Vector spaces

Types

```
# Field
abstract type Field end
struct RealNumbers <: Field end
struct ComplexNumbers <: Field end
const ℝ = RealNumbers()
const ℂ = ComplexNumbers()

# Vector Space
abstract type VectorSpace end

## Elementary Space
abstract type ElementarySpace{𝕜} <: VectorSpace end
const IndexSpace = ElementarySpace
struct GeneralSpace{𝕜} <: ElementarySpace{𝕜}
    d::Int
    dual::Bool
    conj::Bool
end

abstract type InnerProductSpace{𝕜} <: ElementarySpace{𝕜} end
abstract type EuclideanSpace{𝕜} <: InnerProductSpace{𝕜} end
struct CartesianSpace <: EuclideanSpace{ℝ}
    d::Int
end
struct ComplexSpace <: EuclideanSpace{ℂ}
  d::Int
  dual::Bool
end
struct GradedSpace{I<:Sector, D} <: EuclideanSpace{ℂ}
    dims::D
    dual::Bool
end

# Composite Space
abstract type CompositeSpace{S<:ElementarySpace} <: VectorSpace end
struct ProductSpace{S<:ElementarySpace, N} <: CompositeSpace{S}

    spaces::NTuple{N, S}
```

```
    spaces::NTuple{N, S}
end
const TensorSpace{S<:ElementarySpace} = Union{S, ProductSpace{S}}

# Space of Morphisms
struct HomSpace{S<:ElementarySpace, P1<:CompositeSpace{S}, P2<:CompositeSpace{S}}
    codomain::P1
    domain::P2
end
const TensorMapSpace{S<:ElementarySpace, N₁, N₂} = HomSpace{S, ProductSpace{S, N₁}, Pr
```

Properties

On both `VectorSpace` instances and types:

```
spacetype # type of ElementarySpace associated with a composite space or a tensor or a
field # field of a vector space or a tensor map or a HomSpace
Base.oneunit # the corresponding vector space that represents the trivial 1D space isor
sectortype # sector type of a space or a tensor or a HomSpace
one(::S) where {S<:ElementarySpace} -> ProductSpace{S, 0}
one(::ProductSpace{S}) where {S<:ElementarySpace} -> ProductSpace{S, 0}  # Return a ter
```

On `VectorSpace` instances:

```
sectors # an iterator over the different sectors of an ElementarySpace
sectors(P::ProductSpace{S, N}) # Return an iterator over all possible combinations of s
blocksectors(V::ElementarySpace) = sectors(V) # make ElementarySpace instances behave s
blocksectors(P::ProductSpace) # Return an iterator over the different unique coupled se
blocksectors(W::HomSpace) # Return an iterator over the different unique coupled secto
blocksectors(t::TensorMap) # Return an iterator over the different unique coupled secto
dim # total dimension of a vector space or a product space
dim(V::ElementarySpace, ::Trivial) # return dim(V)
dim(V::GradedSpace, c::I) # the degeneracy or multiplicity of sector c in a Graded Spa
dim(W::HomSpace) # Return the total dimension of a `HomSpace`, i.e. the number of line
dim(P::ProductSpace, n::Int) # dim for the `n`th vector space of the product space
dim(P::ProductSpace{S, N}, s::NTuple{N, sectortype(S)}) # Return the total degeneracy d
dim(t::AbstractTensorMap) # total dim for corresponding HomSpace
dim(t::TensorMap) # Return the total dimension of the tensor map, i.e., the number of d
dims(P::ProductSpace) # Return the dimensions of the spaces in the tensor product spac
dims(P::ProductSpace{S, N}, s::NTuple{N, sectortype(S)}) # Return the degeneracy dimen
blockdim(V::ElementarySpace, c::Sector) = dim(V, c) # make ElementarySpace instances be
blockdim(P::ProductSpace, c::Sector) # Return the total dimension of a coupled sector
hassector(V::ElementarySpace, a::Sector) # whether a vector space `V` has a subspace co
hassector(P::ProductSpace{S, N}, s::NTuple{N, sectortype(S)}) # Query whether `P` has a
Base.axes(V::ElementarySpace) # the axes of an elementary space as `1:dim(V)`
```

```
Base.axes(V::ElementarySpace) # the axes of an elementary space as `1:dim(V)`
Base.axes(V::ElementarySpace, a::Sector) # axes corresponding to the sector `a` in an e
Base.axes(P::ProductSpace) # the axes for all index spaces in product space `P`
Base.axes(P::ProductSpace, n::Int) # the axes for `n`th index space of product space `P
Base.axes(P::ProductSpace{<:ElementarySpace, N}, sectors::NTuple{N, <:Sector}) where {N
Base.conj(V::ElementarySpace) # returns the complex conjugate space (conj(V)==V̄)
dual(V::EuclideanSpace) = conj(V) # returns the dual space (dual(V)==V^*); for product
dual(P::ProductSpace) # Return a new product space with reversed order of index spaces
dual(W::HomSpace) # Return the dual of a HomSpace which contains the dual of morphisms
Base.adjoint(V::VectorSpace) = dual(V) # make V' as the dual of V
Base.adjoint(W::HomSpace{<:EuclideanSpace}) # Return the adjoint of a HomSpace which co
representation.
isdual(V::ElementarySpace) # wether an ElementarySpace `V` is normal or rather a dual s
flip(V::ElementarySpace) # flip(V)==V̄^*
⊕ # direct sum of the elementary spaces `V1`, `V2`, ...
⊗ # representing the tensor product of several elementary vector spaces
Base.:*(V1::VectorSpace, V2::VectorSpace) = ⊗(V1, V2)
fuse(V1::S, V2::S, V3::S...) where {S<:ElementarySpace} # returns a single vector spac
fuse(P::ProductSpace{S}) where {S<:ElementarySpace}
ismonomorphic # Return whether there exist monomorphisms from `V1` to `V2`, i.e. 'injec
isepimorphic # Return whether there exist epimorphisms from `V1` to `V2`, i.e. 'surject
isisomorphic # Return if `V1` and `V2` are isomorphic, meaning that there exists isomo
const ≲ = ismonomorphic
const ≳ = isepimorphic
const ≅ = isisomorphic
<(V1::VectorSpace, V2::VectorSpace) = V1 ≲ V2 && !(V1 ≳ V2)
>(V1::VectorSpace, V2::VectorSpace) = V1 ≳ V2 && !(V1 ≲ V2)
infimum # Return the infimum of a number of elementary spaces
supremum # Return the supremum of a number of elementary spaces
Base.:(==)
Base.hash
Base.length(P::ProductSpace) # number of vector spaces
Base.iterate
Base.indexed_iterate
Base.eltype
Base.IteratorEltype
Base.IteratorSize
Base.convert
codomain(W::HomSpace) # codomain of a HomSpace.
domain(W::HomSpace) # domain of a HomSpace.
```

Constructors

```
# Cartesian and Complex Space
CartesianSpace(d::Integer = 0; dual = false) # Constructed by an integer number which
ComplexSpace(d::Integer = 0; dual = false)
CartesianSpace(dim::Pair; dual = false) # Constructed by (Trivial(),d)
ComplexSpace(dim::Pair; dual = false)
CartesianSpace(dims::AbstractDict; kwargs...) # Constructed by (Trivial() => d)
ComplexSpace(dims::AbstractDict; kwargs...)
Base.getindex(::RealNumbers) = CartesianSpace # Make ℝ[] a synonyms for CartesianSpace
Base.getindex(::ComplexNumbers) = ComplexSpace # make ℂ[] a synonyms for ComplexSpace
Base.:^(::RealNumbers, d::Int) # Return a CartesianSpace with dimension `d`.
Base.:^(::ComplexNumbers, d::Int) # Return a ComplexSpace with dimension `d`

# Graded Space
GradedSpace{I, NTuple{N, Int}}(dims; dual::Bool = false) where {I, N} # dims = (c=>dc,
GradedSpace{I, NTuple{N, Int}}(dims::Pair; dual::Bool = false) where {I, N}
GradedSpace{I, SectorDict{I, Int}}(dims; dual::Bool = false) where {I<:Sector}
GradedSpace{I, SectorDict{I, Int}}(dims::Pair; dual::Bool = false) where {I<:Sector}
GradedSpace{I,D}(; kwargs...) where {I<:Sector,D}
GradedSpace{I,D}(d1::Pair, d2::Pair, dims::Vararg{Pair}; kwargs...) where {I<:Sector,D
GradedSpace{I}(args...; kwargs...) where {I<:Sector}
GradedSpace(dims::Tuple{Vararg{Pair{I, <:Integer}}}; dual::Bool = false) where {I<:Sec
GradedSpace(dims::Vararg{Pair{I, <:Integer}}; dual::Bool = false) where {I<:Sector}
GradedSpace(dims::AbstractDict{I, <:Integer}; dual::Bool = false) where {I<:Sector}

struct SpaceTable end
const Vect = SpaceTable()
Base.getindex(::SpaceTable) = ComplexSpace # Vect[] = ComplexSpace
Base.getindex(::SpaceTable, ::Type{Trivial}) = ComplexSpace
Base.getindex(::SpaceTable, I::Type{<:Sector}) # # Vect[I]; Return `GradedSpace{I, NTu
Base.getindex(::ComplexNumbers, I::Type{<:Sector}) = Vect[I] # Make ℂ[I] = Vect[I]

struct RepTable end
const Rep = RepTable()
Base.getindex(::RepTable, G::Type{<:Group}) # Rep[G] = Vect[Irrep[G]]
const ZNSpace{N} = GradedSpace{ZNIrrep{N}, NTuple{N,Int}}
const Z2Space = ZNSpace{2}
const Z3Space = ZNSpace{3}
const Z4Space = ZNSpace{4}
const U1Space = Rep[U₁]
const CU1Space = Rep[CU₁]
const SU2Space = Rep[SU₂]
const ℤ₂Space = Z2Space
const ℤ₃Space = Z3Space
const ℤ₄Space = Z4Space
const U₁Space = U1Space
const CU₁Space = CU1Space
```

```
const SU₂Space = SU2Space

# Product Space
ProductSpace(spaces::Vararg{S, N}) where {S<:ElementarySpace, N}
ProductSpace{S, N}(spaces::Vararg{S, N}) where {S<:ElementarySpace, N}
ProductSpace{S}(spaces) where {S<:ElementarySpace}
ProductSpace(P::ProductSpace)
⊗(V1::S, V2::S) where {S<:ElementarySpace}= ProductSpace((V1, V2))
⊗(P1::ProductSpace{S}, V2::S) where {S<:ElementarySpace}
⊗(V1::S, P2::ProductSpace{S}) where {S<:ElementarySpace}
⊗(P1::ProductSpace{S}, P2::ProductSpace{S}) where {S<:ElementarySpace}
⊗(P::ProductSpace{S, 0}, ::ProductSpace{S, 0}) where {S<:ElementarySpace} = P
⊗(P::ProductSpace{S}, ::ProductSpace{S, 0}) where {S<:ElementarySpace} = P
⊗(::ProductSpace{S, 0}, P::ProductSpace{S}) where {S<:ElementarySpace} = P
⊗(V::ElementarySpace) = ProductSpace((V,))
⊗(P::ProductSpace) = P
Base.:^(V::ElementarySpace, N::Int) = ProductSpace{typeof(V), N}(ntuple(n->V, N))
Base.:^(V::ProductSpace, N::Int) = ⊗(ntuple(n->V, N)...)
Base.literal_pow(::typeof(^), V::ElementarySpace, p::Val{N}) where N =
    ProductSpace{typeof(V), N}(ntuple(n->V, p))
insertunit(P::ProductSpace, i::Int = length(P)+1; dual = false, conj = false) # For `P

# HomSpace
→(dom::TensorSpace{S}, codom::TensorSpace{S}) where {S<:ElementarySpace} =
    HomSpace(ProductSpace(codom), ProductSpace(dom))
←(codom::TensorSpace{S}, dom::TensorSpace{S}) where {S<:ElementarySpace} =
    HomSpace(ProductSpace(codom), ProductSpace(dom))
```

Others structures

```
struct TrivialOrEmptyIterator
    isempty::Bool
end # returns nothing is isempty = true, otherwise returns Trivial()
```

Details about `dual`, `conj`, `flip`

In `vectorspaces.jl`:

```
function dual end
dual(V::EuclideanSpace) = conj(V)
Base.adjoint(V::VectorSpace) = dual(V)
function flip end
```

In `generalspace.jl`:

```
dual(V::GeneralSpace{𝕜}) where {𝕜} =
    GeneralSpace{𝕜}(dim(V), !isdual(V), isconj(V))
Base.conj(V::GeneralSpace{𝕜}) where {𝕜} =
    GeneralSpace{𝕜}(dim(V), isdual(V), !isconj(V))
```

In `cartesianspace.jl`:

```
flip(V::CartesianSpace) = V
```

In `complexspace.jl`:

```
Base.conj(V::ComplexSpace) = ComplexSpace(dim(V), !isdual(V))
flip(V::ComplexSpace) = dual(V)
```

In `sectors.jl`:

`Base.conj(a::I)`: $\overline{a}$, conjugate or dual label of a.

```
dual(a::Sector) = conj(a)
```

In `trivial.jl`:

```
Base.conj(::Trivial) = Trivial()
```

In `anyons.jl`:

```
Base.conj(s::IsingAnyon) = s
Base.conj(s::FibonacciAnyon) = s
```

In `irreps.jl`:

```
Base.conj(c::ZNIrrep{N}) where {N} = ZNIrrep{N}(-c.n)
Base.conj(c::U1Irrep) = U1Irrep(-c.charge)
Base.conj(s::SU2Irrep) = s
Base.conj(c::CU1Irrep) = c
```

In `gradedspace.jl`:

In fact, `GradedSpace` is the reason `flip` exists, cause in this case it is different than `dual`. The existence of flip originates from the non-trivial isomorphism between $R_{\bar{a}}$ and R_a^*, i.e. the representation space of the dual $\bar{a}$ of sector a and the dual of the representation space of sector a. In order for `flip(V)` to be isomorphic to `V`, it is such that, if `V = GradedSpace(a=>n_a,...)` then `flip(V) = dual(GradedSpace(dual(a)=>n_a,....))`.

In the structure of `TensorXD.jl`, we only keep the simple objects. It means that we don't have objects correspond to a^* in the language of category. Therefore, `dual(a) = conj(a)` both correspond to $\bar{a}$. The dual space of a space is denoted in the field named as `dual` in the type definitions. If `dual = true`, it means that we represent the space R_a^* which is isomorphic to $R_{\bar{a}}$, and in the methods like `sectors` and `dim` we get the sectors and corresponding dims in the corresponding $R_{\bar{a}}$.

```
sectors(V::GradedSpace{I,<:AbstractDict}) where {I<:Sector} =
    SectorSet{I}(s->isdual(V) ? dual(s) : s, keys(V.dims))
sectors(V::GradedSpace{I,NTuple{N,Int}}) where {I<:Sector, N} =
    SectorSet{I}(Iterators.filter(n->V.dims[n]!=0, 1:N)) do n
        isdual(V) ? dual(values(I)[n]) : values(I)[n]
dim(V::GradedSpace{I,<:AbstractDict}, c::I) where {I<:Sector} =
    get(V.dims, isdual(V) ? dual(c) : c, 0)
dim(V::GradedSpace{I,<:Tuple}, c::I) where {I<:Sector} =
    V.dims[findindex(values(I), isdual(V) ? dual(c) : c)]
Base.conj(V::GradedSpace) = typeof(V)(V.dims, !V.dual)
function flip(V::GradedSpace{I}) where {I<:Sector}
    if isdual(V)
        typeof(V)(c=>dim(V, c) for c in sectors(V))
    else
        typeof(V)(dual(c)=>dim(V, c) for c in sectors(V))'
    end
end
```

In `productspace.jl`:

The order of the spaces are reversed before taking the dual of each elementray vecor space:

```
dual(P::ProductSpace{<:ElementarySpace, 0}) = P
dual(P::ProductSpace) = ProductSpace(map(dual, reverse(P.spaces)))
```

In `homespace.jl`:

For a morphism the dual of the morphism is different with the adjoint of it. In the tensor category language, the dual of a morphism is called the transpose of the morphism, while the adjoint of a morphism is called the dagger of the morphism.

```
dual(W::HomSpace) = HomSpace(dual(W.domain), dual(W.codomain))
Base.adjoint(W::HomSpace{<:EuclideanSpace}) = HomSpace(W.domain, W.codomain)
```

The sequence of the elementary spaces in a TensorMapSpace is defined as $1 : N_1$ for codomain vectors, and $N_1 + 1 : N_1 + N_2$ for domain dual vectors. Note that the sequence of the domain vectors are not reversed, and the dual is taken individually for each elementary space.

```
Base.getindex(W::TensorMapSpace{<:IndexSpace, N₁, N₂}, i) where {N₁, N₂} =
    i <= N₁ ? codomain(W)[i] : dual(domain(W)[i-N₁])
```

VectorSpace type

From the Introduction, it should be clear that an important aspect in the definition of a tensor (map) is specifying the vector spaces and their structure in the domain and codomain of the map. The starting point is an abstract type `VectorSpace`

```
abstract type VectorSpace end
```

which is actually a too restricted name. All instances of subtypes of `VectorSpace` will represent objects in $\mathbb{k}$-linear monoidal categories, but this can go beyond normal vector spaces (i.e. objects in the category $\mathbf{Vect}$) and even beyond objects of $\mathbf{SVect}$. However, in order not to make the remaining discussion too abstract or complicated, we will simply refer to subtypes of `VectorSpace` instead of specific categories, and to spaces (i.e. `VectorSpace` instances) instead of objects from these categories. In particular, we define two abstract subtypes

```
abstract type ElementarySpace{𝕜} <: VectorSpace end
const IndexSpace = ElementarySpace

abstract type CompositeSpace{S<:ElementarySpace} <: VectorSpace end
```

Here, `ElementarySpace` is a super type for all vector spaces (objects) that can be associated with the individual indices of a tensor, as hinted to by its alias `IndexSpace`. It is parametrically dependent on $\mathbb{k}$, the field of scalars (see the next section on Fields).

On the other hand, subtypes of `CompositeSpace{S}` where `S<:ElementarySpace` are composed of a number of elementary spaces of type `S`. So far, there is a single concrete type `ProductSpace{S,N}` that represents the homogeneous tensor product of `N` vector spaces of type `S`. Its properties are discussed in the section on Composite spaces, together with possible extensions for the future.

Throughout TensorXD.jl, the function `spacetype` returns the type of `ElementarySpace` associated with e.g. a composite space or a tensor. It works both on instances and in the type domain. Its use will be illustrated below.

Fields

Vector spaces (linear categories) are defined over a field of scalars $\mathbb{k}$. We define a type hierarchy to specify the scalar field, but so far only support real and complex numbers, via

```
abstract type Field end

struct RealNumbers <: Field end
struct ComplexNumbers <: Field end

const ℝ = RealNumbers()
const ℂ = ComplexNumbers()
```

Note that $\mathbb{R}$ and $\mathbb{C}$ can be typed as `\bbR`+TAB and `\bbC`+TAB. One reason for defining this new type hierarchy instead of recycling the types from Julia's `Number` hierarchy is to introduce some syntactic sugar without committing type piracy. In particular, we now have

```
julia> 3 ∈ ℝ
true

julia> 5.0 ∈ ℂ
true

julia> 5.0+1.0*im ∈ ℝ
false

julia> Float64 ⊆ ℝ
true

julia> ComplexF64 ⊆ ℂ
true

julia> ℝ ⊆ ℂ
true

julia> ℂ ⊆ ℝ
false
```

and furthermore —probably more usefully— `ℝ^n` and `ℂ^n` create specific elementary vector spaces as described in the next section. The underlying field of a vector space or tensor `a` can be obtained with `field(a)`.

Elementary spaces

As mentioned at the beginning of this section, vector spaces that are associated with the individual indices of a tensor should be implemented as subtypes of `ElementarySpace`. As the domain and codomain of a tensor map will be the tensor product of such objects which all have the same type, it is important that related vector spaces, e.g. the dual space, are objects of the same concrete type (i.e. with the same type parameters in case of a parametric type). In particular, every `ElementarySpace` should implement the following methods

- `dim(::ElementarySpace) -> ::Int` returns the dimension of the space as an `Int`
- `dual(::S) where {S<:ElementarySpace} -> ::S` returns the dual space `dual(V)`, using an instance of the same concrete type (i.e. not via type parameters); this should satisfy `dual(dual(V))==V`
- `conj(::S) where {S<:ElementarySpace} -> ::S` returns the complex conjugate space `conj(V)`, using an instance of the same concrete type (i.e. not via type parameters); this should satisfy `conj(conj(V))==V` and we automatically have `conj(V::ElementarySpace{ℝ}) = V`.

For convenience, the dual of a space `V` can also be obtained as `V'`.

There is concrete type `GeneralSpace` which is completely characterized by its field `𝕜`, its dimension and whether its the dual and/or complex conjugate of $\mathbb{k}^d$.

```
struct GeneralSpace{𝕜} <: ElementarySpace{𝕜}
    d::Int
    dual::Bool
    conj::Bool
end
```

We furthermore define the abstract type

```
abstract type InnerProductSpace{𝕜} <: ElementarySpace{𝕜} end
```

to contain all vector spaces `V` which have an inner product and thus a canonical mapping from `dual(V)` to `conj(V)`. This mapping is provided by the metric, but no further support for working with metrics is currently implemented.

Finally there is

```
abstract type EuclideanSpace{𝕜} <: InnerProductSpace{𝕜} end
```

to contain all spaces `V` with a standard Euclidean inner product (i.e. where the metric is the identity). These spaces have the natural isomorphisms `dual(V) == conj(V)`. In the language of the previous section on categories, this subtype represents dagger or unitary categories, and support an `adjoint` operation. In particular, we have two concrete types

```
struct CartesianSpace <: EuclideanSpace{ℝ}
    d::Int
end
struct ComplexSpace <: EuclideanSpace{ℂ}
  d::Int
  dual::Bool
end
```

to represent the Euclidean spaces $\mathbb{R}^d$ or $\mathbb{C}^d$ without further inner structure. They can be created using the syntax `CartesianSpace(d) == ℝ^d == ℝ[](d)` and `ComplexSpace(d) == ℂ^d == ℂ[](d)`, or `ComplexSpace(d, true) == ComplexSpace(d; dual = true) == (ℂ^d)' == ℂ[](d)'` for the dual space of $\mathbb{C}^d$. Note that the brackets are required because of the precedence rules, since `d' == d` for `d::Integer`.

Some examples:

```
julia> dim(ℝ^10)
10

julia> (ℝ^10)' == ℝ^10 == ℝ[](10)
true

julia> isdual((ℂ^5))
false

julia> isdual((ℂ^5)')
true

julia> isdual((ℝ^5)')
false

julia> dual(ℂ^5) == (ℂ^5)' == conj(ℂ^5) == ComplexSpace(5; dual = true)
true

julia> typeof(ℝ^3)
CartesianSpace
```

```
julia> spacetype(ℝ^3)
CartesianSpace

julia> spacetype(ℝ[])
CartesianSpace
```

Note that `ℝ[]` and `ℂ[]` are synonyms for `CartesianSpace` and `ComplexSpace` respectively. This is not very useful in itself, and is motivated by its generalization to `GradedSpace`. We refer to the subsection on graded spaces on the next page for further information about `GradedSpace`, which is another subtype of `EuclideanSpace{ℂ}` with an inner structure corresponding to the irreducible representations of a group, or more generally, the simple objects of a fusion category.

> **Note**
>
> For `ℂ^n` the dual space is equal (or naturally isomorphic) to the conjugate space, but not to the space itself. This means that even for `ℂ^n`, arrows matter in the diagrammatic notation for categories or for tensors, and in particular that a contraction between two tensor indices will check that one is living in the space and the other in the dual space. This is in contrast with several other software packages, especially in the context of tensor networks, where arrows are only introduced when discussing symmetries. We believe that our more purist approach can be useful to detect errors (e.g. unintended contractions). Only with `ℝ^n` will their be no distinction between a space and its dual. When creating tensors with indices in `ℝ^n` that have complex data, a one-time warning will be printed, but most operations should continue to work nonetheless.

Composite spaces

Composite spaces are vector spaces that are built up out of individual elementary vector spaces of the same type. The most prominent (and currently only) example is a tensor product of `N` elementary spaces of the same type `S`, which is implemented as

```
struct ProductSpace{S<:ElementarySpace, N} <: CompositeSpace{S}
    spaces::NTuple{N, S}
end
```

Given some `V1::S`, `V2::S`, `V3::S` of the same type `S<:ElementarySpace`, we can easily construct `ProductSpace{S,3}((V1,V2,V3))` as `ProductSpace(V1,V2,V3)` or using `V1 ⊗ V2 ⊗ V3`, where `⊗` is simply obtained by typing `\otimes`+TAB. In fact, for convenience, also the regular multiplication operator `*` acts as tensor product between vector spaces, and as a consequence so does raising a vector space to a positive integer power, i.e.

```
julia> V1 = ℂ^2
ℂ^2

julia> V2 = ℂ^3
ℂ^3

julia> V1 ⊗ V2 ⊗ V1' == V1 * V2 * V1' == ProductSpace(V1,V2,V1') == ProductSpace(V1,V2
true

julia> V1^3
(ℂ^2 ⊗ ℂ^2 ⊗ ℂ^2)

julia> dim(V1 ⊗ V2)
6

julia> dims(V1 ⊗ V2)
(2, 3)

julia> dual(V1 ⊗ V2)
((ℂ^3)' ⊗ (ℂ^2)')

julia> spacetype(V1 ⊗ V2)
ComplexSpace

julia> spacetype(ProductSpace{ComplexSpace,3})
ComplexSpace
```

Here, the new function `dims` maps `dim` to the individual spaces in a `ProductSpace` and returns the result as a tuple. Note that the rationale for $dual(V1 \otimes V2)$ was explained in the subsection on duality in the introduction to category theory.

Following Julia's Base library, the function `one` applied to a `ProductSpace{S,N}` returns the multiplicative identity, which is `ProductSpace{S,0}(())`. The same result is obtained when acting on an instance `V` of `S::ElementarySpace` directly, however note that `V ⊗ one(V)` will yield a `ProductSpace{S,1}(V)` and not `V` itself. The same result can be obtained with `⊗(V)`. Similar to Julia Base, `one` also works in the type domain.

In the future, other `CompositeSpace` types could be added. For example, the wave function of an `N`-particle quantum system in first quantization would require the introduction of a `SymmetricSpace{S,N}` or a `AntiSymmetricSpace{S,N}` for bosons or fermions respectively, which correspond to the symmetric (permutation invariant) or antisymmetric subspace of `V^N`, where `V::S` represents the Hilbert space of the single particle system. Other domains, like general relativity, might also benefit from tensors living in a subspace with certain symmetries under specific index permutations.

Space of morphisms

Given that we define tensor maps as morphisms in a $\mathbb{k}$-linear monoidal category, i.e. linear maps, we also define a type to denote the corresponding space. Indeed, in a $\mathbb{k}$-linear category C, the set of morphisms $\mathrm{Hom}(W, V)$ for $V, W \in C$ is always an actual vector space, irrespective of whether or not C is a subcategory of $\mathbf{(S)Vect}$.

We introduce the type

```
struct HomSpace{S<:ElementarySpace, P1<:CompositeSpace{S}, P2<:CompositeSpace{S}}
    codomain::P1
    domain::P2
end
```

and can create it as either `domain → codomain` or `codomain ← domain` (where the arrows are obtained as `\to+TAB` or `\leftarrow+TAB`, and as `\rightarrow+TAB` respectively). The reason for first listing the codomain and than the domain will become clear in the section on tensor maps.

Note that `HomSpace` is not a subtype of `VectorSpace`, i.e. we restrict the latter to denote certain categories and their objects, and keep `HomSpace` distinct. However, `HomSpace` has a number of properties defined, which we illustrate via examples

```
julia> W = ℂ^2 ⊗ ℂ^3 → ℂ^3 ⊗ dual(ℂ^4)
(ℂ^3 ⊗ (ℂ^4)') ← (ℂ^2 ⊗ ℂ^3)

julia> field(W)
ℂ

julia> dual(W)
((ℂ^3)' ⊗ (ℂ^2)') ← (ℂ^4 ⊗ (ℂ^3)')

julia> adjoint(W)
(ℂ^2 ⊗ ℂ^3) ← (ℂ^3 ⊗ (ℂ^4)')

julia> spacetype(W)
ComplexSpace

julia> spacetype(typeof(W))
ComplexSpace

julia> W[1]
ℂ^3

julia> W[2]
```

```
(ℂ^4)'

julia> W[3]
(ℂ^2)'

julia> W[4]
(ℂ^3)'

julia> dim(W)
72
```

Note that indexing `W` yields first the spaces in the codomain, followed by the dual of the spaces in the domain. This particular convention is useful in combination with the instances of type `TensorMap`, which represent morphisms living in such a `HomSpace`. Also note that `dim(W)` here seems to be the product of the dimensions of the individual spaces, but that this is no longer true once symmetries are involved. At any time will `dim(::HomSpace)` represent the number of linearly independent morphisms in this space.

Partial order among vector spaces

Vector spaces of the same `spacetype` can be given a partial order, based on whether there exist injective morphisms (a.k.a *monomorphisms*) or surjective morphisms (a.k.a. *epimorphisms*) between them. In particular, we define `ismonomorphic(V1, V2)`, with Unicode synonym `V1 ≾ V2` (obtained as `\precsim+TAB`), to express whether there exist monomorphisms in `V1 → V2`. Similarly, we define `isepimorphic(V1, V2)`, with Unicode synonym `V1 ≿ V2` (obtained as `\succsim+TAB`), to express whether there exist epimorphisms in `V1 → V2`. Finally, we define `isisomorphic(V1, V2)`, with Unicode alternative `V1 ≅ V2` (obtained as `\cong+TAB`), to express whether there exist isomorphism in `V1 → V2`. In particular `V1 ≅ V2` if and only if `V1 ≾ V2 && V1 ≿ V2`.

For completeness, we also export the strict comparison operators `<` and `>` (`\prec+TAB` and `\succ+TAB`), with definitions

```
<(V1::VectorSpace, V2::VectorSpace) = V1 ≾ V2 && !(V1 ≿ V2)
>(V1::VectorSpace, V2::VectorSpace) = V1 ≿ V2 && !(V1 ≾ V2)
```

However, as we expect these to be less commonly used, no ASCII alternative is provided.

In the context of `spacetype(V) <: EuclideanSpace`, `V1 ≾ V2` implies that there exists isometries $W : V1 \to V2$ such that $W^\dagger \circ W = \mathrm{id}_{V1}$, while `V1 ≅ V2` implies that there exist unitaries $U : V1 \to V2$ such that $U^\dagger \circ U = \mathrm{id}_{V1}$ and $U \circ U^\dagger = \mathrm{id}_{V2}$.

Note that spaces that are isomorphic are not necessarily equal. One can be a dual space, and the other a normal space, or one can be an instance of `ProductSpace`, while the other is an `ElementarySpace`. There

will exist (infinitely) many isomorphisms between the corresponding spaces, but in general none of those will be canonical.

There are also a number of convenience functions to create isomorphic spaces. The function `fuse(V1, V2, ...)` or `fuse(V1 ⊗ V2 ⊗ ...)` returns an elementary space that is isomorphic to `V1 ⊗ V2 ⊗ ...`. The function `flip(V::ElementarySpace)` returns a space that is isomorphic to `V` but has `isdual(flip(V)) == isdual(V')`, i.e. if `V` is a normal space than `flip(V)` is a dual space. `flip(V)` is different from `dual(V)` in the case of `GradedSpace`. It is useful to flip a tensor index from a ket to a bra (or vice versa), by contracting that index with a unitary map from `V1` to `flip(V1)`. (**In the language of category, the we have `flip(a)==`** $\overline{a}^*$.) We refer to Index operations for further information. Some examples:

```
julia> ℝ^3 ≾ ℝ^5
true

julia> ℂ^3 ≾ (ℂ^5)'
true

julia> (ℂ^5) ≅ (ℂ^5)'
true

julia> fuse(ℝ^5, ℝ^3)
ℝ^15

julia> fuse(ℂ^3, (ℂ^5)' ⊗ ℂ^2)
ℂ^30

julia> fuse(ℂ^3, (ℂ^5)') ⊗ ℂ^2 ≅ fuse(ℂ^3, (ℂ^5)', ℂ^2) ≅ ℂ^3 ⊗ (ℂ^5)' ⊗ ℂ^2
true

julia> flip(ℂ^4)
(ℂ^4)'

julia> flip(ℂ^4) ≅ ℂ^4
true

julia> flip(ℂ^4) == ℂ^4
false
```

We also define the direct sum `V1` and `V2` as `V1 ⊕ V2`, where `⊕` is obtained by typing `\oplus`+TAB. This is possible only if `isdual(V1) == isdual(V2)`. With a little pun on Julia Base, `oneunit` applied to an elementary space (in the value or type domain) returns the one-dimensional space, which is isomorphic to the scalar field of the space itself. Some examples illustrate this better

```
julia> ℝ^5 ⊕ ℝ^3
ℝ^8

julia> ℂ^5 ⊕ ℂ^3
ℂ^8

julia> ℂ^5 ⊕ (ℂ^3)'
ERROR: SpaceMismatch(Direct sum of a vector space and its dual does not exist)

julia> oneunit(ℝ^3)
ℝ^1

julia> ℂ^5 ⊕ oneunit(ComplexSpace)
ℂ^6

julia> oneunit((ℂ^3)')
ℂ^1

julia> (ℂ^5) ⊕ oneunit((ℂ^5))
ℂ^6

julia> (ℂ^5)' ⊕ oneunit((ℂ^5)')
ERROR: SpaceMismatch(Direct sum of a vector space and its dual does not exist)
```

Finally, while spaces have a partial order, there is no unique infimum or supremum of a two or more spaces. However, if `V1` and `V2` are two `ElementarySpace` instances with `isdual(V1) == isdual(V2)`, then we can define a unique infimum `V::ElementarySpace` with the same value of `isdual` that satisfies `V ≾ V1` and `V ≾ V2`, as well as a unique supremum `W::ElementarySpace` with the same value of `isdual` that satisfies `W ≿ V1` and `W ≿ V2`. For `CartesianSpace` and `ComplexSpace`, this simply amounts to the space with minimal or maximal dimension, i.e.

```
julia> infimum(ℝ^5, ℝ^3)
ℝ^3

julia> supremum(ℂ^5, ℂ^3)
ℂ^5

julia> supremum(ℂ^5, (ℂ^3)')
ERROR: SpaceMismatch(Supremum of space and dual space does not exist)
```

The names `infimum` and `supremum` are especially suited in the case of `GradedSpace`, as the infimum of two spaces might be different from either of those two spaces, and similar for the supremum.

« Optional introduction to category theory Sectors, representation spaces and fusion trees »