TensorOperationsCore.jl

Core functionality and interface for TensorOperations.

This package sets the minimal interface required for user-defined types to work with TensorOperations.jl, or for implementing custom backends.

Implementing TensorOperations.jl for my custom type

The interface for TensorOperations.jl is composed of the following methods, which are divided into 2 categories.

Firstly, there should be support for creating new tensors based off the structure of indices of already existing tensors. Additionaly, custom (de-)allocation styles can also be provided. This is done through the use of:

1. `tensoralloc(::Backend, TC, pC, A, conjA)` or `tensoralloc(::Backend, TC, pC, A, iA, conjA, B, iB, conjB)`

This function allocates memory for a tensor with indices `pC` and scalartype `TC` based on the indices of `opA(A)`, or based on indices `iA` of `opA(A)` and `iB` of `opB(B)`. The operation `opA` (`opB`) acts as `conj` if `conjA` (`conjB`) equals `:C` or as the identity if `conjA` (`conjB`) equals `:N`.

2. `tensorfree(::Backend, C)`

This function releases the allocated memory of `C`. Note that usually it is sufficient to implement `tensoralloc` to create new objects, which will then be cleaned up by Julia's garbage collector, in which case `tensorfree` should not actually do anything.

Secondly, there are 4 operations on tensors that should be defined, as well as the support for `scalartype` which is re-exported from VectorInterface:

1. `tensoradd!(::Backend, C, A, pA, conjA, α, β)`

This function implements `C = β * C + α * permutedims(opA(A), pA)` without creating the intermediate temporary. The operation `opA` acts as `conj` if `conjA` equals `:C` or as the identity if `conjA` equals `:N`.

2. `tensorcontract!(::Backend, C, pC, A, pA, conjA, B, pB, conjB, α, β)`

This function implements `C = β * C + α * permutedims(contract(opA(A), opB(B)), pC)` without creating the intermediate temporary, where `A` and `B` are contracted such that the indices `pA[2]` of `A` are contracted with indices `pB[1]` of `B`. The remaining indices `(pA[1]..., pB[2]...)` are then permuted according to `pC`. The operation `opA` (`opB`) acts as `conj` if `conjA` (`conjB`) equals `:C` or as the identity if `conjA` (`conjB`) equals `:N`.

3. `tensortrace!(::Backend, C, pC, A, pA, conjA, α, β)`

This function implements `C = β * C + α * permutedims(partialtrace(opA(A)), pC)` without creating the intermediate temporary, where `A` is partially traced, such that indices in `pA[1]` are contracted with indices in `pA[2]`, and the remaining indices are permuted according to `pC`. The operation `opA` acts as `conj` if `conjA` equals `:C` or as the identity if `conjA` equals `:N`.

4. `tensorscalar(C)`

This function returns the single element of a tensor-like object with zero indices or dimensions.

Imlementing a custom backend for supported types

Similarly, the implementation of a custom backend, either for generic types or for specific types, can be done through the implementation of the same 4 functions,