Implementation

** Warning: this section still needs to be updated for version 2.0 **

Index notation and the `@tensor` macro

We start by describing the implementation of the `@tensor` and `@tensoropt` macro. The macros end up transforming the index expression to the corresponding function calls to the primitive building blocks, which are discussed in the next section. In principle, anyone interested in making the `@tensor` macro work for custom array types should only reimplement these building blocks, but it is useful to understand how the tensor expressions are processed.

Tensors, a.k.a indexed objects

The central objects in tensor expressions that follow the `@tensor` macro are the index expressions of the form `A[a,b,c]`. These are detected as subexpressions of the form `Expr(:ref, args)`. In fact, what is recognized by `@tensor` as an indexed object is more general, and also includes expressions of the form `A[a b c]` or `A[a b; c d e]` (`Expr(:typed_hcat, args)` and `Expr(:typed_vcat, args)`). The last form in particular is useful in more general contexts, and allows to distinguish between two sets of indices, referred to as left (`a` and `b`) and right (`c`, `d` and `e`) indices. This can be used when the `@tensor` macro wants to be generalized to user types, which for example distinguish between contravariant (upper) and covariant (lower) indices. For `AbstractArray` subtypes, such distinction is of course meaningless.

Note that the object being indexed, i.e. `A` in all of the above examples or `args[1]` in the corresponding `Expr` objects, can itself be an expression, that is not further analyzed. In particular, this may itself contain further indexing objects, such that one can get tensor objects from a list, e.g. `list[3][a,b,c]` or slice an array before using it in the `@tensor` expression, e.g. `A[1:2:end,:,3:end][a,b,c]`.

Everything appearing in the `[ ]`, e.g. `args[2:end]` (in case of `:ref` or `:typed_hcat`, the argument structure of `:typed_vcat` is slightly more complicated) is considered to be a valid index. This can be any valid Julia variable name, which is just kept as symbol, or any literal integer constant, (for legacy reasons, any literal character constant,) or finally an `Expr(Symbol("'"),args)`, i.e. an expression of the form `:(a')`. The latter is converted to a symbol of the form `Symbol("a'")` when `a` is itself a symbol or integer, or this is applied recursively if `a` contains more primes.

The implementation for detecting tensors and indices (`istensor`, `isindex`) and actually converting them to a useful format (`maketensor`, `makeindex`) are found in `src/indexnotation/tensorexpressions.jl`. In particular, `maketensor` will return the indexed object, which is just `esc(args[1])`, the list of left indices and the list of right indices.

Furthermore, there is `isscalar` and `makescalar` to detect and process subexpressions that will evaluate to a scalar. Finally, there is `isgeneraltensor` and `makegeneraltensor` to detect and process a a tensor (indexed object), that is possibly conjugated or multiplied with a scalar. This is useful because the primitive tensor operations (i.e. see Building blocks below), accept a scalar factor and conjugation flag, so that these operations can be done simultaneously and do not need to be evaluated at first (which would require additional temporaries). The function `makegeneraltensor` in particular will return the indexed object, the list of left indices, the list of right indices, the scalar factor, and a flag (`Bool`) that indicates whether the object needs to be conjugated (`true`) or not (`false`).

The file `src/indexnotation/tensorexpressions.jl` also contains simple methods to detect assignment (`isassignment`) into existing objects (i.e. `=`, `+=` and `-=`) or so-called definitions (`isdefinition`), that create a new object (via `:=` or its Unicode variant `≔`, obtained as `\coloneq + TAB`). The function `getlhsrhs` will return the left hand side and right hand side of an assignment or definition expression separately.

Finally, there are methods to detect whether the right hand side is a valid tensor expression (`istensorexpr`) and to get the indices of a complete tensor expressions. In particular, `getindices` returns a list of indices that will remain after the expression is evaluated (i.e. any index that is not contracted in the expression because it only appears once), whereas `getallindices` returns a list of all indices that appear in the expression. The latter is used to analyze complete tensor contraction graphs.

The macros `@tensor` and `@tensoropt`

Actual processing of the complete expression that follows the `@tensor` macro and converting it into a list of actual calls to the primitive tensor operations is handled by the functions defined in `src/indexnotation/tensormacro.jl`. The integral expression received by the `@tensor` macro is passed on to the `tensorify` function. The `@tensoropt` macro will first generate the data required to optimize contraction order, by calling `optdata`. If no actual costs are specified, i.e. `@tensoropt` receives a single expression, then `optdata` just assigns the same cost to all indices in the expression. Otherwise, the expression that specifies the costs need to be parsed first (`parsecost`). Finally, `@tensoropt` also calls `tensorify` passing the optdata as a second optional argument (whose default value `nothing` is used by `@tensor`).

The function `tensorify` consists of several steps. Firstly, it canonicalizes the expression. Currently, this involves a single pass which expands all `conj` calls of e.g. products or sums to a `conj` call on each of the arguments (via the `expandconj` function). Secondly, `tensorify` will process the contraction order using `processcontractorder`. This starts from the fact that a product of several tensors, specified as `A[...]*B[...]*C[...]` yields a single `Expr(:call,[:*,...])` object where all the factors are still together. If a user wants a specific order, he can do so by grouping them with parenthesis. Whenever an expression `Expr(:call,[:*,...])` is found where more than two of the arguments satisfy `isgeneraltensor`, `processcontractorder` will convert it into a nested set of pairwise multiplications according to a number of strategies discussed in the next subsection.

The major part of `tensorify` is to generate the correct function calls corresponding to the tensor expression. It detects assignments or definitions (the most common case) and validates the left hand side thereof (i.e. it should satisfy `istensor` and have no duplicate indices). Then, it generates the corresponding function calls corresponding to the index expression by passing onto the `deindexify` function, which takes the signature `julia deindexify(dst, β, ex, α, leftind, rightind, istemporary = false)` Here, `dst` is the symbol or expression corresponding to the destination object; it's `nothing` in case of a definition (`:=`), i.e. if the object corresponding to the result needs to be created/allocated. `β` and `α` are `isscalar` expressions, and `deindexify` will create the function calls required to update `dst` with multiplying `dst` with `β` and adding `α` times the result of the expression `ex` to it; this is supported as a one step process by each of the primitive operations. `leftind` and `rightind` correspond to the list of indices of the left hand side of the defition or assignment. The final argument `istemporary` indicates, if `dst == nothing` and a new object needs to be created/allocated, whether it is a temporary object. If `istemporary == true`, it can be stored in the cache and later retrieved. If `istemporary == false`, it corresponds to an explicit left hand side created by the user in a definition, and should not be in the cache.

The function `deindexify` will determine the top level operation represented by `ex` (which should be a `istensorexpr`), and then pass on to `deindexify_generaltensor`, `deindexify_linearcombination`, `deindexify_contraction` for actually creating the correct function call expressions. If any of the arguments of e.g. a linear combination or a tensor contraction is itself a composite tensor expression (i.e. not a `isgeneraltensor`), `deindexify` is called recursively.

Analyzing contraction graphs (a.k.a tensor networks) and optimizing contraction order

The function `processcontractorder`, which is excuted before the index expression is converted to function calls, will detect any multiplication with more than two `isgeneraltensor` factors, and divide it up into a nested sequence of pairwise multiplications (tensor contractions), i.e. a tree. If the `@tensor` macro was used, `optdata = nothing` and in principle the multiplication will be performed from left to right. There is one exception, which is that if the indices follow the NCON convention, i.e. negative integers are used for uncontracted indices and positive integers for contracted indices. Then the contraction tree is built such that tensors that share the contraction index which is the lowest positive integer are contracted first. Relevant code can be found in `src/indexnotation/ncontree.jl`

When the `@tensoropt` macro was used, `optdata` is a dictionary associating a cost (either a number or a polynomial in some abstract scaling parameter) to every index, and this information is used to determine the (asymptotically) optimal contraction tree (in terms of number of floating point operations). The code for the latter is in `src/indexnotation/optimaltree.jl`, with the lightweight polynomial implementation in `src/indexnotation/polynomial.jl`. Aside from a generic polynomial type `Poly`, the latter also contains a `Power` type which represents a single term of a polynomial (i.e. a scalar coefficient and an exponent). This type is closed under multiplication, and can be multiplied much more efficiently. Only under addition is a generic `Poly` returned.

Building blocks

The `@tensor` macro converts the index expression into a set of function calls corresponding to three primitive operations: addition, tracing and contraction. These operations are implemented for arbitrary strided arrays from Julia Base, i.e. `Arrays`, views with ranges thereof, and certain reshape operations. This includes certain arrays that can only be determined to be strided on runtime, and does therefore not coincide with the type union `StridedArray` from Julia Base. In fact, the methods accept `AbstractArray` objects, but convert these to `(Unsafe)StridedView` objects from the package Strided.jl, and we refer to this package for a more detailed discussion on which arrays are supported and why.

The primitive tensor operations are captured by the following mutating methods (note that these are not exported)

`TensorOperations.add!` — Function

```
add!(α, A, conjA, β, C, indleft, indright)
```

Implements `C = β*C+α*permute(op(A))` where `A` is permuted such that the left (right) indices of `C` correspond to the indices `indleft` (`indright`) of `A`, and `op` is `conj` if `conjA == :C` or the identity map if `conjA == :N` (default). Together, `(indleft..., indright...)` is a permutation of 1 to the number of indices (dimensions) of `A`.

`TensorOperations.trace!` — Function

```
trace!(α, A, conjA, β, C, indleft, indright, cind1, cind2)
```

Implements `C = β*C+α*partialtrace(op(A))` where `A` is permuted and partially traced, such that the left (right) indices of `C` correspond to the indices `indleft` (`indright`) of `A`, and indices `cindA1` are contracted with indices `cindA2`. Furthermore, `op` is `conj` if `conjA == :C` or the identity map if `conjA=:N` (default). Together, `(indleft..., indright..., cind1, cind2)` is a permutation of 1 to the number of indices (dimensions) of `A`.

`TensorOperations.contract!` — Function

```
contract!(α, A, conjA, B, conjB, β, C, oindA, cindA, oindB, cindB, indleft, indri
```

Implements `C = β*C+α*contract(opA(A),opB(B))` where `A` and `B` are contracted, such that the indices `cindA` of `A` are contracted with indices `cindB` of `B`. The open indices `oindA` of `A` and `oindB` of `B` are permuted such that `C` has left (right) indices corresponding to indices `indleft` (`indright`) out of `(oindA..., oindB...)`. The operation `opA` (`opB`) acts as `conj` if `conjA` (`conjB`) equal `:C` or as the identity map if `conjA` (`conjB`) equal `:N`. Together, `(oindA..., cindA...)` is a permutation of 1 to the number of indices of `A` and `(oindB..., cindB...)` is a permutation of 1 to the number of indices of `C`. Furthermore, `length(cindA) == length(cindB)`, `length(oindA)+length(oindB)` equals the number of indices of `C` and `(indleft..., indright...)` is a permutation of `1` ot the number of indices of `C`.

The final argument `syms` is optional and can be either `nothing`, or a tuple of three symbols, which are used to identify temporary objects in the cache to be used for permuting `A`, `B` and `C` so as to perform the contraction as a matrix multiplication.

These are the central objects that should be overloaded by custom tensor types that would like to be used within the `@tensor` environment. They are also used by the function based methods discussed in the section Functions.

Furthermore, it is essential to be able to construct new tensor objects that are similar to existing ones, i.e. to place the result of the computation in case no output is specified. In order to reuse temporary objects stored in the global cache, this method also receives a candidate similar object, which it can return if it matches the requirements.

Missing docstring.

Missing docstring for `TensorOperations.checked_similar_from_indices`. Check Documenter's build log for details.

Note that the type of the cached object is not known to the compiler, as the cache stores objects as `Any`. Therefore, the function `checked_similar_from_indices` should try to restore the type information. By passing any object retrieved from the cache through this function, type stability within the `@tensor` macro can then still be guaranteed.

Finally, there is a particularly simple method `scalar` whose sole purpose is to extract the single entry of an object with zero indices, i.e. an instance of `AbstractArray{T,0}` in case of Julia Base arrays:

`TensorOperations.scalar` — Function

```
scalar(C)
```

Returns the single element of a tensor-like object with zero indices or dimensions.

The implementation of all of these methods can be found in `src/implementation/stridedarray.jl`.

By implementing these five methods for other types that represent some kind of tensor or multidimensional object, they can be used in combination with the `@tensor` macro. In particular, we also provide basic support for contracting a `Diagonal` matrix with an arbitrary strided array in `src/implementation/diagonal.jl`.

« Cache for temporaries