as `operand`; this is the name used by the base R S3 generic.
`...`	No additional arguments.

Value

arrayish
Same shape and data type as `operand`.

NaN handling

`NaN` values sort to the **end** (ascending) or **beginning** (descending), regardless of sign. `+0` and `-0` compare equal.

See Also

`prim_sort()` for the underlying primitive, `nv_argsort()`, `nv_top_k()`, `nv_median()`, `nv_argmax()`, `nv_argmin()`.

Examples

```
x <- nv_array(c(3, 1, 4, 1, 5, 9, 2, 6))
nv_sort(x)
sort(x) # via the S3 generic
nv_sort(x, decreasing = TRUE)

m <- nv_matrix(c(3, 1, 5, 2, 4, 0), nrow = 2, byrow = TRUE)
nv_sort(m, dim = 2L)
```

---

nv_sqrt *Square Root*

Description

Element-wise square root. You can also use `sqrt()`.

Usage

```
nv_sqrt(operand)
```

Arguments

operand	(arrayish) Operand.

Value

arrayish
Has the same shape and data type as the input.

See Also

`prim_sqrt()` for the underlying primitive.

Examples

```
x <- nv_array(c(1, 4, 9))
sqrt(x)
```

nv_squeeze *Squeeze*

Description

Removes dimensions of size 1 from an array.

Usage

```
nv_squeeze(operand, dims = NULL)
```

Arguments

operand	(arrayish) Operand.
dims	(integer() | NULL) Dimensions to squeeze. If NULL (default), all dimensions of size 1 are removed.

Value

arrayish
Has the same data type as operand with the specified dimensions removed.

See Also

nv_unsqueeze(), nv_reshape()

Examples

```
x <- nv_array(1:6, shape = c(1, 6, 1))
nv_squeeze(x)
```

nv_static_slice *Static Slice*

Description

Extracts a slice from an array using static (compile-time) indices. For dynamic indexing, use nv_subset() instead.

Usage

```
nv_static_slice(operand, start_indices, limit_indices, strides)
```

Arguments

operand	(arrayish) Operand.
start_indices	(integer()) Start indices (inclusive), one per dimension.
limit_indices	(integer()) End indices (inclusive), one per dimension.
strides	(integer()) Step sizes, one per dimension. A stride of 1 selects every element.

Value

arrayish
Has the same data type as operand.

See Also

nv_subset(), prim_static_slice() for the underlying primitive.

Examples

```
x <- nv_array(1:10)
nv_static_slice(x, start_indices = 2L, limit_indices = 5L, strides = 1L)
```

---

nv_sub *Subtraction*

Description

Subtracts two arrays element-wise. You can also use the - operator.

Usage

```
nv_sub(lhs, rhs)
```

Arguments

lhs, rhs	(arrayish) Left and right operand. Operands are promoted to a common data type. Scalars are broadcast to the shape of the other operand.

Value

arrayish
Has the same shape and the promoted common data type of the inputs.

See Also

prim_sub() for the underlying primitive.

Examples

```
x <- nv_array(c(4, 5, 6))
y <- nv_array(c(1, 2, 3))
x - y
```

---

nv_svd *Singular Value Decomposition*

Description

Computes the reduced ("economy") singular value decomposition of a matrix operand of shape (m, n):

$$A = u \operatorname{diag}(d)\, vt,$$

where u has orthonormal columns, vt has orthonormal rows, and d is the length-k (k = min(m, n)) vector of non-negative singular values in descending order.

Note: unlike base::svd(), which returns the right singular vectors as v of shape (n, k) (so that a = u %*% diag(d) %*% t(v)), this primitive returns them already transposed as vt of shape (k, n) (matching the underlying LAPACK / cuSOLVER output and avoiding an extra transpose).

Supports any matrix shape on both the host (LAPACK gesdd) and CUDA (cuSOLVER gesvd) backends. cuSOLVER's m >= n requirement is handled transparently via a layout flip for wide matrices.

Usage

```
nv_svd(operand)
```

Arguments

operand	(arrayish) Matrix of data type floating-point with exactly 2 dimensions.

Value

Named list with elements d (length k), u (shape (m, k)), and vt (shape (k, n)). All have the same dtype as the input.

See Also

prim_svd(), base::svd()

Examples

```
x <- nv_matrix(c(1, 0, 0, 1, 0, 1), nrow = 3, dtype = "f64")
nv_svd(x)
```

`nv_tan` *Tangent*

Description

Element-wise tangent. You can also use `tan()`.

Usage

```
nv_tan(operand)
```

Arguments

operand	(arrayish) Operand.

Value

arrayish
Has the same shape and data type as the input.

See Also

`prim_tan()` for the underlying primitive.

Examples

```
x <- nv_array(c(0, 0.5, 1))
tan(x)
```

nv_tanh *Hyperbolic Tangent*

Description

Element-wise hyperbolic tangent. You can also use `tanh()`.

Usage

```
nv_tanh(operand)
```

Arguments

operand	(arrayish) Operand.

Value

arrayish
Has the same shape and data type as the input.

See Also

`prim_tanh()` for the underlying primitive.

Examples

```
x <- nv_array(c(-1, 0, 1))
tanh(x)
```

nv_tcrossprod *Transpose Cross Product (Matrix)*

Description

Computes `lhs %*% t(rhs)`. If `rhs` is missing, computes `lhs %*% t(lhs)`.

Usage

```
nv_tcrossprod(lhs, rhs = NULL)

## S3 method for class 'AnvlArray'
tcrossprod(x, y = NULL, ...)
```

Arguments

lhs	(arrayish) An array with at least 2 dimensions.
rhs	(arrayish | NULL) Optional second array. If NULL, uses lhs.
x, y	Same as lhs and rhs; the names used by the base R S3 generic.
...	No additional arguments.

Value

arrayish

See Also

nv_crossprod(), nv_matmul()

Examples

```
x <- nv_matrix(1:6, nrow = 2, dtype = "f32")
nv_tcrossprod(x)
```

---

nv_top_k *Top-K Elements*

Description

Returns the k largest values along a dimension, sorted in decreasing order.

Usage

```
nv_top_k(operand, k, dim = NULL, with_indices = FALSE)
```

Arguments

operand	(arrayish) Operand.
k	(integer(1)) Number of top elements to return. Must satisfy 1 <= k <= shape(operand)[dim].
dim	(integer(1) | NULL) Dimension along which to take the top k. If NULL (default), uses the last dimension.
with_indices	(logical(1)) If FALSE (default), returns just the top-k values. If TRUE, returns list(values = ..., indices = ...) where indices is the 1-based position of each top-k value along dim (dtype i32).

Value

arrayish (when with_indices = FALSE) or named list of two arrays (when with_indices = TRUE). Output shape matches operand with dim resized to k; values are sorted decreasing along dim.

NaN handling

NaN ranks larger than any finite value (so it appears first in the top-k output); -NaN ranks smaller. Unlike nv_sort(), the sign bit is not canonicalized.

See Also

prim_top_k() for the underlying primitive, nv_sort().

Examples

```
x <- nv_array(c(3, 1, 4, 1, 5, 9, 2, 6))
nv_top_k(x, k = 3L)
nv_top_k(x, k = 3L, with_indices = TRUE)

m <- nv_matrix(c(3, 1, 5, 2, 4, 0), nrow = 2, byrow = TRUE)
nv_top_k(m, k = 2L, dim = 2L)
```

---

nv_trace *Matrix Trace*

Description

Computes the trace (sum of diagonal elements) of a 2-D array.

Usage

```
nv_trace(operand)
```

Arguments

operand	(arrayish) Operand.

Value

arrayish
A scalar with the same data type as operand.

See Also

nv_extract_diag(), nv_diag()

Examples

```
x <- nv_array(c(1, 0, 0, 0, 2, 0, 0, 0, 3), shape = c(3, 3))
nv_trace(x)
```

`nv_transpose` *Transpose*

Description

Permutes the dimensions of an array. You can also use `t()` for matrices.

Usage

```
nv_transpose(operand, permutation = NULL)

## S3 method for class 'AnvlArray'
t(x)
```

Arguments

`operand`	(arrayish) Operand.
`permutation`	(`integer()` | `NULL`) New ordering of dimensions. If `NULL` (default), reverses the dimensions.
`x`	(arrayish) Same as `operand`; this is the name used by the base R S3 generic.

Value

arrayish
Has the same data type as `operand` and shape `nv_shape(operand)[permutation]`.

See Also

`prim_transpose()` for the underlying primitive.

Examples

```
x <- nv_matrix(1:6, nrow = 2)
t(x)
```

`nv_triangular_solve` *Triangular Solve*

Description

Solves a triangular system of linear equations. When `left_side = TRUE`, returns `x` such that `op(a) %*% x = b`. When `left_side = FALSE`, returns `x` such that `x %*% op(a) = b`. Here `op` is `a` or `t(a)` depending on `transpose_a`.

Usage

```
nv_triangular_solve(
  a,
  b,
  left_side = TRUE,
  lower = TRUE,
  unit_diagonal = FALSE,
  transpose_a = FALSE
)
```

Arguments

`a`	(arrayish) Triangular coefficient matrix with at least 2 dimensions. The last two dimensions must be equal; any leading dimensions are batch dimensions.
`b`	(arrayish) Right-hand side. For `a` of shape `(B..., n, n)`, `b` may be either: • full rank — shape `(B..., n, k)` when `left_side = TRUE`, or `(B..., k, n)` when `left_side = FALSE`; • one rank less, shape `(B..., n)`, meaning a single column (`left_side = TRUE`) or row (`left_side = FALSE`) per batch — it is reshaped internally and the reshape is undone on the result so the output rank matches `b`. `b`'s batch dimensions (`B...`) must match `a`'s exactly.
`left_side`	(logical(1)) If `TRUE` (default), solve `op(a) %*% x = b`; if `FALSE`, solve `x %*% op(a) = b`.
`lower`	(logical(1)) Whether `a` is lower or upper triangular. Defaults to `TRUE`.
`unit_diagonal`	(logical(1)) If `TRUE`, the diagonal of `a` is treated as all ones (and the actual values on the diagonal are ignored). Defaults to `FALSE`.
`transpose_a`	(logical(1)) If `TRUE`, solve with `t(a)` in place of `a`. Defaults to `FALSE`.

Details

As a convenience, b may have one fewer dimension than a (a single right-hand side per batch, shape (B..., n) for a of shape (B..., n, n)). It is reshaped internally to a column (left_side = TRUE) or row (left_side = FALSE) and reshaped back on the way out. Because we don't broadcast, this is not ambiguous (as it would be for NumPy).

Value

arrayish
The solution x, with the same shape and dtype as b.

See Also

nv_solve(), nv_chol(), prim_triangular_solve()

Examples

```
L <- nv_matrix(c(2, 1, 0, 3), nrow = 2, dtype = "f32")
b <- nv_matrix(c(4, 3), nrow = 2, dtype = "f32")
nv_triangular_solve(L, b)
```

---

nv_tril *Lower Triangular Matrix*

Description

Returns the lower triangular part of a 2-D array, setting elements above the specified diagonal to zero.

Usage

```
nv_tril(operand, diagonal = 0L)
```

Arguments

operand	(arrayish) Operand.
diagonal	(integer(1)) Diagonal offset. 0 (default) is the main diagonal, positive values include diagonals above, negative values exclude diagonals below.

Value

arrayish
Has the same shape and data type as operand.

See Also

nv_triu()

Examples

```
x <- nv_fill(1, c(3, 3))
nv_tril(x)
```

---

nv_triu *Upper Triangular Matrix*

Description

Returns the upper triangular part of a 2-D array, setting elements below the specified diagonal to zero.

Usage

```
nv_triu(operand, diagonal = 0L)
```

Arguments

operand	(arrayish) Operand.
diagonal	(integer(1)) Diagonal offset. 0 (default) is the main diagonal, positive values exclude diagonals above, negative values include diagonals below.

Value

arrayish
Has the same shape and data type as operand.

See Also

nv_tril()

Examples

```
x <- nv_fill(1, c(3, 3))
nv_triu(x)
```

nv_trunc *Truncate*

Description

Element-wise truncation (round toward zero). You can also use `trunc()`.

Usage

```
nv_trunc(operand)
```

Arguments

operand	(arrayish) Operand.

Value

arrayish
Has the same shape and data type as the input.

See Also

`nv_floor()`, `nv_ceiling()`, `nv_round()`.

Examples

```
x <- nv_array(c(1.2, 2.7, -1.5))
trunc(x)
```

nv_unserialize *Deserialize arrays from raw bytes*

Description

Deserializes arrays from the safetensors format.

Usage

```
nv_unserialize(con, device = NULL, backend = default_backend())
```

Arguments

con	(connection | raw) A connection or raw vector to read from.
device	(NULL | character(1) | PJRTDevice) The device on which to place the loaded arrays ("cpu", "cuda", ...). Default is to use the CPU.
backend	(character(1)) Backend for the loaded arrays. Defaults to default_backend().

Details

The data type, shape, and ambiguity of each array are restored from the serialized data.

Value

Named list of AnvlArray objects.

See Also

nv_serialize(), nv_save(), nv_read()

Examples

```
x <- nv_matrix(1:6, nrow = 2)
x
raw_data <- nv_serialize(list(x = x))
raw_data
nv_unserialize(raw_data)
```

---

nv_unsqueeze *Unsqueeze*

Description

Inserts a dimension of size 1 at the specified position.

Usage

```
nv_unsqueeze(operand, dim)
```

Arguments

operand	(arrayish) Operand.
dim	(integer(1)) Position at which to insert the new dimension.

Value

arrayish
Has the same data type as operand with an extra dimension of size 1.

See Also

nv_squeeze(), nv_reshape()

Examples

```
x <- nv_array(c(1, 2, 3))
nv_unsqueeze(x, dim = 1L)
```

---

nv_var *Variance Reduction*

Description

Computes the variance along the specified dimensions.

Usage

```
nv_var(operand, dims, drop = TRUE, correction = 1L, nan_rm = FALSE)
```

Arguments

operand	(arrayish) Operand.
dims	(integer() | NULL) Dimensions to reduce. If NULL (default), reduces over all dimensions, returning a scalar.
drop	(logical(1)) Whether to drop reduced dimensions.
correction	(integer(1)) Degrees of freedom correction. Default is 1 (Bessel's correction).
nan_rm	(logical(1)) How to handle NaN values in floating-point inputs. If FALSE (default), NaN propagates. If TRUE, NaN values are skipped.

Details

Uses Bessel's correction by default (correction = 1), matching R's var(). Set correction = 0 for population variance.

Value

arrayish
Has the same data type as the input. When drop = TRUE, the reduced dimensions are removed. When drop = FALSE, the reduced dimensions are set to 1.

See Also

nv_sd(), nv_mean()

Examples

```
x <- nv_array(c(1, 2, 3, 4, 5))
nv_var(x, dims = 1L)
nv_var(nv_array(c(1, NaN, 3, 5)), dims = 1L, nan_rm = TRUE)
```

---

nv_while *While Loop*

Description

Executes a functional while loop.

Usage

```
nv_while(init, cond, body)
```

Arguments

init	(list()) Named list of initial state values.
cond	(function) Condition function returning a scalar boolean. Receives the state values as arguments.
body	(function) Body function returning the updated state as a named list with the same structure as init.

Value

Final state after the loop terminates (same structure as init).

See Also

prim_while() for the underlying primitive.

Examples

```
nv_while(
  init = list(i = nv_scalar(0L), total = nv_scalar(0L)),
  cond = function(i, total) i < 5L,
  body = function(i, total) list(
    i = i + 1L,
    total = total + i
  )
)
```

`nv_xor` *Logical Xor*

Description

Element-wise logical XOR.

Usage

```
nv_xor(lhs, rhs)
```

Arguments

`lhs, rhs`	(`arrayish`) Left and right operand. Operands are promoted to a common data type. Scalars are broadcast to the shape of the other operand.

Value

`arrayish`
Has the same shape and the promoted common data type of the inputs.

See Also

`prim_xor()` for the underlying primitive.

Examples

```
x <- nv_array(c(TRUE, FALSE, TRUE))
y <- nv_array(c(TRUE, TRUE, FALSE))
nv_xor(x, y)
```

`platform.AnvlArray` *Get the platform of an array or buffer*

Description

Returns the platform name (e.g. `"cpu"`, `"cuda"`) identifying the compute backend.

Usage

```
## S3 method for class 'AnvlArray'
platform(x, ...)

platform(x, ...)
```

Arguments

`x`	(`arrayish`) An array-like object.
`...`	Additional arguments passed to methods (unused).

Details

Implemented via the generic `pjrt::platform()`.

Value

`character(1)`

See Also

`pjrt::platform()`

Examples

```
x <- nv_array(1:4, dtype = "f32")
platform(x)
```

pmap_tree *Map Over Multiple Trees*

Description

Apply a function leaf-wise over several trees with the same structure. All trees in `.l` must have identical structure.

Usage

```
pmap_tree(.l, .f, ...)
```

Arguments

`.l`	(`list`) A non-empty list of trees, all with the same structure.
`.f`	(`function`) Function to call with one leaf from each tree (positional arguments, in the order given by `.l`).
`...`	Additional arguments passed to `.f` after the per-tree leaves.

Value

A tree with the same structure as `.l[[1]]`, where each leaf is `.f(leaf_1, leaf_2, ..., leaf_n ...)`.

See Also

`map_tree()`, `flatten()`, `unflatten()`

Examples

```
pmap_tree(list(list(a = 1, b = 2), list(a = 10, b = 20)), `+`)
```

prim_abs *Primitive Absolute Value*

Description

Element-wise absolute value.

Usage

```
prim_abs(operand)
```

Arguments

operand (arrayish)
Arrayish value of data type signed integer or floating-point.

Value

arrayish
Has the same shape and data type as the input. It is ambiguous if the input is ambiguous.

Implemented Rules

- stablehlo
- quickr
- reverse

StableHLO

Lowers to stablehlo::hlo_abs().

See Also

nv_abs(), abs()

Examples

```
x <- nv_array(c(-1, 2, -3))
prim_abs(x)
```

---

prim_acos *Primitive Arc Cosine*

Description

Element-wise inverse cosine.

Usage

```
prim_acos(operand)
```

Arguments

operand (arrayish)
Arrayish value of data type floating-point.

Value

arrayish
Has the same shape and data type as the input. It is ambiguous if the input is ambiguous.

Implemented Rules

- stablehlo
- reverse

StableHLO

Lowers to stablehlo::hlo_acos().

See Also

nv_acos(), acos()

Examples

```
x <- nv_array(c(-1, 0, 1))
prim_acos(x)
```

---

prim_acosh *Primitive Inverse Hyperbolic Cosine*

Description

Element-wise inverse hyperbolic cosine.

Usage

```
prim_acosh(operand)
```

Arguments

operand	(arrayish) Arrayish value of data type floating-point.

Value

arrayish
Has the same shape and data type as the input. It is ambiguous if the input is ambiguous.

Implemented Rules

- stablehlo
- reverse

StableHLO

Lowers to stablehlo::hlo_acosh().

See Also

nv_acosh(), acosh()

Examples

```
x <- nv_array(c(1, 2, 10))
prim_acosh(x)
```

---

prim_add *Primitive Addition*

Description

Adds two arrays element-wise.

Usage

```
prim_add(lhs, rhs)
```

Arguments

lhs, rhs	(arrayish) Arrayish values of any data type. Must have the same shape.

Value

arrayish
Has the same shape and data type as the inputs. It is ambiguous if both inputs are ambiguous.

Implemented Rules

- stablehlo
- quickr
- reverse

StableHLO

Lowers to stablehlo::hlo_add().

See Also

nv_add(), +

Examples

```
x <- nv_array(c(1, 2, 3))
y <- nv_array(c(4, 5, 6))
prim_add(x, y)
```

`prim_and` *Primitive And*

Description

Element-wise logical AND.

Usage

```
prim_and(lhs, rhs)
```

Arguments

`lhs`, `rhs`	(`arrayish`) Arrayish values of data type boolean, integer, or unsigned integer. Must have the same shape.

Value

`arrayish`
Has the same shape and data type as the inputs. It is ambiguous if both inputs are ambiguous.

Implemented Rules

- `stablehlo`
- `quickr`
- `reverse`

StableHLO

Lowers to `stablehlo::hlo_and()`.

See Also

`nv_and()`, `&`

Examples

```
x <- nv_array(c(TRUE, FALSE, TRUE))
y <- nv_array(c(TRUE, TRUE, FALSE))
prim_and(x, y)
```

prim_argmax *Primitive Argmax*

Description

Returns the index of the maximum value along a single dimension. Ties are broken by returning the smallest index.

Usage

```
prim_argmax(operand, dim, drop = TRUE)
```

Arguments

operand	(arrayish) Arrayish value of any data type.
dim	(integer(1)) Dimension along which to find the index of the maximum.
drop	(logical(1)) If TRUE (default) the reduced dimension is removed; if FALSE it is kept with size 1.

Value

arrayish of dtype i32
Same shape as operand with dim removed (or set to 1 if drop = FALSE).

Implemented Rules

- stablehlo
- reverse

StableHLO

Lowers to a variadic stablehlo::hlo_reduce() over (values, indices) with a (value > value | (value == value & idx < idx)) selector.

See Also

prim_argmin(), nv_argmax()

Examples

```
prim_argmax(nv_array(c(3, 1, 4, 1, 5)), dim = 1L)
```

prim_argmin *Primitive Argmin*

Description

Returns the index of the minimum value along a single dimension. Ties are broken by returning the smallest index.

Usage

```
prim_argmin(operand, dim, drop = TRUE)
```

Arguments

operand	(arrayish) Arrayish value of any data type.
dim	(integer(1)) Dimension along which to find the index of the maximum.
drop	(logical(1)) If TRUE (default) the reduced dimension is removed; if FALSE it is kept with size 1.

Value

arrayish of dtype i32
Same shape as operand with dim removed (or set to 1 if drop = FALSE).

Implemented Rules

- stablehlo
- reverse

StableHLO

Lowers to a variadic stablehlo::hlo_reduce() over (values, indices) with a (value < value | (value == value & idx < idx)) selector.

See Also

prim_argmax(), nv_argmin()

Examples

```
prim_argmin(nv_array(c(3, 1, 4, 1, 5)), dim = 1L)
```

prim_asin *Primitive Arc Sine*

Description

Element-wise inverse sine.

Usage

```
prim_asin(operand)
```

Arguments

operand	(arrayish) Arrayish value of data type floating-point.

Value

arrayish
Has the same shape and data type as the input. It is ambiguous if the input is ambiguous.

Implemented Rules

- stablehlo
- reverse

StableHLO

Lowers to stablehlo::hlo_asin().

See Also

nv_asin(), asin()

Examples

```
x <- nv_array(c(-1, 0, 1))
prim_asin(x)
```

`prim_asinh` *Primitive Inverse Hyperbolic Sine*

Description

Element-wise inverse hyperbolic sine.

Usage

```
prim_asinh(operand)
```

Arguments

`operand`	(`arrayish`) Arrayish value of data type floating-point.

Value

`arrayish`
Has the same shape and data type as the input. It is ambiguous if the input is ambiguous.

Implemented Rules

- `stablehlo`
- `reverse`

StableHLO

Lowers to `stablehlo::hlo_asinh()`.

See Also

`nv_asinh()`, `asinh()`

Examples

```
x <- nv_array(c(-1, 0, 1))
prim_asinh(x)
```

`prim_atan` *Primitive Arc Tangent*

Description

Element-wise inverse tangent.

Usage

```
prim_atan(operand)
```

Arguments

`operand`	(`arrayish`) Arrayish value of data type floating-point.

Value

`arrayish`
Has the same shape and data type as the input. It is ambiguous if the input is ambiguous.

Implemented Rules

- `stablehlo`
- `reverse`

StableHLO

Lowers to `stablehlo::hlo_atan()`.

See Also

`nv_atan()`, `atan()`

Examples

```
x <- nv_array(c(-1, 0, 1))
prim_atan(x)
```

`prim_atan2` *Primitive Atan2*

Description

Element-wise atan2 operation.

Usage

```
prim_atan2(lhs, rhs)
```

Arguments

`lhs`, `rhs`	(`arrayish`) Arrayish values of data type floating-point. Must have the same shape.

Value

`arrayish`
Has the same shape and data type as the inputs. It is ambiguous if both inputs are ambiguous.

Implemented Rules

- `stablehlo`
- `reverse`

StableHLO

Lowers to `stablehlo::hlo_atan2()`.

See Also

`nv_atan2()`

Examples

```
y <- nv_array(c(1, 0, -1))
x <- nv_array(c(0, 1, 0))
prim_atan2(y, x)
```

`prim_atanh` *Primitive Inverse Hyperbolic Tangent*

Description

Element-wise inverse hyperbolic tangent.

Usage

```
prim_atanh(operand)
```

Arguments

`operand`	(`arrayish`) Arrayish value of data type floating-point.

Value

`arrayish`
Has the same shape and data type as the input. It is ambiguous if the input is ambiguous.

Implemented Rules

- `stablehlo`
- `reverse`

StableHLO

Lowers to `stablehlo::hlo_atanh()`.

See Also

`nv_atanh()`, `atanh()`

Examples

```
x <- nv_array(c(-0.5, 0, 0.5))
prim_atanh(x)
```

`prim_bitcast_convert` *Primitive Bitcast Convert*

Description

Reinterprets the bits of an array as a different data type without modifying the underlying data.

Usage

```
prim_bitcast_convert(operand, dtype)
```

Arguments

operand	(arrayish) Arrayish value of any data type.
dtype	(character(1) | tengen::DataType) Target data type. If it has the same bit width as the input, the output shape is unchanged. If narrower, an extra trailing dimension is added. If wider, the last dimension is consumed.

Value

arrayish
Has the given dtype.

Implemented Rules

- stablehlo
- reverse

StableHLO

Lowers to stablehlo::hlo_bitcast_convert().

See Also

nv_bitcast_convert()

Examples

```
x <- nv_array(1L)
prim_bitcast_convert(x, dtype = "i8")
x <- nv_array(rep(1L, 4), dtype = "i8")
prim_bitcast_convert(x, dtype = "i32")
```

prim_broadcast_in_dim *Primitive Broadcast*

Description

Broadcasts an array to a new shape by replicating the data along new or size-1 dimensions.

Usage

```
prim_broadcast_in_dim(operand, shape, broadcast_dimensions)
```

Arguments

operand	(arrayish) Arrayish value of any data type.
shape	(integer()) Target shape. Each mapped dimension must either match the corresponding operand dimension or the operand dimension must be 1.
broadcast_dimensions	(integer()) Maps each dimension of operand to a dimension of the output. Must have length equal to the number of dimensions of operand.

Value

arrayish
Has the same data type as the input and the given shape. It is ambiguous if the input is ambiguous.

Implemented Rules

- stablehlo
- quickr
- reverse

StableHLO

Lowers to stablehlo::hlo_broadcast_in_dim().

See Also

nv_broadcast_to()

Examples

```
x <- nv_array(c(1, 2, 3))
prim_broadcast_in_dim(x, shape = c(2, 3), broadcast_dimensions = 2L)
```

`prim_cbrt` *Primitive Cube Root*

Description

Element-wise cube root.

Usage

```
prim_cbrt(operand)
```

Arguments

`operand`	(`arrayish`) Arrayish value of data type floating-point.

Value

`arrayish`
Has the same shape and data type as the input. It is ambiguous if the input is ambiguous.

Implemented Rules

- `stablehlo`
- `reverse`

StableHLO

Lowers to `stablehlo::hlo_cbrt()`.

See Also

`nv_cbrt()`

Examples

```
x <- nv_array(c(1, 8, 27))
prim_cbrt(x)
```

`prim_ceil` *Primitive Ceiling*

Description

Element-wise ceiling.

Usage

```
prim_ceil(operand)
```

Arguments

`operand`	(`arrayish`) Arrayish value of data type floating-point.

Value

`arrayish`
Has the same shape and data type as the input. It is ambiguous if the input is ambiguous.

Implemented Rules

- `stablehlo`
- `quickr`
- `reverse`

StableHLO

Lowers to `stablehlo::hlo_ceil()`.

See Also

`nv_ceiling()`, `ceiling()`

Examples

```
x <- nv_array(c(1.2, 2.7, -1.5))
prim_ceil(x)
```

`prim_chol` *Primitive Cholesky Decomposition*

Description

Computes the Cholesky decomposition of a symmetric positive-definite matrix. Dimensions before the last two are batch dimensions.

Usage

```
prim_chol(operand, lower = FALSE)
```

Arguments

`operand`	(`arrayish`) Arrayish value of data type floating-point with at least 2 dimensions. The last two dimensions must be equal (square matrix); any leading dimensions are batch dimensions.
`lower`	(`logical(1)`) If `FALSE` (default, matching base R's `base::chol()`), compute the upper triangular factor `U` such that `operand = t(U) %*% U`. If `TRUE`, compute the lower triangular factor `L` such that `operand = L %*% t(L)`.

Value

`arrayish`
Has the same shape and data type as the input. The values in the triangle not specified by `lower` are implementation-defined. It is ambiguous if the input is ambiguous.

Implemented Rules

- `stablehlo`
- `reverse`

StableHLO

Lowers to `stablehlo::hlo_cholesky()`.

References

Murray, Iain (2016). "Differentiation of the Cholesky decomposition." *arXiv preprint arXiv:1602.07527*.

Walter, Sebastian (2012). *Structured higher-order algorithmic differentiation in the forward and reverse mode with application in optimum experimental design*. Ph.D. thesis, Mathematisch-Naturwissenschaftliche Fakult"at II.

See Also

`nv_solve()`

Examples

```
# Create a positive-definite matrix
x <- nv_matrix(c(4, 2, 2, 3), nrow = 2, dtype = "f32")
prim_chol(x, lower = TRUE)
```

`prim_clamp` *Primitive Clamp*

Description

Clamps every element of `operand` to the range `[min_val, max_val]`, i.e. `max(min_val, min(operand, max_val))`.

Usage

```
prim_clamp(min_val, operand, max_val)
```

Arguments

`min_val`	(`arrayish`) Minimum value. Must be scalar or the same shape as `operand`.
`operand`	(`arrayish`) Arrayish value of any data type.
`max_val`	(`arrayish`) Maximum value. Must be scalar or the same shape as `operand`.

Value

`arrayish`
Has the same data type and shape as `operand`. It is ambiguous if the input is ambiguous.

Implemented Rules

- `stablehlo`
- `reverse`

StableHLO

Lowers to `stablehlo::hlo_clamp()`.

See Also

`nv_clamp()`

Examples

```
x <- nv_array(c(-1, 0.5, 2))
prim_clamp(nv_scalar(0), x, nv_scalar(1))
```

`prim_concatenate` *Primitive Concatenate*

Description

Concatenates arrays along a dimension.

Usage

```
prim_concatenate(..., dimension)
```

Arguments

`...`	(arrayish) Arrays to concatenate. Must all have the same data type, ndims, and shape except along `dimension`.
`dimension`	(`integer(1)`) Dimension along which to concatenate (1-indexed).

Value

arrayish
Has the same data type as the inputs. The output shape matches the inputs in all dimensions except `dimension`, which is the sum of the input sizes along that dimension. It is ambiguous if all inputs are ambiguous.

Implemented Rules

- `stablehlo`
- `quickr`
- `reverse`

StableHLO

Lowers to `stablehlo::hlo_concatenate()`.

See Also

`nv_concatenate()`

Examples

```
x <- nv_array(c(1, 2, 3))
y <- nv_array(c(4, 5, 6))
prim_concatenate(x, y, dimension = 1L)
```

`prim_convert` *Primitive Convert*

Description

Converts the elements of an array to a different data type.

Usage

```
prim_convert(operand, dtype, ambiguous = FALSE)
```

Arguments

operand	(arrayish) Arrayish value of any data type.
dtype	(character(1) | tengen::DataType) Target data type.
ambiguous	(logical(1)) Whether the type is ambiguous. Ambiguous types usually arise from R literals (e.g., 1L, 1.0) and follow special promotion rules. See the vignette("type-promotion") for more details.

Value

arrayish
Has the given dtype and the same shape as operand. Ambiguity is controlled by the ambiguous parameter.

Implemented Rules

- stablehlo
- quickr
- reverse

StableHLO

Lowers to stablehlo::hlo_convert().

See Also

nv_convert()

Examples

```
x <- nv_array(c(1L, 2L, 3L))
prim_convert(x, dtype = "f32")
```

`prim_cos` *Primitive Cosine*

Description

Element-wise cosine.

Usage

```
prim_cos(operand)
```

Arguments

operand	(arrayish) Arrayish value of data type floating-point.

Value

arrayish
Has the same shape and data type as the input. It is ambiguous if the input is ambiguous.

Implemented Rules

- stablehlo
- quickr
- reverse

StableHLO

Lowers to `stablehlo::hlo_cosine()`.

See Also

`nv_cos()`, `cos()`

Examples

```
x <- nv_array(c(0, pi / 2, pi))
prim_cos(x)
```

prim_cosh *Primitive Hyperbolic Cosine*

Description

Element-wise hyperbolic cosine.

Usage

```
prim_cosh(operand)
```

Arguments

operand	(arrayish) Arrayish value of data type floating-point.

Value

arrayish
Has the same shape and data type as the input. It is ambiguous if the input is ambiguous.

Implemented Rules

- stablehlo
- reverse

StableHLO

Lowers to stablehlo::hlo_cosh().

See Also

nv_cosh(), cosh()

Examples

```
x <- nv_array(c(-1, 0, 1))
prim_cosh(x)
```

prim_cummax *Primitive Cumulative Maximum*

Description

Running maximum of array elements along a single dimension along with the index of the last occurrence of the running maximum. At output position `j`, the values output is `max(input[1:j])` and the indices output is the largest `i` in `1:j` with `input[i] == values[j]` (last-occurrence tiebreak).

Usage

```
prim_cummax(operand, dim)
```

Arguments

`operand`	(`arrayish`) Arrayish value of any data type.
`dim`	(`integer(1)`) Dimension along which to accumulate.

Value

`list` of two `arrayish` values:
The running maximum (same dtype as `operand`) and the running argmax (dtype `i32`, 1-based). Both have the same shape as `operand`.

Implemented Rules

- `stablehlo`
- `reverse`

StableHLO

Lowers to a variadic `stablehlo::hlo_reduce_window()` over `(values, iota)`.

See Also

`nv_cummax()`

Examples

```
x <- nv_matrix(c(3, 1, 4, 1, 5, 9), nrow = 2)
prim_cummax(x, dim = 1L)
```

prim_cummin *Primitive Cumulative Minimum*

Description

Running minimum of array elements along a single dimension along with the index of the last occurrence of the running minimum. At output position `j`, the values output is `min(input[1:j])` and the indices output is the largest `i` in `1:j` with `input[i] == values[j]` (last-occurrence tiebreak).

Usage

```
prim_cummin(operand, dim)
```

Arguments

`operand`	(`arrayish`) Arrayish value of any data type.
`dim`	(`integer(1)`) Dimension along which to accumulate.

Value

`list` of two `arrayish` values:
The running minimum (same dtype as `operand`) and the running argmin (dtype `i32`, 1-based). Both have the same shape as `operand`.

Implemented Rules

- `stablehlo`
- `reverse`

StableHLO

Lowers to a variadic `stablehlo::hlo_reduce_window()` over `(values, iota)`.

See Also

`nv_cummin()`

Examples

```
x <- nv_matrix(c(3, 1, 4, 1, 5, 9), nrow = 2)
prim_cummin(x, dim = 1L)
```

`prim_cumprod` *Primitive Cumulative Product*

Description

Cumulative product of array elements along a single dimension. Output position `j` along `dim` equals the product of input positions `1:j`.

Usage

```
prim_cumprod(operand, dim)
```

Arguments

`operand`	(`arrayish`) Arrayish value of any data type.
`dim`	(`integer(1)`) Dimension along which to accumulate.

Value

`arrayish`
Has the same shape and data type as the input. It is ambiguous if the input is ambiguous.

Implemented Rules

- `stablehlo`

StableHLO

Lowers to `stablehlo::hlo_reduce_window()` with `stablehlo::hlo_multiply()` as the reducer.

See Also

`nv_cumprod()`

Examples

```
x <- nv_matrix(1:6, nrow = 2)
prim_cumprod(x, dim = 1L)
```

`prim_cumsum` *Primitive Cumulative Sum*

Description

Cumulative sum of array elements along a single dimension. Output position `j` along `dim` equals the sum of input positions `1:j`.

Usage

```
prim_cumsum(operand, dim)
```

Arguments

`operand`	(arrayish) Arrayish value of any data type.
`dim`	(integer(1)) Dimension along which to accumulate.

Value

arrayish
Has the same shape and data type as the input. It is ambiguous if the input is ambiguous.

Implemented Rules

- `stablehlo`
- `reverse`

StableHLO

Lowers to `stablehlo::hlo_reduce_window()` with `stablehlo::hlo_add()` as the reducer.

See Also

`nv_cumsum()`

Examples

```
x <- nv_matrix(1:6, nrow = 2)
prim_cumsum(x, dim = 1L)
```

prim_digamma *Primitive Digamma*

Description

Element-wise digamma function (logarithmic derivative of the gamma function).

Usage

```
prim_digamma(operand)
```

Arguments

operand	(arrayish) Arrayish value of data type floating-point.

Value

arrayish
Has the same shape and data type as the input. It is ambiguous if the input is ambiguous.

Implemented Rules

- stablehlo
- reverse

StableHLO

Lowers to stablehlo::hlo_digamma().

See Also

nv_digamma(), digamma()

Examples

```
x <- nv_array(c(0.5, 1, 2, 5))
prim_digamma(x)
```

prim_div *Primitive Division*

Description

Divides two arrays element-wise.

Usage

```
prim_div(lhs, rhs)
```

Arguments

lhs, rhs	(arrayish) Arrayish values of data type integer, unsigned integer, or floating-point. Must have the same shape.

Value

arrayish
Has the same shape and data type as the inputs. It is ambiguous if both inputs are ambiguous.

Implemented Rules

- stablehlo
- quickr
- reverse

StableHLO

Lowers to stablehlo::hlo_divide().

See Also

nv_div(), /

Examples

```
x <- nv_array(c(10, 20, 30))
y <- nv_array(c(2, 5, 10))
prim_div(x, y)
```

`prim_dot_general` *Primitive Dot General*

Description

General dot product of two arrays, supporting contraction over arbitrary dimensions and batching.

Usage

```
prim_dot_general(lhs, rhs, contracting_dims, batching_dims)
```

Arguments

`lhs`, `rhs`	(`arrayish`) Left and right operand. Operands are promoted to a common data type. Scalars are broadcast to the shape of the other operand.
`contracting_dims`	(`list(integer(), integer())`) A list of two integer vectors specifying which dimensions of `lhs` and `rhs` to contract over. The contracted dimensions must have matching sizes.
`batching_dims`	(`list(integer(), integer())`) A list of two integer vectors specifying which dimensions of `lhs` and `rhs` are batch dimensions. These must have matching sizes.

Value

`arrayish`
The output shape is the batch dimensions followed by the remaining (non-contracted, non-batched) dimensions of `lhs`, then `rhs`.

Implemented Rules

- `stablehlo`
- `quickr`
- `reverse`

StableHLO

Lowers to `stablehlo::hlo_dot_general()`.

See Also

`nv_matmul()`, `%*%`

Examples

```
x <- nv_matrix(1:6, nrow = 2)
y <- nv_matrix(1:6, nrow = 3)
prim_dot_general(x, y,
  contracting_dims = list(2L, 1L),
  batching_dims = list(integer(0), integer(0))
)
```

`prim_dynamic_slice` *Primitive Dynamic Slice*

Description

Extracts a slice from an array whose start position is determined at runtime via array-valued indices. The slice shape (`slice_sizes`) is a fixed R integer vector.

Use `prim_static_slice()` instead when all indices are known at compile time and you need stride support.

Usage

```
prim_dynamic_slice(operand, ..., slice_sizes)
```

Arguments

`operand`	(`arrayish`) Arrayish value of any data type.
`...`	(`arrayish` of integer type) Scalar start indices, one per dimension. Each must be a scalar array. Pass one scalar per dimension of `operand`.
`slice_sizes`	(`integer()`) Size of the slice in each dimension. Must have length equal to `ndims(operand)` and satisfy `1 <= slice_sizes <= nv_shape(operand)` per dimension.

Value

`arrayish`
Has the same data type as the input and shape `slice_sizes`. It is ambiguous if the input is ambiguous.

Out Of Bounds Behavior

Start indices are clamped before the slice is extracted: `adjusted_start_indices = clamp(1, start_indices, nv_shape(operand) - slice_sizes + 1)`. This means that out-of-bounds indices will not cause an error, but the effective start position may differ from the requested one.

Implemented Rules

- stablehlo
- quickr
- reverse

StableHLO

Lowers to stablehlo::hlo_dynamic_slice().

See Also

prim_static_slice(), prim_dynamic_update_slice(), prim_scatter(), prim_gather(), nv_subset(), [

Examples

```
# 1-D: extract 3 elements starting at position 3
x <- nv_array(1:10)
start <- nv_scalar(3L)
prim_dynamic_slice(x, start, slice_sizes = 3L)

# 2-D: extract a 2x2 block from a matrix
x <- nv_matrix(1:12, nrow = 3, ncol = 4)
row_start <- nv_scalar(2L)
col_start <- nv_scalar(1L)
prim_dynamic_slice(x, row_start, col_start, slice_sizes = c(2L, 2L))
```

---

prim_dynamic_update_slice

Primitive Dynamic Update Slice

Description

Returns a copy of operand with a slice replaced by update at a runtime-determined position. This is the write counterpart of prim_dynamic_slice(): dynamic slice reads a block from an array, while dynamic update slice writes a block into an array.

Usage

```
prim_dynamic_update_slice(operand, update, ...)
```

Arguments

operand	(arrayish) Arrayish value of any data type.
update	(arrayish) The values to write at the specified position. Must have the same data type and number of dimensions as operand, with nv_shape(update) <= nv_shape(operand) per dimension.
...	(arrayish of integer type) Scalar start indices, one per dimension of operand. Each must be a scalar array.

Value

arrayish
Has the same data type and shape as operand. It is ambiguous if the input is ambiguous.

Implemented Rules

- stablehlo
- quickr
- reverse

StableHLO

Lowers to stablehlo::hlo_dynamic_update_slice().

Out Of Bounds Behavior

Start indices are clamped before the slice is extracted: adjusted_start_indices = clamp(1, start_indices, nv_shape(operand) - slice_sizes + 1). This means that out-of-bounds indices will not cause an error, but the effective start position may differ from the requested one.

See Also

prim_dynamic_slice(), prim_scatter(), prim_gather(), nv_subset_assign(), [<-

Examples

```
# 1-D: overwrite two elements starting at position 2
x <- nv_array(1:5)
update <- nv_array(c(10L, 20L))
start <- nv_scalar(2L)
prim_dynamic_update_slice(x, update, start)

# 2-D: write a 2x2 block into a 3x4 matrix
x <- nv_fill(0L, shape = c(3, 4))
update <- nv_matrix(c(1L, 2L, 3L, 4L), nrow = 2, ncol = 2)
row_start <- nv_scalar(2L)
col_start <- nv_scalar(3L)
prim_dynamic_update_slice(x, update, row_start, col_start)
```

prim_eigh *Primitive Symmetric Eigendecomposition*

Description

Computes the eigendecomposition of a symmetric matrix `operand` of shape `(n, n)`:

$$A = \text{vectors}\,\text{diag}(\text{values})\,\text{vectors}^\top.$$

Only the lower triangle of `operand` is read. The columns of `vectors` are the (orthonormal) eigenvectors and `values` is the length-`n` vector of (real) eigenvalues in ascending order. Output names and order match `base::eigen()`.

Usage

```
prim_eigh(operand)
```

Arguments

`operand`	(`arrayish`) Symmetric square matrix of floating-point data type.

Value

Named `list` with elements `values` (length `n`) and `vectors` (shape `(n, n)`). Both have the same dtype as the input.

Implemented Rules

- `stablehlo`

StableHLO

Lowers to `stablehlo::hlo_custom_call()` with target `"eigh"`.

See Also

`nv_eigh()`

Examples

```
x <- nv_array(c(2, 1, 1, 2), shape = c(2, 2), dtype = "f64")
prim_eigh(x)
```

prim_eq *Primitive Equal*

Description

Element-wise equality comparison.

Usage

```
prim_eq(lhs, rhs)
```

Arguments

lhs, rhs	(arrayish) Arrayish values of any data type. Must have the same shape.

Value

arrayish
Has the same shape as the inputs and boolean data type. It is ambiguous if both inputs are ambiguous.

Implemented Rules

- stablehlo
- quickr
- reverse

StableHLO

Lowers to stablehlo::hlo_compare() with comparison_direction = "EQ".

See Also

nv_eq(), ==

Examples

```
x <- nv_array(c(1, 2, 3))
y <- nv_array(c(1, 3, 2))
prim_eq(x, y)
```

`prim_erf` *Primitive Error Function*

Description

Element-wise error function `erf(x) = (2 / sqrt(pi)) * integral_0^x exp(-t^2) dt`.

Usage

```
prim_erf(operand)
```

Arguments

`operand`	(`arrayish`) Arrayish value of data type floating-point.

Value

`arrayish`
Has the same shape and data type as the input. It is ambiguous if the input is ambiguous.

Implemented Rules

- `stablehlo`
- `reverse`

StableHLO

Lowers to `stablehlo::hlo_erf()`.

See Also

`nv_erf()`

Examples

```
x <- nv_array(c(-1, 0, 1))
prim_erf(x)
```

prim_erf_inv *Primitive Inverse Error Function*

Description

Element-wise inverse error function (the inverse of `erf` on `(-1, 1)`).

Usage

```
prim_erf_inv(operand)
```

Arguments

operand	(arrayish) Arrayish value of data type floating-point.

Value

arrayish
Has the same shape and data type as the input. It is ambiguous if the input is ambiguous.

Implemented Rules

- `stablehlo`
- `reverse`

StableHLO

Lowers to `stablehlo::hlo_erf_inv()`.

See Also

`nv_erf_inv()`

Examples

```
x <- nv_array(c(-0.5, 0, 0.5))
prim_erf_inv(x)
```

prim_erfc *Primitive Complementary Error Function*

Description

Element-wise complementary error function `erfc(x) = 1 - erf(x)`.

Usage

```
prim_erfc(operand)
```

Arguments

operand	(arrayish) Arrayish value of data type floating-point.

Value

arrayish
Has the same shape and data type as the input. It is ambiguous if the input is ambiguous.

Implemented Rules

- stablehlo
- reverse

StableHLO

Lowers to `stablehlo::hlo_erfc()`.

See Also

`nv_erfc()`

Examples

```
x <- nv_array(c(-1, 0, 1))
prim_erfc(x)
```

prim_exp *Primitive Exponential*

Description

Element-wise exponential.

Usage

```
prim_exp(operand)
```

Arguments

operand	(arrayish) Arrayish value of data type floating-point.

Value

arrayish
Has the same shape and data type as the input. It is ambiguous if the input is ambiguous.

Implemented Rules

- stablehlo
- quickr
- reverse

StableHLO

Lowers to stablehlo::hlo_exponential().

See Also

nv_exp(), exp()

Examples

```
x <- nv_array(c(0, 1, 2))
prim_exp(x)
```

prim_expm1 *Primitive Exponential Minus One*

Description

Element-wise exp(x) - 1, more accurate for small x.

Usage

```
prim_expm1(operand)
```

Arguments

operand	(arrayish) Arrayish value of data type floating-point.

Value

arrayish
Has the same shape and data type as the input. It is ambiguous if the input is ambiguous.

Implemented Rules

- stablehlo
- quickr
- reverse

StableHLO

Lowers to stablehlo::hlo_exponential_minus_one().

See Also

nv_expm1()

Examples

```
x <- nv_array(c(0, 0.001, 1))
prim_expm1(x)
```

prim_fill *Primitive Fill*

Description

Creates an array of a given shape and data type, filled with a scalar value. The advantage of using this function instead of e.g. doing `nv_array(1, shape = c(100, 100))` is that lowering of `prim_fill()` is efficiently represented in the compiled program, while the latter uses 100 * 100 * 4 bytes of memory.

Usage

```
prim_fill(value, shape, dtype, ambiguous = FALSE, device = NULL)
```

Arguments

value	(`numeric(1)`) Scalar value to fill the array with.
shape	(`integer()`) Shape of the output array.
dtype	(`character(1)` | `tengen::DataType`) Data type.
ambiguous	(`logical(1)`) Whether the type is ambiguous. Ambiguous types usually arise from R literals (e.g., `1L`, `1.0`) and follow special promotion rules. See the `vignette("type-promotion")` for more details.
device	(`character(1)` | `PJRTDevice` | `quickr_device` | `NULL`) Device for data to live on.

Value

`arrayish`
Has the given shape and dtype.

Implemented Rules

- `stablehlo`
- `quickr`

StableHLO

Lowers to `stablehlo::hlo_tensor()`.

See Also

`nv_fill()`

Examples

```
prim_fill(3.14, shape = c(2, 3), dtype = "f32")
```

`prim_floor` *Primitive Floor*

Description

Element-wise floor.

Usage

```
prim_floor(operand)
```

Arguments

`operand`	(`arrayish`) Arrayish value of data type floating-point.

Value

`arrayish`
Has the same shape and data type as the input. It is ambiguous if the input is ambiguous.

Implemented Rules

- `stablehlo`
- `quickr`
- `reverse`

StableHLO

Lowers to `stablehlo::hlo_floor()`.

See Also

`nv_floor()`, `floor()`

Examples

```
x <- nv_array(c(1.2, 2.7, -1.5))
prim_floor(x)
```

prim_gather *Primitive Gather*

Description

Gathers slices from the operand array at positions specified by start_indices. Each index vector in start_indices identifies a starting position in operand, and a slice of size slice_sizes is extracted from that position. The gathered slices are assembled into the output array.

This is the inverse of prim_scatter(): gather reads slices from a array at given indices, while scatter writes slices into an array at given indices.

Usage

```
prim_gather(
  operand,
  start_indices,
  slice_sizes,
  offset_dims,
  collapsed_slice_dims,
  operand_batching_dims,
  start_indices_batching_dims,
  start_index_map,
  index_vector_dim,
  indices_are_sorted = FALSE,
  unique_indices = FALSE
)
```

Arguments

operand	(arrayish) Arrayish value of any data type.
start_indices	(arrayish of integer type) Array of starting indices. Contains index vectors that map to positions in operand via start_index_map. The dimension specified by index_vector_dim holds the index vectors.
slice_sizes	(integer()) Size of the slice to gather from operand in each dimension. Must have length equal to ndims(operand).
offset_dims	(integer()) Dimensions in the output that correspond to the non-collapsed slice dimensions of operand.
collapsed_slice_dims	(integer()) Dimensions of operand that are collapsed (removed) from the slice. The corresponding entries in slice_sizes must be 1. Together with offset_dims and operand_batching_dims, these must account for all dimensions of operand.

operand_batching_dims
: (integer())
Dimensions of operand that are batch dimensions. Use integer(0) when there are no batch dimensions.

start_indices_batching_dims
: (integer())
Dimensions of start_indices that correspond to batch dimensions. Must have the same length as operand_batching_dims.

start_index_map
: (integer())
Maps each component of the index vector to an operand dimension. For example, start_index_map = c(1L) means each index vector indexes into the first dimension of operand.

index_vector_dim
: (integer(1))
Dimension of start_indices that contains the index vectors. If set to ndims(start_indices) + 1, each scalar element of start_indices is treated as a length-1 index vector.

indices_are_sorted
: (logical(1))
Whether indices are guaranteed to be sorted. Setting to TRUE may improve performance but produces undefined behavior if the indices are not actually sorted. Default FALSE.

unique_indices
: (logical(1))
Whether indices are guaranteed to be unique (no duplicates). Setting to TRUE may improve performance but produces undefined behavior if the indices are not actually unique. Default FALSE.

Value

arrayish
: Has the same data type as operand. The output shape is composed of the offset dimensions (from the slice) and the remaining dimensions from start_indices. See the underluing stableHLO function for more details.

Out Of Bounds Behavior

Start indices are clamped before the slice is extracted: clamp(1, start_index, nv_shape(operand) - slice_sizes + 1). This means that out-of-bounds indices will not cause an error, but the effective start position may differ from the requested one.

Implemented Rules

- stablehlo
- quickr
- reverse

StableHLO

Lowers to stablehlo::hlo_gather().

See Also

prim_scatter(), nv_subset(), nv_subset_assign(), [, [<-

Examples

```
# Gather rows 1 and 3 from a 3x3 matrix
operand <- nv_matrix(1:9, nrow = 3)
indices <- nv_matrix(c(1L, 3L), ncol = 1)
prim_gather(
  operand, indices,
  slice_sizes = c(1L, 3L),
  offset_dims = 2L,
  collapsed_slice_dims = 1L,
  operand_batching_dims = integer(0),
  start_indices_batching_dims = integer(0),
  start_index_map = 1L,
  index_vector_dim = 2L
)
```

---

prim_ge *Primitive Greater Equal*

Description

Element-wise greater than or equal comparison.

Usage

```
prim_ge(lhs, rhs)
```

Arguments

lhs, rhs	(arrayish) Arrayish values of any data type. Must have the same shape.

Value

arrayish
Has the same shape as the inputs and boolean data type. It is ambiguous if both inputs are ambiguous.

Implemented Rules

- stablehlo
- quickr
- reverse

StableHLO

Lowers to stablehlo::hlo_compare() with comparison_direction = "GE".

See Also

nv_ge(), >=

Examples

```
x <- nv_array(c(1, 2, 3))
y <- nv_array(c(3, 2, 1))
prim_ge(x, y)
```

---

prim_gt *Primitive Greater Than*

Description

Element-wise greater than comparison.

Usage

```
prim_gt(lhs, rhs)
```

Arguments

lhs, rhs	(arrayish) Arrayish values of any data type. Must have the same shape.

Value

arrayish
Has the same shape as the inputs and boolean data type. It is ambiguous if both inputs are ambiguous.

Implemented Rules

- stablehlo
- quickr
- reverse

StableHLO

Lowers to stablehlo::hlo_compare() with comparison_direction = "GT".

See Also

nv_gt(), >

Examples

```
x <- nv_array(c(1, 2, 3))
y <- nv_array(c(3, 2, 1))
prim_gt(x, y)
```

---

prim_if *Primitive If*

Description

Conditional execution of one of two branches based on a scalar boolean predicate. Unlike prim_ifelse() which operates element-wise, this evaluates only the selected branch.

Usage

```
prim_if(pred, true, false)
```

Arguments

pred	(arrayish) Scalar boolean predicate that determines which branch to execute.
true, false	(function()) Zero-argument functions for the true and false branches. Both must return outputs with the same structure, dtypes, and shapes.

Value

Result of the executed branch.
An output is ambiguous if it is ambiguous in both branches.

Implemented Rules

- stablehlo
- quickr
- reverse

StableHLO

Lowers to stablehlo::hlo_if().

See Also

nv_if(), prim_ifelse()

Examples

```
prim_if(nv_scalar(TRUE), \() nv_scalar(1), \() nv_scalar(2))
```

`prim_ifelse` *Primitive Ifelse*

Description

Element-wise selection based on a boolean predicate, like R's `ifelse()`. For each element, returns the corresponding element from `true_value` where `pred` is `TRUE` and from `false_value` where `pred` is `FALSE`.

Usage

```
prim_ifelse(pred, true_value, false_value)
```

Arguments

`pred`	(`arrayish` of boolean type) Predicate array. Must be scalar or have the same shape as `true_value`.
`true_value, false_value`	(`arrayish`) Values to select from. Must have the same dtype and shape.

Value

`arrayish`
Has the same dtype and shape as `true_value`. It is ambiguous if both `true_value` and `false_value` are ambiguous.

Implemented Rules

- `stablehlo`
- `quickr`
- `reverse`

StableHLO

Lowers to `stablehlo::hlo_select()`.

See Also

`nv_ifelse()`

Examples

```
pred <- nv_array(c(TRUE, FALSE, TRUE))
prim_ifelse(pred, nv_array(c(1, 2, 3)), nv_array(c(4, 5, 6)))
```

`prim_iota` *Primitive Iota*

Description

Creates an array with values increasing along the specified dimension.

Usage

```
prim_iota(dim, dtype, shape, start = 1L, ambiguous = FALSE, device = NULL)
```

Arguments

`dim`	(`integer(1)`) Dimension along which values increase (1-indexed).
`dtype`	(`character(1)` | `tengen::DataType`) Data type.
`shape`	(`integer()`) Shape of the output array.
`start`	(`integer(1)`) Starting value.
`ambiguous`	(`logical(1)`) Whether the type is ambiguous. Ambiguous types usually arise from R literals (e.g., `1L`, `1.0`) and follow special promotion rules. See the `vignette("type-promotion")` for more details.
`device`	(`character(1)` | `PJRTDevice` | `quickr_device` | `NULL`) Device for data to live on.

Value

`arrayish`
Has the given `dtype` and `shape`.

Implemented Rules

- `stablehlo`
- `quickr`

StableHLO

Lowers to `stablehlo::hlo_iota()`.

See Also

nv_iota()

Examples

```
prim_iota(dim = 1L, dtype = "i32", shape = 5L)
```

---

prim_is_finite *Primitive Is Finite*

Description

Element-wise check if values are finite (not Inf, -Inf, or NaN).

Usage

```
prim_is_finite(operand)
```

Arguments

operand	(arrayish) Arrayish value of data type floating-point.

Value

arrayish
Has the same shape as the input and boolean data type. It is ambiguous if the input is ambiguous.

Implemented Rules

- stablehlo
- reverse

StableHLO

Lowers to stablehlo::hlo_is_finite().

See Also

nv_is_finite()

Examples

```
x <- nv_array(c(1, Inf, NaN, -Inf, 0))
prim_is_finite(x)
```

prim_le *Primitive Less Equal*

Description

Element-wise less than or equal comparison.

Usage

```
prim_le(lhs, rhs)
```

Arguments

lhs, rhs	(arrayish) Arrayish values of any data type. Must have the same shape.

Value

arrayish
Has the same shape as the inputs and boolean data type. It is ambiguous if both inputs are ambiguous.

Implemented Rules

- stablehlo
- quickr
- reverse

StableHLO

Lowers to stablehlo::hlo_compare() with comparison_direction = "LE".

See Also

nv_le(), <=

Examples

```
x <- nv_array(c(1, 2, 3))
y <- nv_array(c(3, 2, 1))
prim_le(x, y)
```

prim_lgamma *Primitive Log-Gamma*

Description

Element-wise natural logarithm of the absolute value of the gamma function.

Usage

```
prim_lgamma(operand)
```

Arguments

operand	(arrayish) Arrayish value of data type floating-point.

Value

arrayish
Has the same shape and data type as the input. It is ambiguous if the input is ambiguous.

Implemented Rules

- stablehlo
- reverse

StableHLO

Lowers to stablehlo::hlo_lgamma().

See Also

nv_lgamma(), lgamma()

Examples

```
x <- nv_array(c(0.5, 1, 2, 5))
prim_lgamma(x)
```

prim_log *Primitive Logarithm*

Description

Element-wise natural logarithm.

Usage

```
prim_log(operand)
```

Arguments

operand	(arrayish) Arrayish value of data type floating-point.

Value

arrayish
Has the same shape and data type as the input. It is ambiguous if the input is ambiguous.

Implemented Rules

- stablehlo
- quickr
- reverse

StableHLO

Lowers to stablehlo::hlo_log().

See Also

nv_log(), log()

Examples

```
x <- nv_array(c(1, 2.718, 7.389))
prim_log(x)
```

prim_log1p *Primitive Log Plus One*

Description

Element-wise log(1 + x), more accurate for small x.

Usage

```
prim_log1p(operand)
```

Arguments

operand	(arrayish) Arrayish value of data type floating-point.

Value

arrayish
Has the same shape and data type as the input. It is ambiguous if the input is ambiguous.

Implemented Rules

- stablehlo
- quickr
- reverse

StableHLO

Lowers to stablehlo::hlo_log_plus_one().

See Also

nv_log1p()

Examples

```
x <- nv_array(c(0, 0.001, 1))
prim_log1p(x)
```

`prim_logistic` *Primitive Logistic (Sigmoid)*

Description

Element-wise logistic sigmoid: 1 / (1 + exp(-x)).

Usage

```
prim_logistic(operand)
```

Arguments

`operand`	(`arrayish`) Arrayish value of data type floating-point.

Value

`arrayish`
Has the same shape and data type as the input. It is ambiguous if the input is ambiguous.

Implemented Rules

- `stablehlo`
- `quickr`
- `reverse`

StableHLO

Lowers to `stablehlo::hlo_logistic()`.

See Also

`nv_logistic()`

Examples

```
x <- nv_array(c(-2, 0, 2))
prim_logistic(x)
```

prim_lt *Primitive Less Than*

Description

Element-wise less than comparison.

Usage

```
prim_lt(lhs, rhs)
```

Arguments

lhs, rhs	(arrayish) Arrayish values of any data type. Must have the same shape.

Value

arrayish
Has the same shape as the inputs and boolean data type. It is ambiguous if both inputs are ambiguous.

Implemented Rules

- stablehlo
- quickr
- reverse

StableHLO

Lowers to stablehlo::hlo_compare() with comparison_direction = "LT".

See Also

nv_lt(), <

Examples

```
x <- nv_array(c(1, 2, 3))
y <- nv_array(c(3, 2, 1))
prim_lt(x, y)
```

prim_lu *Primitive LU Decomposition*

Description

Computes the partial-pivoted LU decomposition of a matrix `operand`:

$$PA = LU,$$

where P is a permutation matrix, L is unit lower triangular, and U is upper triangular. `L` (with implicit unit diagonal) and `U` are packed into a single `LU` output matching LAPACK's `getrf` layout. P is returned in two equivalent forms: `pivots` (LAPACK's sequential row-swap encoding) and `permutation` (an explicit permutation vector).

Usage

```
prim_lu(operand)
```

Arguments

`operand`	(`arrayish`) Matrix of data type floating-point with exactly 2 dimensions.

Value

`list` of three `arrayish` values: `LU` `(m, n)` with the same dtype as the input; `pivots` `(k,)` of dtype `i32` with `k = min(m, n)` (1-based row swaps such that row `i` was exchanged with row `pivots[i]` during elimination step `i`); and `permutation` `(m,)` of dtype `i32`, a 1-based permutation vector for P such that `(P %*% A)[i, ]` equals `A[permutation[i], ]`.

Implemented Rules

- `stablehlo`

StableHLO

Lowers to a `"lu"` `stablehlo::hlo_custom_call()` (backed by LAPACK on CPU and cuSOLVER on CUDA) for `LU` and `pivots`, followed by a `stablehlo::hlo_while()` loop that converts `pivots` to `permutation` in-graph.

See Also

`nv_lu()`

Examples

```
x <- nv_matrix(c(4, 3, 6, 3), nrow = 2, dtype = "f64")
prim_lu(x)
```

`prim_max` *Primitive Maximum*

Description

Element-wise maximum of two arrays.

Usage

```
prim_max(lhs, rhs)
```

Arguments

`lhs`, `rhs`	(`arrayish`) Arrayish values of any data type. Must have the same shape.

Value

`arrayish`
Has the same shape and data type as the inputs. It is ambiguous if both inputs are ambiguous.

Implemented Rules

- `stablehlo`
- `quickr`
- `reverse`

StableHLO

Lowers to `stablehlo::hlo_maximum()`.

See Also

`nv_max()`

Examples

```
x <- nv_array(c(1, 5, 3))
y <- nv_array(c(4, 2, 6))
prim_max(x, y)
```

prim_min *Primitive Minimum*

Description

Element-wise minimum of two arrays.

Usage

```
prim_min(lhs, rhs)
```

Arguments

lhs, rhs	(arrayish) Arrayish values of any data type. Must have the same shape.

Value

arrayish
Has the same shape and data type as the inputs. It is ambiguous if both inputs are ambiguous.

Implemented Rules

- stablehlo
- quickr
- reverse

StableHLO

Lowers to stablehlo::hlo_minimum().

See Also

nv_min()

Examples

```
x <- nv_array(c(1, 5, 3))
y <- nv_array(c(4, 2, 6))
prim_min(x, y)
```

`prim_mul` *Primitive Multiplication*

Description

Multiplies two arrays element-wise.

Usage

```
prim_mul(lhs, rhs)
```

Arguments

`lhs`, `rhs`	(`arrayish`) Arrayish values of any data type. Must have the same shape.

Value

`arrayish`
Has the same shape and data type as the inputs. It is ambiguous if both inputs are ambiguous.

Implemented Rules

- `stablehlo`
- `quickr`
- `reverse`

StableHLO

Lowers to `stablehlo::hlo_multiply()`.

See Also

`nv_mul()`, `*`

Examples

```
x <- nv_array(c(1, 2, 3))
y <- nv_array(c(4, 5, 6))
prim_mul(x, y)
```

prim_ne *Primitive Not Equal*

Description

Element-wise inequality comparison.

Usage

```
prim_ne(lhs, rhs)
```

Arguments

lhs, rhs	(arrayish) Arrayish values of any data type. Must have the same shape.

Value

arrayish
Has the same shape as the inputs and boolean data type. It is ambiguous if both inputs are ambiguous.

Implemented Rules

- stablehlo
- quickr
- reverse

StableHLO

Lowers to stablehlo::hlo_compare() with comparison_direction = "NE".

See Also

nv_ne(), !=

Examples

```
x <- nv_array(c(1, 2, 3))
y <- nv_array(c(1, 3, 2))
prim_ne(x, y)
```

prim_negate *Primitive Negation*

Description

Negates an array element-wise.

Usage

```
prim_negate(operand)
```

Arguments

operand	(arrayish) Arrayish value of data type integer or floating-point.

Value

arrayish
Has the same shape and data type as the input. It is ambiguous if the input is ambiguous.

Implemented Rules

- stablehlo
- quickr
- reverse

StableHLO

Lowers to stablehlo::hlo_negate().

See Also

nv_negate(), unary -

Examples

```
x <- nv_array(c(1, -2, 3))
prim_negate(x)
```

prim_not *Primitive Not*

Description

Element-wise logical NOT.

Usage

```
prim_not(operand)
```

Arguments

operand	(arrayish) Arrayish value of data type boolean, integer, or unsigned integer.

Value

arrayish
Has the same shape and data type as the input. It is ambiguous if the input is ambiguous.

Implemented Rules

- stablehlo
- quickr
- reverse

StableHLO

Lowers to stablehlo::hlo_not().

See Also

nv_not()

Examples

```
x <- nv_array(c(TRUE, FALSE, TRUE))
prim_not(x)
```

prim_or *Primitive Or*

Description

Element-wise logical OR.

Usage

```
prim_or(lhs, rhs)
```

Arguments

lhs, rhs	(arrayish) Arrayish values of data type boolean, integer, or unsigned integer. Must have the same shape.

Value

arrayish
Has the same shape and data type as the inputs. It is ambiguous if both inputs are ambiguous.

Implemented Rules

- stablehlo
- quickr
- reverse

StableHLO

Lowers to stablehlo::hlo_or().

See Also

nv_or(), |

Examples

```
x <- nv_array(c(TRUE, FALSE, TRUE))
y <- nv_array(c(TRUE, TRUE, FALSE))
prim_or(x, y)
```

prim_pad *Primitive Pad*

Description

Pads an array with a given padding value.

Usage

```
prim_pad(
  operand,
  padding_value,
  edge_padding_low,
  edge_padding_high,
  interior_padding
)
```

Arguments

operand	(arrayish) Arrayish value of any data type.
padding_value	(arrayish) Scalar value to use for padding. Must have the same dtype as operand.
edge_padding_low	(integer()) Amount of padding to add at the start of each dimension.
edge_padding_high	(integer()) Amount of padding to add at the end of each dimension.
interior_padding	(integer()) Amount of padding to add between elements in each dimension.

Value

arrayish
Has the same data type as operand. For the output shape see the underlying stablehlo documentation (stablehlo::hlo_pad()). It is ambiguous if the input is ambiguous.

Implemented Rules

- stablehlo
- quickr
- reverse

StableHLO

Lowers to stablehlo::hlo_pad().

Examples

```
x <- nv_array(c(1, 2, 3))
prim_pad(x, nv_scalar(0),
  edge_padding_low = 2L, edge_padding_high = 1L, interior_padding = 0L
)
```

---

prim_polygamma *Primitive Polygamma*

Description

Element-wise polygamma function: the (n+1)-th derivative of the log-gamma function. Both n and x must have the same shape; n typically holds non-negative integer values.

Usage

```
prim_polygamma(n, x)
```

Arguments

n, x	(arrayish) Arrayish values of data type floating-point. Must have the same shape.

Value

arrayish
Has the same shape and data type as the inputs. It is ambiguous if both inputs are ambiguous.

Implemented Rules

- stablehlo
- reverse

StableHLO

Lowers to stablehlo::hlo_polygamma().

See Also

nv_polygamma()

Examples

```
n <- nv_array(c(1, 1, 2))
x <- nv_array(c(0.5, 1, 2))
prim_polygamma(n, x)
```

`prim_popcnt` *Primitive Population Count*

Description

Element-wise population count (number of set bits).

Usage

```
prim_popcnt(operand)
```

Arguments

operand	(arrayish) Arrayish value of data type integer or unsigned integer.

Value

arrayish
Has the same shape and data type as the input. It is ambiguous if the input is ambiguous.

Implemented Rules

- stablehlo
- reverse

StableHLO

Lowers to `stablehlo::hlo_popcnt()`.

See Also

`nv_popcnt()`

Examples

```
x <- nv_array(c(7L, 3L, 15L))
prim_popcnt(x)
```

prim_pow *Primitive Power*

Description

Raises lhs to the power of rhs element-wise.

Usage

```
prim_pow(lhs, rhs)
```

Arguments

lhs, rhs	(arrayish) Arrayish values of data type integer, unsigned integer, or floating-point. Must have the same shape.

Value

arrayish
Has the same shape and data type as the inputs. It is ambiguous if both inputs are ambiguous.

Implemented Rules

- stablehlo
- quickr
- reverse

StableHLO

Lowers to stablehlo::hlo_power().

See Also

nv_pow(), ^

Examples

```
x <- nv_array(c(2, 3, 4))
y <- nv_array(c(3, 2, 1))
prim_pow(x, y)
```

prim_print *Primitive Print*

Description

Prints an array value to the console during execution and returns the input unchanged. This is useful for debugging JIT-compiled code.

Usage

```
prim_print(operand)
```

Arguments

operand	(arrayish) Arrayish value of any data type.

Value

arrayish
Returns operand as-is.

Implemented Rules

- stablehlo

StableHLO

Lowers to stablehlo::hlo_custom_call().

See Also

nv_print()

Examples

```
x <- nv_array(c(1, 2, 3))
prim_print(x)
```

prim_qr *Primitive QR Decomposition*

Description

Computes the reduced QR decomposition of a matrix `operand`:

$$A = QR,$$

where Q has orthonormal columns ($Q^\top Q = I$) and R is upper triangular. For an $m \times n$ input with $k = \min(m, n)$, Q has shape $m \times k$ and R has shape $k \times n$.

Usage

```
prim_qr(operand)
```

Arguments

`operand`	(`arrayish`) Matrix of data type floating-point with exactly 2 dimensions.

Value

Named `list` with elements `Q` (shape `(m, k)`) and `R` (shape `(k, n)`), where `(m, n) = shape(operand)` and `k = min(m, n)`. Both have the same data type as `operand`.

Implemented Rules

- `stablehlo`

StableHLO

Lowers to a `"geqrf"` + `"orgqr"` `stablehlo::hlo_custom_call()` pair (backed by LAPACK on CPU and cuSOLVER on CUDA) + postprocessing.

See Also

`nv_qr()`

Examples

```
x <- nv_array(1:6, shape = c(3, 2), dtype = "f32")
prim_qr(x)
```

`prim_reduce` *Primitive Generic Reduce*

Description

Reduces an array along the specified dimensions using a user-supplied associative reducer.

Usage

```
prim_reduce(operand, init, dims, drop = TRUE, reductor)
```

Arguments

`operand`	(`arrayish`) Arrayish value of any data type.
`init`	(`arrayish`) Scalar (0-dimensional) initial value. Must have the same data type as `operand` and be the neutral element w.r.t. `reductor`.
`dims`	(`integer()`) Dimensions to reduce over.
`drop`	(`logical(1)`) If `TRUE` (default) the reduced dimensions are removed; if `FALSE` they are kept with size 1.
`reductor`	(`function(lhs, rhs)`) Binary reducer producing a scalar of the same dtype as `operand`. Must be associative (see "Associativity Requirement").

Value

`arrayish`
Same data type as `operand`. Shape is `operand` with `dims` removed (or set to 1 if `drop = FALSE`).

Associativity Requirement

The order in which `reductor` is applied across the reduction window is implementation-defined. If the reductor is not associative, the result is ill-defined. Furthermore, `init` must be the neutral element for this reductor. Because floating point math is non-associative, the output of the reduction can differ between backends (GPU, CPU), even if the underlying mathematical function (like `+`) is associative.

Implemented Rules

- `stablehlo`

StableHLO

Lowers to `stablehlo::hlo_reduce()` with `reductor` as the body.

See Also

prim_reduce_sum(), prim_reduce_max()

Examples

```
x <- nv_array(c(1, 2, 3, 4))
prim_reduce(x, init = nv_scalar(0), dims = 1L, reductor = prim_add)
prim_reduce(x, init = nv_scalar(1), dims = 1L, reductor = prim_mul)
```

---

prim_reduce_all *Primitive All Reduction*

Description

Performs logical AND along the specified dimensions.

Usage

```
prim_reduce_all(operand, dims, drop = TRUE)
```

Arguments

operand	(arrayish) Arrayish value of boolean data type.
dims	(integer()) Dimensions to reduce over.
drop	(logical(1)) Whether to drop the reduced dimensions from the output shape. If TRUE, the reduced dimensions are removed. If FALSE, the reduced dimensions are set to 1.

Value

arrayish
Boolean array. Never ambiguous. When drop = TRUE, the shape is that of operand with dims removed. When drop = FALSE, the shape is that of operand with dims set to 1.

Implemented Rules

- stablehlo
- quickr
- reverse

StableHLO

Lowers to stablehlo::hlo_reduce() with stablehlo::hlo_and() as the reducer.

See Also

nv_reduce_all()

Examples

```
x <- nv_matrix(c(TRUE, FALSE, TRUE, TRUE), nrow = 2)
prim_reduce_all(x, dims = 1L)
```

---

prim_reduce_any *Primitive Any Reduction*

Description

Performs logical OR along the specified dimensions.

Usage

```
prim_reduce_any(operand, dims, drop = TRUE)
```

Arguments

operand	(arrayish) Arrayish value of boolean data type.
dims	(integer()) Dimensions to reduce over.
drop	(logical(1)) Whether to drop the reduced dimensions from the output shape. If TRUE, the reduced dimensions are removed. If FALSE, the reduced dimensions are set to 1.

Value

arrayish
Boolean array. Never ambiguous. When drop = TRUE, the shape is that of operand with dims removed. When drop = FALSE, the shape is that of operand with dims set to 1.

Implemented Rules

- stablehlo
- quickr
- reverse

StableHLO

Lowers to stablehlo::hlo_reduce() with stablehlo::hlo_or() as the reducer.

See Also

nv_reduce_any()

Examples

```
x <- nv_matrix(c(TRUE, FALSE, TRUE, TRUE), nrow = 2)
prim_reduce_any(x, dims = 1L)
```

---

prim_reduce_max *Primitive Max Reduction*

Description

Finds the maximum of array elements along the specified dimensions.

Usage

```
prim_reduce_max(operand, dims, drop = TRUE)
```

Arguments

operand	(arrayish) Arrayish value of any data type.
dims	(integer()) Dimensions to reduce over.
drop	(logical(1)) Whether to drop the reduced dimensions from the output shape. If TRUE, the reduced dimensions are removed. If FALSE, the reduced dimensions are set to 1.

Value

arrayish
Has the same data type as the input. When drop = TRUE, the shape is that of operand with dims removed. When drop = FALSE, the shape is that of operand with dims set to 1. It is ambiguous if the input is ambiguous.

Implemented Rules

- stablehlo
- quickr
- reverse

StableHLO

Lowers to stablehlo::hlo_reduce() with stablehlo::hlo_maximum() as the reducer.

See Also

nv_reduce_max()

Examples

```
x <- nv_matrix(1:6, nrow = 2)
prim_reduce_max(x, dims = 1L)
```

---

prim_reduce_min *Primitive Min Reduction*

Description

Finds the minimum of array elements along the specified dimensions.

Usage

```
prim_reduce_min(operand, dims, drop = TRUE)
```

Arguments

operand	(arrayish) Arrayish value of any data type.
dims	(integer()) Dimensions to reduce over.
drop	(logical(1)) Whether to drop the reduced dimensions from the output shape. If TRUE, the reduced dimensions are removed. If FALSE, the reduced dimensions are set to 1.

Value

arrayish
Has the same data type as the input. When drop = TRUE, the shape is that of operand with dims removed. When drop = FALSE, the shape is that of operand with dims set to 1. It is ambiguous if the input is ambiguous.

Implemented Rules

- stablehlo
- quickr
- reverse

StableHLO

Lowers to stablehlo::hlo_reduce() with stablehlo::hlo_minimum() as the reducer.

See Also

nv_reduce_min()

Examples

```
x <- nv_matrix(1:6, nrow = 2)
prim_reduce_min(x, dims = 1L)
```

---

prim_reduce_prod *Primitive Product Reduction*

Description

Multiplies array elements along the specified dimensions.

Usage

```
prim_reduce_prod(operand, dims, drop = TRUE)
```

Arguments

operand	(arrayish) Arrayish value of any data type.
dims	(integer()) Dimensions to reduce over.
drop	(logical(1)) Whether to drop the reduced dimensions from the output shape. If TRUE, the reduced dimensions are removed. If FALSE, the reduced dimensions are set to 1.

Value

arrayish
Has the same data type as the input. When drop = TRUE, the shape is that of operand with dims removed. When drop = FALSE, the shape is that of operand with dims set to 1. It is ambiguous if the input is ambiguous.

Implemented Rules

- stablehlo
- quickr
- reverse

StableHLO

Lowers to stablehlo::hlo_reduce() with stablehlo::hlo_multiply() as the reducer.

See Also

nv_reduce_prod()

Examples

```
x <- nv_matrix(1:6, nrow = 2)
prim_reduce_prod(x, dims = 1L)
```

---

prim_reduce_sum *Primitive Sum Reduction*

Description

Sums array elements along the specified dimensions.

Usage

```
prim_reduce_sum(operand, dims, drop = TRUE)
```

Arguments

operand	(arrayish) Arrayish value of any data type.
dims	(integer()) Dimensions to reduce over.
drop	(logical(1)) Whether to drop the reduced dimensions from the output shape. If TRUE, the reduced dimensions are removed. If FALSE, the reduced dimensions are set to 1.

Value

arrayish
Has the same data type as the input. When drop = TRUE, the shape is that of operand with dims removed. When drop = FALSE, the shape is that of operand with dims set to 1. It is ambiguous if the input is ambiguous.

Implemented Rules

- stablehlo
- quickr
- reverse

StableHLO

Lowers to stablehlo::hlo_reduce() with stablehlo::hlo_add() as the reducer.

See Also

nv_reduce_sum()

Examples

```
x <- nv_matrix(1:6, nrow = 2)
prim_reduce_sum(x, dims = 1L)
```

---

prim_remainder *Primitive Remainder*

Description

Element-wise remainder. Result has sign of the divident, which differs from base R's %%, which is available via nv_mod() and has sign of divisor.

Usage

```
prim_remainder(lhs, rhs)
```

Arguments

lhs, rhs	(arrayish) Arrayish values of data type integer, unsigned integer, or floating-point. Must have the same shape.

Value

arrayish
Has the same shape and data type as the inputs. It is ambiguous if both inputs are ambiguous.

Implemented Rules

- stablehlo
- reverse

StableHLO

Lowers to stablehlo::hlo_remainder().

See Also

nv_remainder()

Examples

```
prim_remainder(1, -3)
1 %% -3
```

prim_reshape *Primitive Reshape*

Description

Reshapes an array to a new shape without changing the underlying data. Note that row-major order is used, which differs from R's column-major order.

Usage

```
prim_reshape(operand, shape)
```

Arguments

operand	(arrayish) Arrayish value of any data type.
shape	(integer()) Target shape. Must have the same number of elements as operand.

Value

arrayish
Has the same data type as the input and the given shape. It is ambiguous if the input is ambiguous.

Implemented Rules

- stablehlo
- quickr
- reverse

StableHLO

Lowers to stablehlo::hlo_reshape().

See Also

nv_reshape()

Examples

```
x <- nv_array(1:6)
prim_reshape(x, shape = c(2, 3))
```

prim_reverse *Primitive Reverse*

Description

Reverses the order of elements along specified dimensions.

Usage

```
prim_reverse(operand, dims)
```

Arguments

operand	(arrayish) Arrayish value of any data type.
dims	(integer()) Dimensions to reverse (1-indexed).

Value

arrayish
Has the same data type and shape as operand. It is ambiguous if the input is ambiguous.

Implemented Rules

- stablehlo
- quickr
- reverse

StableHLO

Lowers to stablehlo::hlo_reverse().

See Also

nv_reverse()

Examples

```
x <- nv_array(c(1, 2, 3, 4, 5))
prim_reverse(x, dims = 1L)
```

`prim_rng_bit_generator` *Primitive RNG Bit Generator*

Description

Generates pseudo-random numbers using the specified algorithm and returns the updated RNG state together with the generated values.

Usage

```
prim_rng_bit_generator(
  initial_state,
  rng_algorithm = "THREE_FRY",
  dtype,
  shape
)
```

Arguments

`initial_state`	(`arrayish`) RNG state (`ui64[2]`).
`rng_algorithm`	(`character(1)`) RNG algorithm name. Default is `"THREE_FRY"`.
`dtype`	(`character(1)` | `tengen::DataType`) Data type of the generated random values.
`shape`	(`integer()`) Shape.

Value

`list` of two `arrayish` values:
The first element is the updated RNG state with the same dtype and shape as `initial_state`. The second element is an array of random values with the given `dtype` and `shape`.

Implemented Rules

- `stablehlo`

StableHLO

Lowers to `stablehlo::hlo_rng_bit_generator()`.

See Also

`nv_runif()`, `nv_rnorm()`

Examples

```
state <- nv_array(c(0L, 0L), dtype = "ui64")
prim_rng_bit_generator(state, dtype = "f32", shape = c(3, 2))
```

---

prim_round *Primitive Round*

Description

Rounds the elements of an array to the nearest integer.

Usage

```
prim_round(operand, method = "nearest_even")
```

Arguments

operand	(arrayish) Arrayish value of data type floating-point.
method	(character(1)) Rounding method. "nearest_even" (default) rounds to the nearest even integer on a tie, "afz" rounds away from zero on a tie.

Value

arrayish
Has the same dtype and shape as operand. It is ambiguous if the input is ambiguous.

Implemented Rules

- stablehlo
- reverse

StableHLO

Lowers to stablehlo::hlo_round_nearest_even() or stablehlo::hlo_round_nearest_afz() depending on the method parameter.

See Also

nv_round()

Examples

```
x <- nv_array(c(1.4, 2.5, 3.6))
prim_round(x)
```

prim_rsqrt *Primitive Reciprocal Square Root*

Description

Element-wise reciprocal square root.

Usage

```
prim_rsqrt(operand)
```

Arguments

operand	(arrayish) Arrayish value of data type floating-point.

Value

arrayish
Has the same shape and data type as the input. It is ambiguous if the input is ambiguous.

Implemented Rules

- stablehlo
- reverse

StableHLO

Lowers to stablehlo::hlo_rsqrt().

See Also

nv_rsqrt()

Examples

```
x <- nv_array(c(1, 4, 9))
prim_rsqrt(x)
```

prim_scatter *Primitive Scatter*

Description

Produces a result array identical to `input` except that slices at positions specified by `scatter_indices` are updated with values from the `update` array. When multiple indices point to the same location, the `update_computation` function determines how to combine the values (by default the new value replaces the old one).

This is the inverse of `prim_gather()`: gather reads slices from an array at given indices, while scatter writes slices into an array at given indices.

Usage

```
prim_scatter(
  input,
  scatter_indices,
  update,
  update_window_dims,
  inserted_window_dims,
  input_batching_dims,
  scatter_indices_batching_dims,
  scatter_dims_to_operand_dims,
  index_vector_dim,
  indices_are_sorted = FALSE,
  unique_indices = FALSE,
  update_computation = NULL
)
```

Arguments

`input` (`arrayish`)
Arrayish value of any data type. The base array to scatter into.

`scatter_indices` (`arrayish` of integer type)
Array of indices. Contains index vectors that map to positions in `input` via `scatter_dims_to_operand_dims`. The dimension specified by `index_vector_dim` holds the index vectors.

`update` (`arrayish`)
Update values array. Must have the same data type as `input`.

`update_window_dims` (`integer()`)
Dimensions of `update` that are window dimensions, i.e. they correspond to the slice being written into `input`.

inserted_window_dims
: (integer())
Dimensions of input whose slices have size 1 and are inserted (not present) in the update window. Together with update_window_dims and input_batching_dims, these must account for all dimensions of input.

input_batching_dims
: (integer())
Dimensions of input that are batch dimensions. Use integer(0) when there are no batch dimensions.

scatter_indices_batching_dims
: (integer())
Dimensions of scatter_indices that correspond to batch dimensions. Must have the same length as input_batching_dims.

scatter_dims_to_operand_dims
: (integer())
Maps each component of the index vector to an input dimension. For example, scatter_dims_to_operand_dims = c(1L) means each index vector indexes into the first dimension of input.

index_vector_dim
: (integer(1))
Dimension of scatter_indices that contains the index vectors. If set to ndims(scatter_indices) + 1, each scalar element of scatter_indices is treated as a length-1 index vector.

indices_are_sorted
: (logical(1))
Whether indices are guaranteed to be sorted. Setting to TRUE may improve performance but produces undefined behavior if the indices are not actually sorted. Default FALSE.

unique_indices (logical(1))
: Whether indices are guaranteed to be unique (no duplicates). Setting to TRUE may improve performance but produces undefined behavior if the indices are not actually unique. Default FALSE.

update_computation
: (function)
Binary function f(old, new) that combines the existing value in input with the value from update. The default (NULL) uses function(old, new) new, which replaces the old value.

Value

arrayish
: Has the same data type and shape as input. It is ambiguous if input is ambiguous.

Out Of Bounds Behavior

If a computed result index falls outside the bounds of input, the update for that index is silently ignored.

Update Order

When multiple indices in `scatter_indices` map to the same element of `input`, the order in which `update_computation` is applied is implementation-defined and may vary between plugins ("cpu", "cuda").

Implemented Rules

- `stablehlo`
- `quickr`
- `reverse`

StableHLO

Lowers to `stablehlo::hlo_scatter()`.

See Also

`prim_gather()`, `nv_subset()`, `nv_subset_assign()`, `[`, `[<-`

Examples

```
# Scatter values 10 and 30 into positions 1 and 3 of a zero vector
input <- nv_array(c(0, 0, 0, 0, 0))
indices <- nv_matrix(c(1L, 3L), ncol = 1)
updates <- nv_array(c(10, 30))
prim_scatter(
  input, indices, updates,
  update_window_dims = integer(0),
  inserted_window_dims = 1L,
  input_batching_dims = integer(0),
  scatter_indices_batching_dims = integer(0),
  scatter_dims_to_operand_dims = 1L,
  index_vector_dim = 2L
)
```

`prim_shift_left` *Primitive Shift Left*

Description

Element-wise left bit shift.

Usage

```
prim_shift_left(lhs, rhs)
```

Arguments

lhs, rhs	(arrayish) Arrayish values of data type boolean, integer, or unsigned integer. Must have the same shape.

Value

arrayish
Has the same shape and data type as the inputs. It is ambiguous if both inputs are ambiguous.

Implemented Rules

- stablehlo
- reverse

StableHLO

Lowers to stablehlo::hlo_shift_left().

See Also

nv_shift_left()

Examples

```
x <- nv_array(c(1L, 2L, 4L))
y <- nv_array(c(1L, 2L, 1L))
prim_shift_left(x, y)
```

---

prim_shift_right_arithmetic

Primitive Arithmetic Shift Right

Description

Element-wise arithmetic right bit shift.

Usage

```
prim_shift_right_arithmetic(lhs, rhs)
```

Arguments

lhs, rhs	(arrayish) Arrayish values of data type boolean, integer, or unsigned integer. Must have the same shape.

Value

arrayish
Has the same shape and data type as the inputs. It is ambiguous if both inputs are ambiguous.

Implemented Rules

- stablehlo
- reverse

StableHLO

Lowers to stablehlo::hlo_shift_right_arithmetic().

See Also

nv_shift_right_arithmetic()

Examples

```
x <- nv_array(c(8L, -16L, 32L))
y <- nv_array(c(1L, 2L, 3L))
prim_shift_right_arithmetic(x, y)
```

---

prim_shift_right_logical

Primitive Logical Shift Right

Description

Element-wise logical right bit shift.

Usage

```
prim_shift_right_logical(lhs, rhs)
```

Arguments

lhs, rhs	(arrayish) Arrayish values of data type boolean, integer, or unsigned integer. Must have the same shape.

Value

arrayish
Has the same shape and data type as the inputs. It is ambiguous if both inputs are ambiguous.

Implemented Rules

- stablehlo
- reverse

StableHLO

Lowers to stablehlo::hlo_shift_right_logical().

See Also

nv_shift_right_logical()

Examples

```
x <- nv_array(c(8L, 16L, 32L))
y <- nv_array(c(1L, 2L, 3L))
prim_shift_right_logical(x, y)
```

---

prim_sign *Primitive Sign*

Description

Element-wise sign.

Usage

```
prim_sign(operand)
```

Arguments

operand	(arrayish) Arrayish value of data type signed integer or floating-point.

Value

arrayish
Has the same shape and data type as the input. It is ambiguous if the input is ambiguous.

Implemented Rules

- stablehlo
- reverse

StableHLO

Lowers to stablehlo::hlo_sign().

See Also

nv_sign(), sign()

Examples

```
x <- nv_array(c(-3, 0, 5))
prim_sign(x)
```

---

prim_sin *Primitive Sine*

Description

Element-wise sine.

Usage

```
prim_sin(operand)
```

Arguments

operand	(arrayish) Arrayish value of data type floating-point.

Value

arrayish
Has the same shape and data type as the input. It is ambiguous if the input is ambiguous.

Implemented Rules

- stablehlo
- quickr
- reverse

StableHLO

Lowers to stablehlo::hlo_sine().

See Also

nv_sin(), sin()

Examples

```
x <- nv_array(c(0, pi / 2, pi))
prim_sin(x)
```

prim_sinh *Primitive Hyperbolic Sine*

Description

Element-wise hyperbolic sine.

Usage

```
prim_sinh(operand)
```

Arguments

operand	(arrayish) Arrayish value of data type floating-point.

Value

arrayish
Has the same shape and data type as the input. It is ambiguous if the input is ambiguous.

Implemented Rules

- stablehlo
- reverse

StableHLO

Lowers to stablehlo::hlo_sinh().

See Also

nv_sinh(), sinh()

Examples

```
x <- nv_array(c(-1, 0, 1))
prim_sinh(x)
```

prim_sort *Primitive Sort*

Description

Sorts arrays along the given dimension.

Sorting is determined by the *first operand* only: it is the sort key, and any additional operands are reordered with the same permutation that sorts the first. This enables idioms like *argsort* (sort `x` paired with an `iota` and read off the second output) and key-value sorts (sort `keys` paired with `values`).

All operands must have the same shape; their dtypes may differ. 1-dimensional slices along `dim` are sorted independently; other dimensions are preserved.

Usage

```
prim_sort(operands, dim = 1L, descending = FALSE, is_stable = FALSE)
```

Arguments

`operands`	(`list` of `arrayish`) One or more arrays to sort. The first is the sort key; the rest are carried along under the same permutation. All must share the same shape.
`dim`	(`integer(1)`) Dimension along which to sort.
`descending`	(`logical(1)`) If `TRUE`, sort the key in descending order (largest first). Default `FALSE`. Additional operands are reordered by the same permutation regardless.
`is_stable`	(`logical(1)`) If `TRUE`, the sort is stable: the relative order of equal *keys* is preserved. Default `FALSE`.

Value

`list` of `arrayish`
One sorted output per element of `operands`, in the same order. Each output has the same shape, data type, and ambiguity as the corresponding input.

Implemented Rules

- `stablehlo`
- `reverse`

StableHLO

Lowers to `stablehlo::hlo_sort()` with a comparator that uses `stablehlo::hlo_compare()` (`LT` for ascending, `GT` for descending) on the first operand. For float keys the comparator uses `compare_type = "TOTALORDER"` and canonicalizes `-0`/`+0` and `-NaN`/`+NaN` to their positive form before comparing, so all `NaN` values land at one end of the result regardless of sign. Integer keys use `SIGNED` / `UNSIGNED` as appropriate.

See Also

`nv_sort()`, `nv_argsort()`, `nv_top_k()`, `nv_median()`

Examples

```
x <- nv_array(c(3, 1, 4, 1, 5))
prim_sort(list(x), dim = 1L)[[1L]]

# Sort indices by the values (argsort): pair x with iota and read off
# the second result.
idx <- nv_iota(dim = 1L, dtype = "i64", shape = 5L)
out <- prim_sort(list(x, idx), dim = 1L)
out[[1L]] # sorted x
out[[2L]] # permutation indices
```

`prim_sqrt` *Primitive Square Root*

Description

Element-wise square root.

Usage

```
prim_sqrt(operand)
```

Arguments

`operand`	(`arrayish`) Arrayish value of data type floating-point.

Value

`arrayish`
Has the same shape and data type as the input. It is ambiguous if the input is ambiguous.

Implemented Rules

- stablehlo
- quickr
- reverse

StableHLO

Lowers to stablehlo::hlo_sqrt().

See Also

nv_sqrt(), sqrt()

Examples

```
x <- nv_array(c(1, 4, 9))
prim_sqrt(x)
```

---

prim_static_slice *Primitive Static Slice*

Description

Extracts a slice from an array using static (compile-time) indices. All indices, limits, and strides are fixed R integers.

Use prim_dynamic_slice() instead when the start position must be computed at runtime (e.g. depends on array values).

Usage

```
prim_static_slice(operand, start_indices, limit_indices, strides)
```

Arguments

operand	(arrayish) Arrayish value of any data type.
start_indices	(integer()) Start indices (inclusive), one per dimension. Must satisfy 1 <= start_indices <= limit_indices per dimension.
limit_indices	(integer()) End indices (inclusive), one per dimension. Must satisfy limit_indices <= nv_shape(operand) per dimension.
strides	(integer()) Step sizes, one per dimension. Must be >= 1. A stride of 1 selects every element; a stride of 2 selects every other element, etc.

Value

arrayish
Has the same data type as the input and shape `ceiling((limit_indices - start_indices + 1) / strides)`. It is ambiguous if the input is ambiguous.

Implemented Rules

- `stablehlo`
- `quickr`
- `reverse`

StableHLO

Lowers to `stablehlo::hlo_slice()`.

See Also

`prim_dynamic_slice()`, `prim_scatter()`, `prim_gather()`, `nv_subset()`, `[`

Examples

```
# 1-D: extract elements 2 through 4 (limit is exclusive)
x <- nv_array(1:10)
prim_static_slice(x, start_indices = 2L, limit_indices = 5L, strides = 1L)

# 1-D: every other element using strides
x <- nv_array(1:10)
prim_static_slice(x, start_indices = 1L, limit_indices = 10L, strides = 2L)

# 2-D: extract a submatrix (rows 1-2, columns 2-3)
x <- nv_matrix(1:12, nrow = 3, ncol = 4)
prim_static_slice(x,
  start_indices = c(1L, 2L),
  limit_indices = c(3L, 4L),
  strides       = c(1L, 1L)
)
```

---

prim_sub *Primitive Subtraction*

Description

Subtracts two arrays element-wise.

Usage

```
prim_sub(lhs, rhs)
```

Arguments

lhs, rhs	(arrayish) Arrayish values of data type integer, unsigned integer, or floating-point. Must have the same shape.

Value

arrayish
Has the same shape and data type as the inputs. It is ambiguous if both inputs are ambiguous.

Implemented Rules

- stablehlo
- quickr
- reverse

StableHLO

Lowers to stablehlo::hlo_subtract().

See Also

nv_sub(), -

Examples

```
x <- nv_array(c(1, 2, 3))
y <- nv_array(c(4, 5, 6))
prim_sub(x, y)
```

---

prim_svd *Primitive Singular Value Decomposition*

Description

Computes the reduced ("economy") singular value decomposition of a matrix operand of shape (m, n):

$$A = u \operatorname{diag}(d)\, vt,$$

where u has orthonormal columns, vt has orthonormal rows, and d is the length-k (k = min(m, n)) vector of non-negative singular values in descending order.

Note: unlike base::svd(), which returns the right singular vectors as v of shape (n, k) (so that a = u %*% diag(d) %*% t(v)), this primitive returns them already transposed as vt of shape (k, n) (matching the underlying LAPACK / cuSOLVER output and avoiding an extra transpose).

Supports any matrix shape on both the host (LAPACK gesdd) and CUDA (cuSOLVER gesvd) backends. cuSOLVER's m >= n requirement is handled transparently via a layout flip for wide matrices.

Usage

```
prim_svd(operand)
```

Arguments

operand	(arrayish) Matrix of data type floating-point with exactly 2 dimensions.

Value

Named `list` with elements `d` (length `k`), `u` (shape `(m, k)`), and `vt` (shape `(k, n)`). All have the same dtype as the input.

Implemented Rules

- `stablehlo`

StableHLO

Lowers to `stablehlo::hlo_custom_call()` with target `"svd"`.

See Also

`nv_svd()`

Examples

```
x <- nv_array(c(1, 0, 0, 1, 0, 1), shape = c(3, 2))
prim_svd(x)
```

---

prim_tan *Primitive Tangent*

Description

Element-wise tangent.

Usage

```
prim_tan(operand)
```

Arguments

operand	(arrayish) Arrayish value of data type floating-point.

Value

arrayish
Has the same shape and data type as the input. It is ambiguous if the input is ambiguous.

Implemented Rules

- stablehlo
- quickr
- reverse

StableHLO

Lowers to stablehlo::hlo_tan().

See Also

nv_tan(), tan()

Examples

```
x <- nv_array(c(0, 0.5, 1))
prim_tan(x)
```

---

prim_tanh *Primitive Hyperbolic Tangent*

Description

Element-wise hyperbolic tangent.

Usage

```
prim_tanh(operand)
```

Arguments

operand	(arrayish) Arrayish value of data type floating-point.

Value

arrayish
Has the same shape and data type as the input. It is ambiguous if the input is ambiguous.

Implemented Rules

- stablehlo
- quickr
- reverse

StableHLO

Lowers to stablehlo::hlo_tanh().

See Also

nv_tanh(), tanh()

Examples

```
x <- nv_array(c(-1, 0, 1))
prim_tanh(x)
```

---

prim_top_k *Primitive Top-K*

Description

Returns the k largest values along the last dimension, sorted in descending order, together with their indices into that dimension.

For other dimensions, transpose so the target dimension is last, call prim_top_k(), then transpose back. nv_top_k() does this.

Usage

```
prim_top_k(operand, k)
```

Arguments

operand	(arrayish) Tensor of integer, unsigned integer, or floating-point dtype with rank >= 1.
k	(integer(1)) Number of top elements. Must satisfy 1 <= k <= shape(operand)[ndims(operand)].

Value

list of two arrayish values:
The top-k values (same dtype as operand) and their indices along the last dimension (dtype i32, matching JAX). Both have the same shape as operand with the last dimension replaced by k. Ties are broken by lower index first.

Implemented Rules

- stablehlo
- reverse

StableHLO

Lowers to stablehlo::hlo_top_k().

See Also

nv_top_k(), prim_sort()

Examples

```
x <- nv_array(c(3, 1, 4, 1, 5, 9, 2, 6))
prim_top_k(x, k = 3L)
```

---

prim_transpose *Primitive Transpose*

Description

Permutes the dimensions of an array.

Usage

```
prim_transpose(operand, permutation)
```

Arguments

operand	(arrayish) Arrayish value of any data type.
permutation	(integer()) Specifies the new ordering of dimensions. Must be a permutation of seq_len(ndims) where ndims is the number of dimensions of operand.

Value

arrayish
Has the same data type as the input and shape nv_shape(operand)[permutation]. It is ambiguous if the input is ambiguous.

Implemented Rules

- stablehlo
- quickr
- reverse

StableHLO

Lowers to stablehlo::hlo_transpose().

See Also

nv_transpose(), t()

Examples

```
x <- nv_matrix(1:6, nrow = 2)
prim_transpose(x, permutation = c(2L, 1L))
```

---

prim_triangular_solve *Primitive Triangular Solve*

Description

Solves a system of linear equations with a triangular coefficient matrix. When left_side is TRUE, solves op(a) %*% x = b for x. When left_side is FALSE, solves x %*% op(a) = b for x. Dimensions before the last two are batch dimensions and must match between a and b (no broadcasting). Here op is A or A^T depending on transpose_a.

Usage

```
prim_triangular_solve(a, b, left_side, lower, unit_diagonal, transpose_a)
```

Arguments

a	(arrayish) Triangular coefficient matrix of data type floating-point with at least 2 dimensions. The last two dimensions must be equal (square matrix); any leading dimensions are batch dimensions.
b	(arrayish) Right-hand side. Same data type and rank as a (rank >= 2), with matching leading batch dimensions. The size of a's last two (square) dimensions must equal b's second-to-last dimension when left_side = TRUE, or b's last dimension when left_side = FALSE.
left_side	(logical(1)) If TRUE, solve op(a) %*% x = b. If FALSE, solve x %*% op(a) = b.
lower	(logical(1)) If TRUE, a is lower triangular. If FALSE, a is upper triangular.
unit_diagonal	(logical(1)) If TRUE, assume diagonal elements of a are 1.
transpose_a	(logical(1)) If TRUE, solve with t(a) in place of a. Defaults to FALSE.

Value

arrayish
Has the same shape and data type as b. It is ambiguous if both a and b are ambiguous.

Implemented Rules

- stablehlo
- reverse

StableHLO

Lowers to stablehlo::hlo_triangular_solve().

References

Giles, Mike (2008). "An extended collection of matrix derivative results for forward and reverse mode automatic differentiation." Oxford University Computing Laboratory.

See Also

nv_solve()

Examples

```
# Solve L %*% x = b where L is lower triangular
L <- nv_matrix(c(2, 0, 1, 3), nrow = 2, dtype = "f32")
b <- nv_matrix(c(4, 3), nrow = 2, dtype = "f32")
prim_triangular_solve(L, b,
  left_side = TRUE, lower = TRUE,
  unit_diagonal = FALSE, transpose_a = FALSE
)
```

---

prim_while *Primitive While Loop*

Description

Repeatedly executes body while cond returns TRUE, like R's while loop. The loop state is initialized with init and passed through each iteration. Otherwise, no state is maintained between iterations.

Usage

```
prim_while(init, cond, body)
```

Arguments

`init`	(`named list()`) Named list of initial state values.
`cond`	(`function`) Condition function that receives the current state as arguments and outputs whether to continue the loop.
`body`	(`function`) Body function that receives the current state as arguments and returns a named list with the same structure, dtypes, and shapes as `init`.

Value

Named list with the same structure as `init` containing the final state after the loop terminates.

Implemented Rules

- `stablehlo`
- `quickr`

StableHLO

Lowers to `stablehlo::hlo_while()`.

See Also

`nv_while()`

Examples

```
prim_while(
  init = list(i = nv_scalar(0L), total = nv_scalar(0L)),
  cond = function(i, total) i <= 5L,
  body = function(i, total) list(
    i = i + 1L,
    total = total + i
  )
)
```

---

prim_xor *Primitive Xor*

Description

Element-wise logical XOR.

Usage

```
prim_xor(lhs, rhs)
```

Arguments

lhs, rhs	(arrayish) Arrayish values of data type boolean, integer, or unsigned integer. Must have the same shape.

Value

arrayish
Has the same shape and data type as the inputs. It is ambiguous if both inputs are ambiguous.

Implemented Rules

- stablehlo
- quickr
- reverse

StableHLO

Lowers to stablehlo::hlo_xor().

See Also

nv_xor()

Examples

```
x <- nv_array(c(TRUE, FALSE, TRUE))
y <- nv_array(c(TRUE, TRUE, FALSE))
prim_xor(x, y)
```

PrimitiveCall *Primitive Call*

Description

Call of a primitive in an AnvlGraph.

Usage

```
PrimitiveCall(primitive, inputs, params, outputs)
```

Arguments

primitive	(AnvlPrimitive) The function.
inputs	(list(GraphValue)) The (array) inputs to the primitive.
params	(list(<any>)) The (static) parameters of the function call.
outputs	(list(GraphValue)) The (array) outputs of the primitive.

Value

(PrimitiveCall)

---

quickr_device *Quickr device*

Description

Device descriptor for the quickr backend. The only supported type is "cpu".

Usage

```
quickr_device(x = "cpu")
```

Arguments

x	(character(1)) Device type. Currently only supports "cpu".

Value

A QuickrDevice object.

See Also

nv_device(), AnvlBackendQuickr().

reindex_tree *Reindex Tree*

Description

Reassigns leaf indices so they form a contiguous sequence starting from the current counter value. This is used internally after filtering nodes from a tree (e.g. via `filter_list_node()`) to ensure leaf indices still map correctly to positions in a flat list. Not intended for direct use.

Usage

```
reindex_tree(x, counter)
```

Arguments

`x`	(`Node`) A tree node to reindex.
`counter`	(environment) A mutable counter created by `new_counter()`.

Value

A new `Node` with updated leaf indices.

rule_reverse *Reverse Rule*

Description

Construct a reverse-mode autodiff rule for a primitive. The `backward` argument should be provided if the `forward` call for the primitive should run un-modified. This covers most use-cases. The `backward` argument should have this signature: `function(inputs, outputs, grads, params, required) -> list(inpu`

In some scenarios, it can be beneficial to perform a slightly different forward pass to enable a more efficient backward pass. In this case, pass the `forward` argument. It should return a list containing the results from the forward pass, as well as closure that has the same signature as the one above. It can make use of intermediate values computed in the forward pass via lexical scoping.

Usage

```
rule_reverse(backward = NULL, forward = NULL)
```

Arguments

`backward`	(`function`) Backward hook for default case.
`forward`	(`function`) Alternative-forward hook that returns both primals and backward closure.

Value

An anvl_rule_reverse object.

See Also

transform_gradient()

---

shape *Get the shape of an array*

Description

Returns the shape of an array as an integer() vector.

Usage

```
shape(x, ...)
```

Arguments

x	(arrayish) An array-like object.
...	Additional arguments passed to methods (unused).

Details

This is implemented via the generic tengen::shape().

Value

integer()

Examples

```
x <- nv_array(1:4, dtype = "f32")
shape(x)
```

Shape *Create a Shape object*

Description

Constructs a Shape representing array dimensions.

Usage

```
Shape(dims = integer())
```

Arguments

dims	An integer() vector of dimension sizes (>= 0).

Value

A Shape object.

See Also

shape(), stablehlo::Shape()

Examples

```
Shape(c(2L, 3L))
```

stablehlo *Lower a graph to StableHLO*

Description

Converts a traced AnvlGraph into the StableHLO intermediate representation (IR). Each graph operation is translated to its corresponding StableHLO op. The result can be serialized to MLIR text via stablehlo::repr() and subsequently compiled to an XLA executable with pjrt::pjrt_compile().

The rules for translating to stablehlo are stored in $rules[["stablehlo"]] of the primitives.

This is a low-level function; most users should use jit() or xla() instead.

Usage

```
stablehlo(
  graph,
  constants_as_inputs = TRUE,
  env = NULL,
  donate = character(),
  platform = NULL
)
```

Arguments

graph (AnvlGraph)
The graph to lower (e.g. produced by trace_fn()).

constants_as_inputs (logical(1))
If TRUE (default), constants are registered as inputs to the StableHLO function so they can be passed in at execution time. If FALSE, they are not added as inputs. Set to FALSE for closures. Note that GraphLiterals are always inlined into the StableHLO function.

env (HloEnv | NULL)
Optional environment for reusing variable mappings across nested function lowerings (e.g. for higher-order primitives like nv_while).

donate (character())
Names of the arguments whose buffers should be donated. Donated buffers can be aliased with outputs of the same type, enabling in-place operations.

platform (NULL | character(1))
Target platform name (e.g. "cpu", "cuda"). Stored on a process-wide global during the call so that platform-aware lowering rules (queried via current_platform()) can branch on it. NULL (the default) leaves the current value untouched — recursive calls from higher-order primitives inherit the platform of the enclosing call.

Value

A list of length 2:

- the stablehlo::Func
- The list of GraphValues holding ConcreteArrays.

See Also

trace_fn(), jit(), xla(), current_platform()

Examples

```
x <- nv_array(c(1, 2))
graph <- trace_fn(function(y) y + x, list(y = nv_aval("f32", shape = c())))
graph
stablehlo(graph)
```

subgraphs *Get Subgraphs from Higher-Order Primitive*

Description

Extracts subgraphs from the parameters of a higher-order primitive call.

Usage

```
subgraphs(call)
```

Arguments

call	(PrimitiveCall) The primitive call.

Value

(list(AnvlGraph))
List of subgraphs found in the parameters.

to_abstract *Convert to Abstract Array*

Description

Convert an object to its abstract array representation (AbstractArray).

Usage

```
to_abstract(x, pure = FALSE)
```

Arguments

x	(any) Object to convert.
pure	(logical(1)) Whether to convert to a pure AbstractArray and not e.g. LiteralArray or ConcreteArray.

Value

AbstractArray

Examples

```
# R literals become LiteralArrays (ambiguous by default, except logicals)
to_abstract(1.5)
to_abstract(1L)
to_abstract(TRUE)

# AnvlArrays become ConcreteArrays
to_abstract(nv_array(1:4))

# Use pure = TRUE to strip subclass info
to_abstract(nv_array(1:4), pure = TRUE)
```

`trace_fn` *Trace an R function into a Graph*

Description

Executes `f` with abstract array arguments and records every primitive operation into an `AnvlGraph`.

The resulting graph can be lowered to StableHLO (via `stablehlo()`) or transformed (e.g. via `transform_gradient()`).

Usage

```
trace_fn(
  f,
  args = NULL,
  desc = NULL,
  mode = NULL,
  args_flat = NULL,
  in_tree = NULL
)
```

Arguments

`f`	(`function`) The function to trace. Must not be a `JitFunction` (i.e. already jitted).
`args`	(`list` of (`AnvlArray` | `AbstractArray`)) The (unflattened) arguments to the function. Mutually exclusive with the `args_flat`/`in_tree` pair.
`desc`	(`NULL` | `GraphDescriptor`) Optional descriptor. When `NULL` (default), a new descriptor is created.
`mode`	(`character(1)`) How to handle the inputs. Options are: • `"toplevel"`: Used for jit(). Default. • `"subgraph"`: Use for tracing subgraphs in higher-order primitives like `prim_while()`.

- `"inline"`: Use for transformations like jit, where the graph is later inlined into the parent graph.

`args_flat` (`list`)
Flattened arguments. Must be accompanied by `in_tree`.

`in_tree` (`Node`)
Tree structure describing how `args_flat` maps back to `f`'s arguments.

Value

An `AnvlGraph` containing the traced operations.

See Also

`stablehlo()` to lower the graph, `jit()` / `xla()` for end-to-end compilation.

Examples

```
graph <- trace_fn(function(x, y) x + y,
  args = list(x = nv_array(1, dtype = "f32"), y = nv_array(2, dtype = "f32")))
graph
```

---

transform_gradient *Transform a graph to its gradient*

Description

Low-level graph transformation that transforms a graph into its gradient. The function `f` represented by `graph` must return a single float scalar. The resulting graph computes the gradients of that scalar with respect to the inputs specified by `wrt`.

Usage

```
transform_gradient(graph, wrt)
```

Arguments

`graph` (`AnvlGraph`)
The graph to transform. Must produce a single scalar float output.

`wrt` (`character`)
Names of the graph inputs to differentiate with respect to.

Details

To support alternative forward passes for more efficient backward passes, we replay and possibly rewrite the graph into a new descriptor. Afterwards, we traverse it backwards and call the gradient rules where necessary.

See `rule_reverse()` for more information.

This is the building block used by `gradient()` and `value_and_gradient()`; prefer those higher-level wrappers unless you need to operate on graphs directly.

Value

An `AnvlGraph` whose outputs are the requested gradients.

See Also

`gradient()`, `value_and_gradient()`, `rule_reverse()`

Examples

```
graph <- trace_fn(prim_mul, list(nv_aval("f32", c()), nv_aval("f32", c())))
graph
transform_gradient(graph, "lhs")
```

---

tree_path *Tree Path*

Description

Returns the human-readable path for a single leaf node identified by its flat index. Only descends into the branch containing the target leaf, making it efficient for error reporting.

Usage

```
tree_path(node, i, prefix = "")
```

Arguments

`node`	(Node) A tree node as returned by `build_tree()`.
`i`	(`integer(1)`) The flat index of the leaf (as stored in `LeafNode$i`).
`prefix`	(`character(1)`) Path prefix. Used internally during recursion; callers should leave as `""`.

Value

A scalar `character` string.

See Also

build_tree(), flatten()

---

tree_size *Tree Size*

Description

Counts the number of leaf nodes in a tree. This equals the length of the flat list produced by flatten() on the original structure.

Usage

```
tree_size(x)
```

Arguments

x	(Node) A tree node as returned by build_tree().

Value

A scalar integer.

See Also

build_tree(), flatten()

Examples

```
tree <- build_tree(list(a = 1, b = list(c = 2, d = 3)))
tree_size(tree)

tree_size(build_tree(list(1)))
```

unflatten *Unflatten*

Description

Reconstructs a nested structure from a flat list by using a tree previously created with `build_tree()`. Each `LeafNode` in the tree selects the corresponding element from `x` by index, and `ListNodes` restore the original nesting and names.

Usage

```
unflatten(node, x)
```

Arguments

`node`	(`Node`) Tree describing the target structure, as returned by `build_tree()`.
`x`	(list) Flat list of leaf values, typically produced by `flatten()`.

Value

The reconstructed nested structure (list or single value).

See Also

`flatten()`, `build_tree()`

Examples

```
x <- list(a = 1, b = list(c = 2, d = 3))
tree <- build_tree(x)
flat <- flatten(x)

unflatten(tree, flat)

unflatten(tree, list(10, 20, 30))
```

`value_and_gradient` *Value and Gradient*

Description

Returns a new function that computes both the output of `f` and its gradient in a single forward+reverse pass. The result is a named list with elements `value` (the original return value of `f`) and `grad` (the gradients, structured like the inputs or the `wrt` subset).

Usage

```
value_and_gradient(f, wrt = NULL)
```

Arguments

`f`	(`function`) Function to differentiate. Arguments can be arrayish (`AnvlArray`) or static (non-array) values. Must return a single scalar float array.
`wrt`	(`character` | `integer` | `NULL`) Names or positions of the arguments to compute the gradient with respect to. Only arrayish (float array) arguments can be included; static arguments must not appear in `wrt`. If `NULL` (the default), the gradient is computed with respect to all arguments (which must all be arrayish in that case).

Value

A function with the same formals as `f` that returns `list(value = ..., grad = ...)`.

See Also

`gradient()`

Examples

```
loss_fn <- function(x) sum(x^2L)
vg <- jit(value_and_gradient(loss_fn))
result <- vg(nv_array(c(3, 4), dtype = "f32"))
result$value
result$grad
```

vt *Construct a stablehlo ValueType*

Description

Shorthand for building a tensor `stablehlo::ValueType` from a dtype and shape — convenient inside stablehlo lowering rules that need to declare custom-call output types or similar.

Usage

```
vt(dtype, shape)
```

Arguments

dtype	A dtype (string or `tengen::DataType`).
shape	An integer vector or `stablehlo::Shape`.

Value

(ValueType)

vt2at *Convert ValueType to AbstractArray*

Description

Convert a `ValueType` to an `AbstractArray`.

Usage

```
vt2at(x)
```

Arguments

x	(ValueType)

Value

(AbstractArray)

with_backend *Run code with a specific backend*

Description

Sets the `anvl.default_backend` option for the duration of the expression. This affects `jit()` and data construction (e.g. via `nv_array`).

Usage

```
with_backend(backend, code)
```

Arguments

backend	(`character(1)`) Backend to use (`"xla"` or `"quickr"`).
code	An expression to evaluate with the given backend.

Value

The result of evaluating `code`.

xla *Ahead-of-time compile a function to XLA*

Description

Compiles a function to an XLA executable via tracing.

Returns a callable R function that executes the compiled binary. Unlike `jit()`, compilation happens eagerly at definition time rather than on first call, so the input shapes and dtypes must be specified upfront via abstract arrays (see `nv_aval()`).

Usage

```
xla(f, args, donate = character(), device = NULL)
```

Arguments

f	(`function`) Function to compile. Must accept and return `AnvlArray`s.
args	(`list`) List of abstract array specifications (e.g. from `nv_aval()`) describing the expected shapes and dtypes of `f`'s arguments.
donate	(`character()`) Names of the arguments whose buffers should be donated.
device	(`character(1)` | `PJRTDevice`) Target device such as `"cpu"` (default) or `"cuda"`.

Details

Traces `f` with the given abstract `args` (via `trace_fn()`), lowers the resulting graph via `stablehlo()` and then compiles it to an XLA executable via `pjrt::pjrt_compile()`.

Value

(`function`)
A function that accepts `AnvlArray` arguments (matching the flat inputs) and returns the result as `AnvlArray`s.

See Also

`jit()` for lazy compilation, `compile_xla()` for the lower-level API.

Examples

```
f_compiled <- xla(function(x, y) x + y,
  args = list(x = nv_aval("f32", c(2, 2)), y = nv_aval("f32", c(2, 2)))
)
a <- nv_matrix(1:4, nrow = 2, dtype = "f32")
b <- nv_matrix(5:8, nrow = 2, dtype = "f32")
f_compiled(a, b)
```

Index