```julia
function cached_permute(sym::Symbol, t::TensorMap{S},
                            p1::IndexTuple{N₁},  p2::IndexTuple{N₂}=();
                            copy::Bool = false) where {S, N₁, N₂}
    cod = ProductSpace{S, N₁}(map(n->space(t, n), p1))
    dom = ProductSpace{S, N₂}(map(n->dual(space(t, n)), p2))
    # share data if possible
    if !copy
        if p1 === codomainind(t) && p2 === domainind(t)
            return t
        elseif has_shared_permute(t, p1, p2)
            return TensorMap(reshape(t.data, dim(cod), dim(dom)), cod, dom)
        end
    end
    # general case
    @inbounds begin
        tp = TO.cached_similar_from_indices(sym, eltype(t), p1, p2, t, :N)
        return add!(true, t, false, tp, p1, p2)
    end
end

function cached_permute(sym::Symbol, t::AdjointTensorMap{S},
                            p1::IndexTuple,  p2::IndexTuple=();
                            copy::Bool = false) where {S, N₁, N₂}
    p1′ = adjointtensorindices(t, p2)
    p2′ = adjointtensorindices(t, p1)
    adjoint(cached_permute(sym, adjoint(t), p1′, p2′; copy = copy))
end

scalar(t::AbstractTensorMap{S}) where {S<:IndexSpace} =
    dim(codomain(t)) == dim(domain(t)) == 1 ?
        first(blocks(t))[2][1, 1] : throw(SpaceMismatch())



"""
    _add_trivial_kernel!(α, tsrc::AbstractTensorMap, β, tdst::AbstractTensorMap,
                            p1::IndexTuple, p2::IndexTuple, fusiontreemap)

`tsrc` and `tdst` are TrivialTensorMap, thus `tsrc[]` and `tdst[]` gives the data of
the tensor map in a form of multidimensional array. `pdata = (p1..., p2...)` gives
the
permutation of the indices of the tensor map `tsrc`. After permutation `tsrc`
changes
`to `tsrc2` and has the same space with `tdst`.

After running, `tdst` will be replaced in-place by the tensor map `tsrc2*α +
tdst*β`.
"""
function _add_trivial_kernel!(α, tsrc::AbstractTensorMap, β,
tdst::AbstractTensorMap,
                                p1::IndexTuple, p2::IndexTuple, fusiontreemap)
    #cod = codomain(tsrc)
    #dom = domain(tsrc)
    #n = length(cod)
```

```julia
    pdata = (p1..., p2...)
    axpby!(α, permutedims(tsrc[], pdata), β, tdst[])
    return nothing
end

"""
    _addabelianblock!(α, tsrc::AbstractTensorMap, β, tdst::AbstractTensorMap,
        p1::IndexTuple, p2::IndexTuple, f1::FusionTree, f2::FusionTree,
fusiontreemap)

Only works for tensor maps with UniqueFusion.

Here `fusiontreemap` is a function that manipulate the fusiontree and gives the
resulting
fusiontrees as keys and coefficients as values. In the Abelian case, it only gives
one pair.

Consider the fusion tree `(f1,f2)` in `tsrc`. After the operation of `fusiontreemap`
(which result in `(f1′, f2′) => coeff`) and the permitation based on
`pdata = (p1..., p2...)` (result in `tsrc2`), the corresponding block of data has
the same
shape as `tdst[f1′, f2′]`.

After running, the block `tdst[f1′, f2′]` is replaced in-place by
`tsrc2*α*coeff+tdst[f1′, f2′]*β`.
"""
function _addabelianblock!(α, tsrc::AbstractTensorMap, β, tdst::AbstractTensorMap,
        p1::IndexTuple, p2::IndexTuple, f1::FusionTree, f2::FusionTree,
fusiontreemap)
    #cod = codomain(tsrc)
    #dom = domain(tsrc)
    (f1′, f2′), coeff = first(fusiontreemap(f1, f2))
    pdata = (p1..., p2...)
    @inbounds axpby!(α*coeff, permutedims(tsrc[f1, f2], pdata), β, tdst[f1′, f2′])
end

"""
    _add_abelian_kernel!(α, tsrc::AbstractTensorMap, β, tdst::AbstractTensorMap,
                         p1::IndexTuple, p2::IndexTuple, fusiontreemap)

Only works for tensor maps with UniqueFusion.

For every fusion tree in `tsrc`, apply the operation `_addabelianblock!`.
"""
function _add_abelian_kernel!(α, tsrc::AbstractTensorMap, β,
tdst::AbstractTensorMap,
                              p1::IndexTuple, p2::IndexTuple, fusiontreemap)
    if Threads.nthreads() > 1
        nstridedthreads = Strided.get_num_threads()
        Strided.set_num_threads(1)
        Threads.@sync for (f1, f2) in fusiontrees(tsrc)
            Threads.@spawn _addabelianblock!(α, tsrc, β, tdst, p1, p2, f1, f2,
                                             fusiontreemap)
        end
        Strided.set_num_threads(nstridedthreads)
```

```julia
    else # debugging is easier this way
        for (f1, f2) in fusiontrees(tsrc)
            _addabelianblock!(α, tsrc, β, tdst, p1, p2, f1, f2, fusiontreemap)
        end
    end
    return nothing
end

"""
    _add_general_kernel!(α, tsrc::AbstractTensorMap, β, tdst::AbstractTensorMap,
                            p1::IndexTuple, p2::IndexTuple, fusiontreemap)

Works for general tensor maps.

Same operations on each fusion trees.
"""
function _add_general_kernel!(α, tsrc::AbstractTensorMap, β,
tdst::AbstractTensorMap,
                                p1::IndexTuple, p2::IndexTuple, fusiontreemap)
    #cod = codomain(tsrc)
    #dom = domain(tsrc)
    #n = length(cod)
    pdata = (p1..., p2...)
    if iszero(β)
        fill!(tdst, β)
    elseif β != 1
        mul!(tdst, β, tdst)
    end
    for (f1, f2) in fusiontrees(tsrc)
        for ((f1′, f2′), coeff) in fusiontreemap(f1, f2)
            @inbounds axpy!(α*coeff, permutedims(tsrc[f1, f2], pdata), tdst[f1′,
f2′])
        end
    end
    return nothing
end

const _add_kernels = (_add_trivial_kernel!, _add_abelian_kernel!,
_add_general_kernel!)

"""
    _add!(α, tsrc::AbstractTensorMap{S}, β, tdst::AbstractTensorMap{S, N₁, N₂},
            p1::IndexTuple{N₁}, p2::IndexTuple{N₂}, fusiontreemap) where {S, N₁, N₂}

Make the general add operation between two tensor maps. For each fusion tree of
`tsrc`,
apply the manipulation `fusiontreemap` which give new fusion trees and the
corresponding
coefficients. Apply the permutation on the data of the new fusion trees based on
`(p1..., p2...)`, where the permutation should be consistent with requirement of the
`fusiontreemap` manipulation.

Return `tdst`, whose data for each fusion tree is replaced in-place by
`(data of new fusion tree of tsrc)*α*coefficient + (data of fusion tree of tdst)*β`.
"""
```

```julia
function _add!(α, tsrc::AbstractTensorMap{S}, β, tdst::AbstractTensorMap{S, N₁, N₂},
                p1::IndexTuple{N₁}, p2::IndexTuple{N₂}, fusiontreemap) where {S,
N₁, N₂}
    @boundscheck begin
        all(i->space(tsrc, p1[i]) == space(tdst, i), 1:N₁) ||
            throw(SpaceMismatch("tsrc = $(codomain(tsrc))←$(domain(tsrc)),
            tdst = $(codomain(tdst))←$(domain(tdst)), p1 = $(p1), p2 = $(p2)"))
        all(i->space(tsrc, p2[i]) == space(tdst, N₁+i), 1:N₂) ||
            throw(SpaceMismatch("tsrc = $(codomain(tsrc))←$(domain(tsrc)),
            tdst = $(codomain(tdst))←$(domain(tdst)), p1 = $(p1), p2 = $(p2)"))
    end

    # do some kind of dispatch which is compiled away if S is known at compile time,
    # and makes the compiler give up quickly if S is unknown
    I = sectortype(S)
    i = I === Trivial ? 1 : (FusionStyle(I) isa UniqueFusion ? 2 : 3)
    if p1 == codomainind(tsrc) && p2 == domainind(tsrc)
        axpby!(α, tsrc, β, tdst)
    else
        _add_kernel! = _add_kernels[i]
        _add_kernel!(α, tsrc, β, tdst, p1, p2, fusiontreemap)
    end
    return tdst
end

"""
    add_permute!(α, tsrc::AbstractTensorMap{S}, β, tdst::AbstractTensorMap{S, N₁,
N₂},
                    p1::IndexTuple{N₁}, p2::IndexTuple{N₂}) where {S, N₁, N₂}

Replace `tdst` with `permutaion(tsrc)*α + tdst*β`.
"""
@propagate_inbounds function add_permute!(α, tsrc::AbstractTensorMap{S},
                                            β, tdst::AbstractTensorMap{S, N₁, N₂},
                                            p1::IndexTuple{N₁},
                                            p2::IndexTuple{N₂}) where {S, N₁, N₂}

    _add!(α, tsrc, β, tdst, p1, p2, (f1, f2)->permute(f1, f2, p1, p2))
end

"""
    add_braid!(α, tsrc::AbstractTensorMap{S}, β, tdst::AbstractTensorMap{S, N₁, N₂},
            p1::IndexTuple{N₁}, p2::IndexTuple{N₂}, levels::IndexTuple) where {S,
N₁, N₂}

Replace `tdst` with `braid(tsrc)*α + tdst*β`.
"""
@propagate_inbounds function add_braid!(α, tsrc::AbstractTensorMap{S},
                                            β, tdst::AbstractTensorMap{S, N₁, N₂},
                                            p1::IndexTuple{N₁},
                                            p2::IndexTuple{N₂},
                                            levels::IndexTuple) where {S, N₁, N₂}

    length(levels) == numind(tsrc) ||
        throw(ArgumentError("incorrect levels $levels for tensor map
```

```
                                                        $(codomain(tsrc)) ← $(domain(tsrc))"))

    levels1 = TupleTools.getindices(levels, codomainind(tsrc))
    levels2 = TupleTools.getindices(levels, domainind(tsrc))
    _add!(α, tsrc, β, tdst, p1, p2, (f1, f2)->braid(f1, f2, levels1, levels2, p1,
p2))
end

"""
    add_transpose!(α, tsrc::AbstractTensorMap{S}, β, tdst::AbstractTensorMap{S, N₁,
N₂},
                    p1::IndexTuple{N₁}, p2::IndexTuple{N₂}) where {S, N₁, N₂}

Replace `tdst` with `transpose(tsrc)*α + tdst*β`.

?????? It seems that transpose is the same with permutation...
"""
@propagate_inbounds function add_transpose!(α, tsrc::AbstractTensorMap{S},
                                            β, tdst::AbstractTensorMap{S, N₁, N₂},
                                            p1::IndexTuple{N₁},
                                            p2::IndexTuple{N₂}) where {S, N₁, N₂}

    _add!(α, tsrc, β, tdst, p1, p2, (f1, f2)->transpose(f1, f2, p1, p2))
end

"""
    add!(α, tsrc::AbstractTensorMap{S}, β, tdst::AbstractTensorMap{S},
            p1::IndexTuple, p2::IndexTuple) where {S}

Replace `tdst` with `permutaion(tsrc)*α + tdst*β`.
"""
@propagate_inbounds function add!(α, tsrc::AbstractTensorMap{S},
                                β, tdst::AbstractTensorMap{S},
                                p1::IndexTuple, p2::IndexTuple) where {S}
    I = sectortype(S)
    if BraidingStyle(I) isa SymmetricBraiding
        add_permute!(α, tsrc, β, tdst, p1, p2)
    else
        throw(ArgumentError("add! without levels is defined only if
                            `BraidingStyle(sectortype(...)) isa
SymmetricBraiding`"))
    end
end

"""
    add!(α, tsrc::AbstractTensorMap{S}, β, tdst::AbstractTensorMap{S},
            p1::IndexTuple, p2::IndexTuple, levels::IndexTuple) where {S}

Replace `tdst` with `braid(tsrc)*α + tdst*β`.
"""
@propagate_inbounds function add!(α, tsrc::AbstractTensorMap{S},
                                β, tdst::AbstractTensorMap{S},
                                p1::IndexTuple, p2::IndexTuple,
                                levels::IndexTuple) where {S}
    add_braid!(α, tsrc, β, tdst, p1, p2, levels)
```

```
end

"""
    trace!(α, tsrc::AbstractTensorMap{S}, β, tdst::AbstractTensorMap{S, N₁, N₂},
            p1::IndexTuple{N₁}, p2::IndexTuple{N₂},
            q1::IndexTuple{N₃}, q2::IndexTuple{N₃}) where {S, N₁, N₂, N₃}


"""
function trace!(α, tsrc::AbstractTensorMap{S}, β, tdst::AbstractTensorMap{S, N₁,
N₂},
                p1::IndexTuple{N₁}, p2::IndexTuple{N₂},
                q1::IndexTuple{N₃}, q2::IndexTuple{N₃}) where {S, N₁, N₂, N₃}

    if !(BraidingStyle(sectortype(S)) isa SymmetricBraiding)
        throw(SectorMismatch("only tensors with symmetric braiding rules can be
contracted;
                                    try `@planar` instead"))
    end
    @boundscheck begin
        all(i->space(tsrc, p1[i]) == space(tdst, i), 1:N₁) ||
            throw(SpaceMismatch("trace: tsrc = $(codomain(tsrc))←$(domain(tsrc)),
                    tdst = $(codomain(tdst))←$(domain(tdst)), p1 = $(p1), p2 =
$(p2)"))
        all(i->space(tsrc, p2[i]) == space(tdst, N₁+i), 1:N₂) ||
            throw(SpaceMismatch("trace: tsrc = $(codomain(tsrc))←$(domain(tsrc)),
                    tdst = $(codomain(tdst))←$(domain(tdst)), p1 = $(p1), p2 =
$(p2)"))
        all(i->space(tsrc, q1[i]) == dual(space(tsrc, q2[i])), 1:N₃) ||
            throw(SpaceMismatch("trace: tsrc = $(codomain(tsrc))←$(domain(tsrc)),
                    q1 = $(q1), q2 = $(q2)"))
    end

    I = sectortype(S)
    if I === Trivial
        cod = codomain(tsrc)
        dom = domain(tsrc)
        n = length(cod)
        pdata = (p1..., p2...)
        TO._trace!(α, tsrc[], β, tdst[], pdata, q1, q2)
    # elseif FusionStyle(I) isa UniqueFusion
    # TODO: is it worth multithreading UniqueFusion case for traces?
    else
        cod = codomain(tsrc)
        dom = domain(tsrc)
        n = length(cod)
        pdata = (p1..., p2...)
        if iszero(β)
            fill!(tdst, β)
        elseif β != 1
            mul!(tdst, β, tdst)
        end
        r1 = (p1..., q1...)
        r2 = (p2..., q2...)
        for (f1, f2) in fusiontrees(tsrc)
```

```
            for ((f1′, f2′), coeff) in permute(f1, f2, r1, r2)
                f1′′, g1 = split(f1′, N₁)
                f2′′, g2 = split(f2′, N₂)
                if g1 == g2
                    coeff *= dim(g1.coupled)/dim(g1.uncoupled[1])
                    for i = 2:length(g1.uncoupled)
                        if !(g1.isdual[i])
                            coeff *= twist(g1.uncoupled[i])
                        end
                    end
                    TO._trace!(α*coeff, tsrc[f1, f2], true, tdst[f1′′, f2′′],
pdata, q1, q2)
                end
            end
        end
    end
    return tdst
end

# TODO: contraction with either A or B a rank (1, 1) tensor does not require to
# permute the fusion tree and should therefore be special cased. This will speed
# up MPS algorithms
function contract!(α, A::AbstractTensorMap{S}, B::AbstractTensorMap{S},
                    β, C::AbstractTensorMap{S},
                    oindA::IndexTuple{N₁}, cindA::IndexTuple,
                    oindB::IndexTuple{N₂}, cindB::IndexTuple,
                    p1::IndexTuple, p2::IndexTuple,
                    syms::Union{Nothing, NTuple{3, Symbol}} = nothing) where {S,
N₁, N₂}
    # find optimal contraction scheme
    hsp = has_shared_permute
    ipC = TupleTools.invperm((p1..., p2...))
    oindAinC = TupleTools.getindices(ipC, ntuple(n->n, N₁))
    oindBinC = TupleTools.getindices(ipC, ntuple(n->n+N₁, N₂))

    qA = TupleTools.sortperm(cindA)
    cindA′ = TupleTools.getindices(cindA, qA)
    cindB′ = TupleTools.getindices(cindB, qA)

    qB = TupleTools.sortperm(cindB)
    cindA′′ = TupleTools.getindices(cindA, qB)
    cindB′′ = TupleTools.getindices(cindB, qB)

    dA, dB, dC = dim(A), dim(B), dim(C)

    # keep order A en B, check possibilities for cind
    memcost1 = memcost2 = dC*(!hsp(C, oindAinC, oindBinC))
    memcost1 += dA*(!hsp(A, oindA, cindA′)) +
                dB*(!hsp(B, cindB′, oindB))
    memcost2 += dA*(!hsp(A, oindA, cindA′′)) +
                dB*(!hsp(B, cindB′′, oindB))

    # reverse order A en B, check possibilities for cind
    memcost3 = memcost4 = dC*(!hsp(C, oindBinC, oindAinC))
    memcost3 += dB*(!hsp(B, oindB, cindB′)) +
```

```
                dA∗(!hsp(A, cindA′, oindA))
    memcost4 += dB∗(!hsp(B, oindB, cindB′′)) +
                dA∗(!hsp(A, cindA′′, oindA))

    if min(memcost1, memcost2) <= min(memcost3, memcost4)
        if memcost1 <= memcost2
            return _contract!(α, A, B, β, C, oindA, cindA′, oindB, cindB′, p1, p2,
syms)
        else
            return _contract!(α, A, B, β, C, oindA, cindA′′, oindB, cindB′′, p1,
p2, syms)
        end
    else
        p1′ = map(n->ifelse(n>N₁, n-N₁, n+N₂), p1)
        p2′ = map(n->ifelse(n>N₁, n-N₁, n+N₂), p2)
        if memcost3 <= memcost4
            return _contract!(α, B, A, β, C, oindB, cindB′, oindA, cindA′, p1′,
p2′, syms)
        else
            return _contract!(α, B, A, β, C, oindB, cindB′′, oindA, cindA′′, p1′,
p2′, syms)
        end
    end
end

function _contract!(α, A::AbstractTensorMap{S}, B::AbstractTensorMap{S},
                    β, C::AbstractTensorMap{S},
                    oindA::IndexTuple{N₁}, cindA::IndexTuple,
                    oindB::IndexTuple{N₂}, cindB::IndexTuple,
                    p1::IndexTuple, p2::IndexTuple,
                    syms::Union{Nothing, NTuple{3, Symbol}} = nothing) where {S,
N₁, N₂}

    if !(BraidingStyle(sectortype(S)) isa SymmetricBraiding)
        throw(SectorMismatch("only tensors with symmetric braiding rules can be
contracted;
                              try `@planar` instead"))
    end
    copyA = false
    if BraidingStyle(sectortype(S)) isa Fermionic
        for i in cindA
            if !isdual(space(A, i))
                copyA = true
            end
        end
    end
    if syms === nothing
        A′ = permute(A, oindA, cindA; copy = copyA)
        B′ = permute(B, cindB, oindB)
    else
        A′ = cached_permute(syms[1], A, oindA, cindA; copy = copyA)
        B′ = cached_permute(syms[2], B, cindB, oindB)
    end
    if BraidingStyle(sectortype(S)) isa Fermionic
        for i in domainind(A′)
```

```
            if !isdual(space(A′, i))
                A′ = twist!(A′, i)
            end
        end
    end
    ipC = TupleTools.invperm((p1..., p2...))
    oindAinC = TupleTools.getindices(ipC, ntuple(n->n, N₁))
    oindBinC = TupleTools.getindices(ipC, ntuple(n->n+N₁, N₂))
    if has_shared_permute(C, oindAinC, oindBinC)
        C′ = permute(C, oindAinC, oindBinC)
        mul!(C′, A′, B′, α, β)
    else
        if syms === nothing
            C′ = A′*B′
        else
            p1′ = ntuple(identity, N₁)
            p2′ = N₁ .+ ntuple(identity, N₂)
            TC = eltype(C)
            C′ = TO.cached_similar_from_indices(syms[3], TC, oindA, oindB, p1′,
p2′, A, B,
                                                                :N, :N)
            mul!(C′, A′, B′)
        end
        add!(α, C′, β, C, p1, p2)
    end
    return C
end

# Add support for cache and API (`@tensor` macro & friends) from
  TensorOperations.jl:
# compatibility layer
function TensorOperations.memsize(t::TensorMap)
    s = 0
    for (c, b) in blocks(t)
        s += sizeof(b)
    end
    return s
end
TensorOperations.memsize(t::AdjointTensorMap) = TensorOperations.memsize(t')

function TO.similarstructure_from_indices(T::Type, p1::IndexTuple, p2::IndexTuple,
        A::AbstractTensorMap, CA::Symbol = :N)
    if CA == :N
        _similarstructure_from_indices(T, p1, p2, A)
    else
        p1 = adjointtensorindices(A, p1)
        p2 = adjointtensorindices(A, p2)
        _similarstructure_from_indices(T, p1, p2, adjoint(A))
    end
end

function TO.similarstructure_from_indices(T::Type, poA::IndexTuple, poB::IndexTuple,
        p1::IndexTuple, p2::IndexTuple,
        A::AbstractTensorMap, B::AbstractTensorMap,
        CA::Symbol = :N, CB::Symbol = :N)
```

```
    if CA == :N && CB == :N
        _similarstructure_from_indices(T, poA, poB, p1, p2, A, B)
    elseif CA == :C && CB == :N
        poA = adjointtensorindices(A, poA)
        _similarstructure_from_indices(T, poA, poB, p1, p2, adjoint(A), B)
    elseif CA == :N && CB == :C
        poB = adjointtensorindices(B, poB)
        _similarstructure_from_indices(T, poA, poB, p1, p2, A, adjoint(B))
    else
        poA = adjointtensorindices(A, poA)
        poB = adjointtensorindices(B, poB)
        _similarstructure_from_indices(T, poA, poB, p1, p2, adjoint(A), adjoint(B))
    end
end

function _similarstructure_from_indices(::Type{T}, p1::IndexTuple{N₁},
p2::IndexTuple{N₂},
        t::AbstractTensorMap{S}) where {T, S<:IndexSpace, N₁, N₂}

    cod = ProductSpace{S, N₁}(space.(Ref(t), p1))
    dom = ProductSpace{S, N₂}(dual.(space.(Ref(t), p2)))
    return dom→cod
end
function _similarstructure_from_indices(::Type{T}, oindA::IndexTuple,
oindB::IndexTuple,
        p1::IndexTuple{N₁}, p2::IndexTuple{N₂},
        tA::AbstractTensorMap{S}, tB::AbstractTensorMap{S}) where {T,
S<:IndexSpace, N₁, N₂}

    spaces = (space.(Ref(tA), oindA)..., space.(Ref(tB), oindB)...)
    cod = ProductSpace{S, N₁}(getindex.(Ref(spaces), p1))
    dom = ProductSpace{S, N₂}(dual.(getindex.(Ref(spaces), p2)))
    return dom→cod
end

TO.scalar(t::AbstractTensorMap) = scalar(t)

function TO.add!(α, tsrc::AbstractTensorMap{S}, CA::Symbol, β,
    tdst::AbstractTensorMap{S, N₁, N₂}, p1::IndexTuple, p2::IndexTuple) where {S,
N₁, N₂}

    if CA == :N
        p = (p1..., p2...)
        pl = TupleTools.getindices(p, codomainind(tdst))
        pr = TupleTools.getindices(p, domainind(tdst))
        add!(α, tsrc, β, tdst, pl, pr)
    else
        p = adjointtensorindices(tsrc, (p1..., p2...))
        pl = TupleTools.getindices(p, codomainind(tdst))
        pr = TupleTools.getindices(p, domainind(tdst))
        add!(α, adjoint(tsrc), β, tdst, pl, pr)
    end
    return tdst
end
```

```julia
function TO.trace!(α, tsrc::AbstractTensorMap{S}, CA::Symbol, β,
    tdst::AbstractTensorMap{S, N₁, N₂}, p1::IndexTuple, p2::IndexTuple,
    q1::IndexTuple, q2::IndexTuple) where {S, N₁, N₂}

    if CA == :N
        p = (p1..., p2...)
        pl = TupleTools.getindices(p, codomainind(tdst))
        pr = TupleTools.getindices(p, domainind(tdst))
        trace!(α, tsrc, β, tdst, pl, pr, q1, q2)
    else
        p = adjointtensorindices(tsrc, (p1..., p2...))
        pl = TupleTools.getindices(p, codomainind(tdst))
        pr = TupleTools.getindices(p, domainind(tdst))
        q1 = adjointtensorindices(tsrc, q1)
        q2 = adjointtensorindices(tsrc, q2)
        trace!(α, adjoint(tsrc), β, tdst, pl, pr, q1, q2)
    end
    return tdst
end

function TO.contract!(α,
    tA::AbstractTensorMap{S}, CA::Symbol,
    tB::AbstractTensorMap{S}, CB::Symbol,
    β, tC::AbstractTensorMap{S, N₁, N₂},
    oindA::IndexTuple, cindA::IndexTuple,
    oindB::IndexTuple, cindB::IndexTuple,
    p1::IndexTuple, p2::IndexTuple,
    syms::Union{Nothing, NTuple{3, Symbol}} = nothing) where {S, N₁, N₂}

    p = (p1..., p2...)
    pl = ntuple(n->p[n], N₁)
    pr = ntuple(n->p[N₁+n], N₂)
    if CA == :N && CB == :N
        contract!(α, tA, tB, β, tC, oindA, cindA, oindB, cindB, pl, pr, syms)
    elseif CA == :N && CB == :C
        oindB = adjointtensorindices(tB, oindB)
        cindB = adjointtensorindices(tB, cindB)
        contract!(α, tA, tB', β, tC, oindA, cindA, oindB, cindB, pl, pr, syms)
    elseif CA == :C && CB == :N
        oindA = adjointtensorindices(tA, oindA)
        cindA = adjointtensorindices(tA, cindA)
        contract!(α, tA', tB, β, tC, oindA, cindA, oindB, cindB, pl, pr, syms)
    elseif CA == :C && CB == :C
        oindA = adjointtensorindices(tA, oindA)
        cindA = adjointtensorindices(tA, cindA)
        oindB = adjointtensorindices(tB, oindB)
        cindB = adjointtensorindices(tB, cindB)
        contract!(α, tA', tB', β, tC, oindA, cindA, oindB, cindB, pl, pr, syms)
    else
        error("unknown conjugation flags: $CA and $CB")
    end
    return tC
end
```