```julia
# BASIC MANIPULATIONS:
#----------------------------------------------
# -> rewrite generic fusion tree in basis of fusion trees in standard form
# -> only depend on Fsymbol

"""
    insertat(f::FusionTree{I, N₁}, i::Int, f2::FusionTree{I, N₂})
    -> <:AbstractDict{<:FusionTree{I, N₁+N₂-1}, <:Number}

Attach a fusion tree `f2` to the uncoupled leg `i` of the fusion tree `f1` and
bring it
into a linear combination of fusion trees in standard form. This requires that
`f2.coupled == f1.uncoupled[i]` and `f1.isdual[i] == false`.
"""
function insertat(f1::FusionTree{I}, i::Int, f2::FusionTree{I, 0}) where {I}
    # this actually removes uncoupled line i, which should be trivial
    (f1.uncoupled[i] == f2.coupled && !f1.isdual[i]) ||
        throw(SectorMismatch("cannot connect $(f2.uncoupled) to
$(f1.uncoupled[i])"))
    coeff = Fsymbol(one(I), one(I), one(I), one(I), one(I), one(I))[1,1]

    uncoupled = TupleTools.deleteat(f1.uncoupled, i)
    coupled = f1.coupled
    isdual = TupleTools.deleteat(f1.isdual, i)
    if length(uncoupled) <= 2
        inner = ()
    else
        inner = TupleTools.deleteat(f1.innerlines, max(1, i-2))
    end
    if length(uncoupled) <= 1
        vertices = ()
    else
        vertices = TupleTools.deleteat(f1.vertices, max(1, i-1))
    end
    f = FusionTree(uncoupled, coupled, isdual, inner, vertices)
    return fusiontreedict(I)(f => coeff)
end
function insertat(f1::FusionTree{I}, i, f2::FusionTree{I, 1}) where {I}
    # identity operation
    (f1.uncoupled[i] == f2.coupled && !f1.isdual[i]) ||
        throw(SectorMismatch("cannot connect $(f2.uncoupled) to
$(f1.uncoupled[i])"))
    coeff = Fsymbol(one(I), one(I), one(I), one(I), one(I), one(I))[1,1,1,1]
    isdual′ = TupleTools.setindex(f1.isdual, f2.isdual[1], i)
    f = FusionTree{I}(f1.uncoupled, f1.coupled, isdual′, f1.innerlines,
f1.vertices)
    return fusiontreedict(I)(f => coeff)
end
function insertat(f1::FusionTree{I}, i, f2::FusionTree{I, 2}) where {I}
    # elementary building block,
    (f1.uncoupled[i] == f2.coupled && !f1.isdual[i]) ||
        throw(SectorMismatch("cannot connect $(f2.uncoupled) to
$(f1.uncoupled[i])"))
    uncoupled = f1.uncoupled
```

```
    coupled = f1.coupled
    inner = f1.innerlines
    b, c = f2.uncoupled
    isdual = f1.isdual
    isdualb, isdualc = f2.isdual
    if i == 1
        uncoupled′ = (b, c, tail(uncoupled)...)
        isdual′ = (isdualb, isdualc, tail(isdual)...)
        inner′ = (uncoupled[1], inner...)
        vertices′ = (f2.vertices..., f1.vertices...)
        coeff = Fsymbol(one(I), one(I), one(I), one(I), one(I), one(I))[1,1,1,1]
        f′ = FusionTree(uncoupled′, coupled, isdual′, inner′, vertices′)
        return fusiontreedict(I)(f′ => coeff)
    end
    uncoupled′ = TupleTools.insertafter(TupleTools.setindex(uncoupled, b, i), i,
(c,))
    isdual′ = TupleTools.insertafter(TupleTools.setindex(isdual, isdualb, i), i,
(isdualc,))
    a = i == 2 ? uncoupled[1] : inner[i-2]
    d = i == length(f1) ? coupled : inner[i-1]
    e′ = uncoupled[i]
    if FusionStyle(I) isa MultiplicityFreeFusion
        local newtrees
        for e in a ⊗ b
            coeff = conj(Fsymbol(a, b, c, d, e, e′))
            iszero(coeff) && continue
            inner′ = TupleTools.insertafter(inner, i-2, (e,))
            f′ = FusionTree(uncoupled′, coupled, isdual′, inner′)
            if @isdefined newtrees
                push!(newtrees, f′=> coeff)
            else
                newtrees = fusiontreedict(I)(f′ => coeff)
            end
        end
        return newtrees
    else
        local newtrees
        κ = f2.vertices[1]
        λ = f1.vertices[i-1]
        for e in a ⊗ b
            inner′ = TupleTools.insertafter(inner, i-2, (e,))
            Fmat = Fsymbol(a, b, c, d, e, e′)
            for μ = 1:size(Fmat, 1), ν = 1:size(Fmat, 2)
                coeff = conj(Fmat[μ,ν,κ,λ])
                iszero(coeff) && continue
                vertices′ = TupleTools.setindex(f1.vertices, ν, i-1)
                vertices′ = TupleTools.insertafter(vertices′, i-2, (μ,))
                f′ = FusionTree(uncoupled′, coupled, isdual′, inner′, vertices′)
                if @isdefined newtrees
                    push!(newtrees, f′=> coeff)
                else
                    newtrees = fusiontreedict(I)(f′ => coeff)
                end
            end
```

```
        end
        return newtrees
    end
end
function insertat(f1::FusionTree{I,N₁}, i, f2::FusionTree{I,N₂}) where {I,N₁,N₂}
    F = fusiontreetype(I, N₁ + N₂ - 1)
    (f1.uncoupled[i] == f2.coupled && !f1.isdual[i]) ||
        throw(SectorMismatch("cannot connect $(f2.uncoupled) to
$(f1.uncoupled[i])"))
    coeff = Fsymbol(one(I), one(I), one(I), one(I), one(I), one(I))[1,1]
    T = typeof(coeff)
    if length(f1) == 1
        return fusiontreedict(I){F,T}(f2 => coeff)
    end
    if i == 1
        uncoupled = (f2.uncoupled..., tail(f1.uncoupled)...)
        isdual = (f2.isdual..., tail(f1.isdual)...)
        inner = (f2.innerlines..., f2.coupled, f1.innerlines...)
        vertices = (f2.vertices..., f1.vertices...)
        coupled = f1.coupled
        f′ = FusionTree(uncoupled, coupled, isdual, inner, vertices)
        return fusiontreedict(I){F,T}(f′ => coeff)
    else # recursive definition
        N2 = length(f2)
        f2′, f2′′ = split(f2, N2 - 1)
        local newtrees::fusiontreedict(I){F,T}
        for (f, coeff) in insertat(f1, i, f2′′)
            for (f′, coeff′) in insertat(f, i, f2′)
                if @isdefined newtrees
                    coeff′′ = coeff*coeff′
                    newtrees[f′] = get(newtrees, f′, zero(coeff′′)) + coeff′′
                else
                    newtrees = fusiontreedict(I){F,T}(f′ => coeff*coeff′)
                end
            end
        end
        return newtrees
    end
end

"""
    split(f::FusionTree{I, N}, M::Int)
    -> (::FusionTree{I, M}, ::FusionTree{I, N-M+1})

Split a fusion tree into two. The first tree has as uncoupled sectors the first `M`
uncoupled sectors of the input tree `f`, whereas its coupled sector corresponds to
the
internal sector between uncoupled sectors `M` and `M+1` of the original tree `f`.
The
second tree has as first uncoupled sector that same internal sector of `f`,
followed by
remaining `N-M` uncoupled sectors of `f`. It couples to the same sector as `f`.
This
operation is the inverse of `insertat` in the sense that if
```

```julia
`f1, f2 = split(t, M) ⇒ f == insertat(f2, 1, f1)`.
"""
@inline function split(f::FusionTree{I, N}, M::Int) where {I, N}
    if M > N || M < 0
        throw(ArgumentError("M should be between 0 and N = $N"))
    elseif M === N
        (f, FusionTree{I}((f.coupled,), f.coupled, (false,), (), ()))
    elseif M === 1
        isdual1 = (f.isdual[1],)
        isdual2 = Base.setindex(f.isdual, false, 1)
        f1 = FusionTree{I}((f.uncoupled[1],), f.uncoupled[1], isdual1, (), ())
        f2 = FusionTree{I}(f.uncoupled, f.coupled, isdual2, f.innerlines,
f.vertices)
        return f1, f2
    elseif M === 0
        f1 = FusionTree{I}((), one(I), (), ())
        uncoupled2 = (one(I), f.uncoupled...)
        coupled2 = f.coupled
        isdual2 = (false, f.isdual...)
        innerlines2 = N >= 2 ? (f.uncoupled[1], f.innerlines...) : ()
        if FusionStyle(I) isa GenericFusion
            vertices2 = (1, f.vertices...)
            return f1, FusionTree{I}(uncoupled2, coupled2, isdual2, innerlines2,
vertices2)
        else
            return f1, FusionTree{I}(uncoupled2, coupled2, isdual2, innerlines2)
        end
    else
        uncoupled1 = ntuple(n->f.uncoupled[n], M)
        isdual1 = ntuple(n->f.isdual[n], M)
        innerlines1 = ntuple(n->f.innerlines[n], max(0, M-2))
        coupled1 = f.innerlines[M-1]
        vertices1 = ntuple(n->f.vertices[n], M-1)

        uncoupled2 = ntuple(N - M + 1) do n
            n == 1 ? f.innerlines[M - 1] : f.uncoupled[M + n - 1]
        end
        isdual2 = ntuple(N - M + 1) do n
            n == 1 ? false : f.isdual[M + n - 1]
        end
        innerlines2 = ntuple(n->f.innerlines[M-1+n], N-M-1)
        coupled2 = f.coupled
        vertices2 = ntuple(n->f.vertices[M-1+n], N-M)

        f1 = FusionTree{I}(uncoupled1, coupled1, isdual1, innerlines1, vertices1)
        f2 = FusionTree{I}(uncoupled2, coupled2, isdual2, innerlines2, vertices2)
        return f1, f2
    end
end

"""
    merge(f1::FusionTree{I, N₁}, f2::FusionTree{I, N₂}, c::I, μ = nothing)
    -> <:AbstractDict{<:FusionTree{I, N₁+N₂}, <:Number}
```

```julia
Merge two fusion trees together to a linear combination of fusion trees whose
uncoupled
sectors are those of `f1` followed by those of `f2`, and where the two coupled
sectors of
`f1` and `f2` are further fused to `c`. In case of
`FusionStyle(I) == GenericFusion()`, also a degeneracy label `μ` for the fusion of
the coupled sectors of `f1` and `f2` to `c` needs to be specified.
"""
function merge(f1::FusionTree{I, N₁}, f2::FusionTree{I, N₂},
                c::I, μ = nothing) where {I, N₁, N₂}
    if FusionStyle(I) isa GenericFusion && μ === nothing
        throw(ArgumentError("vertex label for merging required"))
    end
    if !(c in f1.coupled ⊗ f2.coupled)
        throw(SectorMismatch("cannot fuse sectors $(f1.coupled) and $(f2.coupled)
to $c"))
    end
    f0 = FusionTree((f1.coupled, f2.coupled), c, (false, false), (), (μ,))
    f, coeff = first(insertat(f0, 1, f1)) # takes fast path, single output
    @assert coeff == one(coeff)
    return insertat(f, N₁+1, f2)
end
function merge(f1::FusionTree{I, 0}, f2::FusionTree{I, 0}, c::I, μ =nothing) where
{I}
    c == one(I) ||
        throw(SectorMismatch("cannot fuse sectors $(f1.coupled) and $(f2.coupled)
to $c"))
    return fusiontreedict(I)(f1=>Fsymbol(c, c, c, c, c, c))
end

# ELEMENTARY DUALITY MANIPULATIONS: A- and B-moves
#---------------------------------------------------------
# -> elementary manipulations that depend on the duality (rigidity) and pivotal
  structure
# -> planar manipulations that do not require braiding, everything is in Fsymbol
  (A/Bsymbol)
# -> B-move (bendleft, bendright) is simple in standard basis
# -> A-move (foldleft, foldright) is complicated, needs to be reexpressed in
  standard form

# change to N₁ - 1, N₂ + 1
function bendright(f1::FusionTree{I, N₁}, f2::FusionTree{I, N₂}) where {I<:Sector,
N₁, N₂}
    # map final splitting vertex (a, b)<-c to fusion vertex a<-(c, dual(b))
    @assert N₁ > 0
    c = f1.coupled
    a = N₁ == 1 ? one(I) : (N₁ == 2 ? f1.uncoupled[1] : f1.innerlines[end])
    b = f1.uncoupled[N₁]

    uncoupled1 = Base.front(f1.uncoupled)
    isdual1 = Base.front(f1.isdual)
    inner1 = N₁ > 2 ? Base.front(f1.innerlines) : ()
    vertices1 = N₁ > 1 ? Base.front(f1.vertices) : ()
    f1′ = FusionTree(uncoupled1, a, isdual1, inner1, vertices1)
```

```
    uncoupled2 = (f2.uncoupled..., dual(b))
    isdual2 = (f2.isdual..., !(f1.isdual[N₁]))
    inner2 = N₂ > 1 ? (f2.innerlines..., c) : ()

    if FusionStyle(I) isa MultiplicityFreeFusion
        coeff = sqrtdim(c) * isqrtdim(a) * Bsymbol(a, b, c)
        if f1.isdual[N₁]
            coeff *= conj(frobeniusschur(dual(b)))
        end
        vertices2 = N₂ > 0 ? (f2.vertices..., nothing) : ()
        f2′ = FusionTree(uncoupled2, a, isdual2, inner2, vertices2)
        return SingletonDict( (f1′, f2′) => coeff )
    else
        local newtrees
        Bmat = Bsymbol(a, b, c)
        μ = N₁ > 1 ? f1.vertices[end] : 1
        for ν = 1:size(Bmat, 2)
            coeff = sqrtdim(c) * isqrtdim(a) * Bmat[μ,ν]
            iszero(coeff) && continue
            if f1.isdual[N₁]
                coeff *= conj(frobeniusschur(dual(b)))
            end
            vertices2 = N₂ > 0 ? (f2.vertices..., ν) : ()
            f2′ = FusionTree(uncoupled2, a, isdual2, inner2, vertices2)
            if @isdefined newtrees
                push!(newtrees, (f1′, f2′) => coeff)
            else
                newtrees = FusionTreeDict( (f1′, f2′) => coeff )
            end
        end
        return newtrees
    end
end
# change to N₁ + 1, N₂ - 1
function bendleft(f1::FusionTree{I}, f2::FusionTree{I}) where I
    # map final fusion vertex c<-(a, b) to splitting vertex (c, dual(b))<-a
    return fusiontreedict(I)((f1′, f2′) => conj(coeff) for
                                    ((f2′, f1′), coeff) in bendright(f2, f1))
end

# change to N₁ - 1, N₂ + 1
function foldright(f1::FusionTree{I, N₁}, f2::FusionTree{I, N₂}) where {I<:Sector,
N₁, N₂}
    # map first splitting vertex (a, b)<-c to fusion vertex b<-(dual(a), c)
    @assert N₁ > 0
    if FusionStyle(I) isa UniqueFusion
        a = f1.uncoupled[1]
        isduala = f1.isdual[1]
        factor = sqrtdim(a)
        if !isduala
            factor *= frobeniusschur(a)
        end
        c1 = dual(a)
```

```
        c2 = f1.coupled
        c = first(c1 ⊗ c2)
        fl = FusionTree{I}(Base.tail(f1.uncoupled), c, Base.tail(f1.isdual))
        fr = FusionTree{I}((c1, f2.uncoupled...), c, (!isduala, f2.isdual...))
        return fusiontreedict(I)((fl,fr)=>1)
    else
        a = f1.uncoupled[1]
        isduala = f1.isdual[1]
        factor = sqrtdim(a)
        if !isduala
            factor *= frobeniusschur(a)
        end
        c1 = dual(a)
        c2 = f1.coupled
        hasmultiplicities = FusionStyle(a) isa GenericFusion
        local newtrees
        for c in c1 ⊗ c2
            N₁ == 1 && c != one(c) && continue
            for μ in (hasmultiplicities ? (1:Nsymbol(c1, c2, c)) : (nothing,))
                fc = FusionTree((c1, c2), c, (!isduala, false), (), (μ,))
                for (fl′, coeff1) in insertat(fc, 2, f1)
                    N₁ > 1 && fl′.innerlines[1] != one(I) && continue
                    coupled = fl′.coupled
                    uncoupled = Base.tail(Base.tail(fl′.uncoupled))
                    isdual = Base.tail(Base.tail(fl′.isdual))
                    inner = N₁ <= 3 ? () : Base.tail(Base.tail(fl′.innerlines))
                    vertices = N₁ <= 2 ? () : Base.tail(Base.tail(fl′.vertices))
                    fl = FusionTree{I}(uncoupled, coupled, isdual, inner, vertices)
                    for (fr, coeff2) in insertat(fc, 2, f2)
                        coeff = factor * coeff1 * coeff2
                        if (@isdefined newtrees)
                            newtrees[(fl,fr)] = get(newtrees, (fl, fr),
zero(coeff)) + coeff
                        else
                            newtrees = fusiontreedict(I)((fl,fr)=>coeff)
                        end
                    end
                end
            end
        end
        return newtrees
    end
end
# change to N₁ + 1, N₂ - 1
function foldleft(f1::FusionTree{I}, f2::FusionTree{I}) where I
    # map first fusion vertex c<-(a, b) to splitting vertex (dual(a), c)<-b
    return fusiontreedict(I)((f1′, f2′) => conj(coeff) for
                                        ((f2′, f1′), coeff) in foldright(f2, f1))
end


# COMPOSITE DUALITY MANIPULATIONS PART 1: Repartition and transpose
#-------------------------------------------------------------------
# -> composite manipulations that depend on the duality (rigidity) and pivotal
```

```
    structure
  # -> planar manipulations that do not require braiding, everything is in Fsymbol
    (A/Bsymbol)
  # -> transpose expressed as cyclic permutation
  function iscyclicpermutation(p)
      N = length(p)
      @inbounds for i = 1:N
          p[mod1(i+1, N)] == mod1(p[i] + 1, N) || return false
      end
      return true
  end

  # clockwise cyclic permutation while preserving (N₁, N₂): foldright & bendleft
  function cycleclockwise(f1::FusionTree{I}, f2::FusionTree{I}) where {I<:Sector}
      local newtrees
      if length(f1) > 0
          for ((f1a, f2a), coeffa) in foldright(f1, f2)
              for ((f1b, f2b), coeffb) in bendleft(f1a, f2a)
                  coeff = coeffa * coeffb
                  if (@isdefined newtrees)
                      newtrees[(f1b,f2b)] = get(newtrees, (f1b, f2b), zero(coeff)) +
  coeff
                  else
                      newtrees = fusiontreedict(I)((f1b,f2b)=>coeff)
                  end
              end
          end
      else
          for ((f1a, f2a), coeffa) in bendleft(f1, f2)
              for ((f1b, f2b), coeffb) in foldright(f1a, f2a)
                  coeff = coeffa * coeffb
                  if (@isdefined newtrees)
                      newtrees[(f1b,f2b)] = get(newtrees, (f1b, f2b), zero(coeff)) +
  coeff
                  else
                      newtrees = fusiontreedict(I)((f1b,f2b)=>coeff)
                  end
              end
          end
      end
      return newtrees
  end

  # anticlockwise cyclic permutation while preserving (N₁, N₂): foldleft & bendright
  function cycleanticlockwise(f1::FusionTree{I}, f2::FusionTree{I}) where {I<:Sector}
      local newtrees
      if length(f2) > 0
          for ((f1a, f2a), coeffa) in foldleft(f1, f2)
              for ((f1b, f2b), coeffb) in bendright(f1a, f2a)
                  coeff = coeffa * coeffb
                  if (@isdefined newtrees)
                      newtrees[(f1b,f2b)] = get(newtrees, (f1b, f2b), zero(coeff)) +
  coeff
                  else
```

```
                    newtrees = fusiontreedict(I)((f1b,f2b)=>coeff)
                end
            end
        end
    else
        for ((f1a, f2a), coeffa) in bendright(f1, f2)
            for ((f1b, f2b), coeffb) in foldleft(f1a, f2a)
                coeff = coeffa * coeffb
                if (@isdefined newtrees)
                    newtrees[(f1b,f2b)] = get(newtrees, (f1b, f2b), zero(coeff)) +
coeff
                else
                    newtrees = fusiontreedict(I)((f1b,f2b)=>coeff)
                end
            end
        end
    end
    return newtrees
end

# repartition double fusion tree
"""
    repartition(f1::FusionTree{I, N₁}, f2::FusionTree{I, N₂}, N::Int) where {I,
N₁, N₂}
    -> <:AbstractDict{Tuple{FusionTree{I, N}, FusionTree{I, N₁+N₂-N}}, <:Number}

Input is a double fusion tree that describes the fusion of a set of incoming
uncoupled
sectors to a set of outgoing uncoupled sectors, represented using the individual
trees of
outgoing (`f1`) and incoming sectors (`f2`) respectively (with identical coupled
sector
`f1.coupled == f2.coupled`). Computes new trees and corresponding coefficients
obtained from
repartitioning the tree by bending incoming to outgoing sectors (or vice versa) in
order to
have `N` outgoing sectors.
"""
@inline function repartition(f1::FusionTree{I, N₁},
                            f2::FusionTree{I, N₂},
                            N::Int) where {I<:Sector, N₁, N₂}
    f1.coupled == f2.coupled || throw(SectorMismatch())
    @assert 0 <= N <= N₁+N₂
    return _recursive_repartition(f1, f2, Val(N))
end

function _recursive_repartition(f1::FusionTree{I, N₁},
                                f2::FusionTree{I, N₂},
                                ::Val{N}) where {I<:Sector, N₁, N₂, N}
    # recursive definition is only way to get correct number of loops for
    # GenericFusion, but is too complex for type inference to handle, so we
    # precompute the parameters of the return type
    F1 = fusiontreetype(I, N)
    F2 = fusiontreetype(I, N₁ + N₂ - N)
```

```
    coeff = @inbounds Fsymbol(one(I), one(I), one(I), one(I), one(I),
one(I))[1,1,1,1]
    T = typeof(coeff)
    if N == N₁
        return fusiontreedict(I){Tuple{F1, F2}, T}( (f1, f2) => coeff)
    else
        local newtrees::fusiontreedict(I){Tuple{F1, F2}, T}
        for ((f1′, f2′), coeff1) in (N < N₁ ? bendright(f1, f2) : bendleft(f1, f2))
            for ((f1′′, f2′′), coeff2) in _recursive_repartition(f1′, f2′, Val(N))
                if (@isdefined newtrees)
                    push!(newtrees, (f1′′, f2′′) => coeff1*coeff2)
                else
                    newtrees =
                        fusiontreedict(I){Tuple{F1, F2}, T}((f1′′, f2′′) =>
coeff1*coeff2)
                end
            end
        end
        return newtrees
    end
end

# transpose double fusion tree
const transposecache = LRU{Any, Any}(; maxsize = 10^5)
const usetransposecache = Ref{Bool}(true)

"""
    transpose(f1::FusionTree{I}, f2::FusionTree{I},
            p1::NTuple{N₁, Int}, p2::NTuple{N₂, Int}) where {I, N₁, N₂}
    -> <:AbstractDict{Tuple{FusionTree{I, N₁}, FusionTree{I, N₂}}, <:Number}

Input is a double fusion tree that describes the fusion of a set of incoming
uncoupled
sectors to a set of outgoing uncoupled sectors, represented using the individual
trees of
outgoing (`t1`) and incoming sectors (`t2`) respectively (with identical coupled
sector
`t1.coupled == t2.coupled`). Computes new trees and corresponding coefficients
obtained from
repartitioning and permuting the tree such that sectors `p1` become outgoing and
sectors
`p2` become incoming.
"""
function Base.transpose(f1::FusionTree{I}, f2::FusionTree{I},
                    p1::IndexTuple{N₁}, p2::IndexTuple{N₂}) where {I<:Sector, N₁,
N₂}
    N = N₁ + N₂
    @assert length(f1) + length(f2) == N
    p = linearizepermutation(p1, p2, length(f1), length(f2))
    @assert iscyclicpermutation(p)
    if usetransposecache[]
        u = one(I)
        T = eltype(Fsymbol(u, u, u, u, u, u))
        F₁ = fusiontreetype(I, N₁)
```

```
        F₂ = fusiontreetype(I, N₂)
        D = fusiontreedict(I){Tuple{F₁, F₂}, T}
        return _get_transpose(D, (f1, f2, p1, p2))
    else
        return _transpose((f1, f2, p1, p2))
    end
end

@noinline function _get_transpose(::Type{D}, @nospecialize(key)) where D
    d::D = get!(transposecache, key) do
        _transpose(key)
    end
    return d
end

const TransposeKey{I<:Sector, N₁, N₂} = Tuple{<:FusionTree{I}, <:FusionTree{I},
                                                IndexTuple{N₁}, IndexTuple{N₂}}

function _transpose((f1, f2, p1, p2)::TransposeKey{I,N₁,N₂}) where {I<:Sector, N₁,
N₂}
    N = N₁ + N₂
    p = linearizepermutation(p1, p2, length(f1), length(f2))
    i1 = findfirst(==(1), p)
    @assert i1 !== nothing
    newtrees = repartition(f1, f2, N₁)
    Nhalf = N >> 1
    while 1 < i1 <= Nhalf
        local newtrees′
        for ((f1a, f2a), coeffa) in newtrees
            for ((f1b, f2b), coeffb) in cycleanticlockwise(f1a, f2a)
                coeff = coeffa * coeffb
                if (@isdefined newtrees′)
                    newtrees′[(f1b, f2b)] = get(newtrees′, (f1b, f2b),
zero(coeff)) + coeff
                else
                    newtrees′ = fusiontreedict(I)((f1b, f2b) => coeff)
                end
            end
        end
        newtrees = newtrees′
        i1 -= 1
    end
    while Nhalf < i1
        local newtrees′
        for ((f1a, f2a), coeffa) in newtrees
            for ((f1b, f2b), coeffb) in cycleclockwise(f1a, f2a)
                coeff = coeffa * coeffb
                if (@isdefined newtrees′)
                    newtrees′[(f1b, f2b)] = get(newtrees′, (f1b, f2b),
zero(coeff)) + coeff
                else
                    newtrees′ = fusiontreedict(I)((f1b, f2b) => coeff)
                end
            end
```

```
        end
        newtrees = newtrees′
        i1 = mod1(i1 + 1, N)
    end
    return newtrees
end

# COMPOSITE DUALITY MANIPULATIONS PART 2: Planar traces
#-------------------------------------------------------------------
# -> composite manipulations that depend on the duality (rigidity) and pivotal
  structure
# -> planar manipulations that do not require braiding, everything is in Fsymbol
  (A/Bsymbol)

function planar_trace(f1::FusionTree{I}, f2::FusionTree{I},
                        p1::IndexTuple{N₁}, p2::IndexTuple{N₂},
                        q1::IndexTuple{N₃}, q2::IndexTuple{N₃}) where {I<:Sector, N₁,
N₂, N₃}

    N = N₁ + N₂ + 2N₃
    @assert length(f1) + length(f2) == N
    if N₃ == 0
        return transpose(f1, f2, p1, p2)
    end

    linearindex = (ntuple(identity, Val(length(f1)))...,
                        reverse(length(f1) .+ ntuple(identity, Val(length(f2))))...)


    q1′ = TupleTools.getindices(linearindex, q1)
    q2′ = TupleTools.getindices(linearindex, q2)
    p1′, p2′ = let q′ = (q1′..., q2′...)
        (map(l-> l - count(l .> q′), TupleTools.getindices(linearindex, p1)),
            map(l-> l - count(l .> q′), TupleTools.getindices(linearindex, p2)))
    end

    u = one(I)
    T = typeof(Fsymbol(u, u, u, u, u, u)[1, 1, 1, 1])
    F₁ = fusiontreetype(I, N₁)
    F₂ = fusiontreetype(I, N₂)
    newtrees = FusionTreeDict{Tuple{F₁,F₂}, T}()
    for ((f1′, f2′), coeff′) in repartition(f1, f2, N)
        for (f1′′, coeff′′) in planar_trace(f1′, q1′, q2′)
            for (f12′′′, coeff′′′) in transpose(f1′′, f2′, p1′, p2′)
                coeff = coeff′ * coeff′′ * coeff′′′
                if !iszero(coeff)
                    newtrees[f12′′′] = get(newtrees, f12′′′, zero(coeff)) + coeff
                end
            end
        end
    end
    return newtrees
end
```

```
function planar_trace(f::FusionTree{I,N},
                        q1::IndexTuple{N₃}, q2::IndexTuple{N₃}) where {I<:Sector,
N, N₃}


    u = one(I)
    T = typeof(Fsymbol(u, u, u, u, u, u)[1, 1, 1, 1])
    F = fusiontreetype(I, N - 2*N₃)
    newtrees = FusionTreeDict{F,T}()
    N₃ === 0 && return push!(newtrees, f=>one(T))

    for (i,j) in zip(q1, q2)
        (f.uncoupled[i] == dual(f.uncoupled[j]) && f.isdual[i] != f.isdual[j]) ||
            return newtrees
    end
    k = 1
    local i, j
    while k <= N₃
        if mod1(q1[k] + 1, N) == q2[k]
            i = q1[k]
            j = q2[k]
            break
        elseif mod1(q2[k] + 1, N) == q1[k]
            i = q2[k]
            j = q1[k]
            break
        else
            k += 1
        end
    end
    k > N₃ && throw(ArgumentError("Not a planar trace"))

    q1′ = let i = i, j = j
        map(l->(l - (l>i) - (l>j)), TupleTools.deleteat(q1, k))
    end
    q2′ = let i = i, j = j
        map(l->(l - (l>i) - (l>j)), TupleTools.deleteat(q2, k))
    end
    for (f′, coeff′) in elementary_trace(f, i)
        for (f′′, coeff′′) in planar_trace(f′, q1′, q2′)
            coeff = coeff′ * coeff′′
            if !iszero(coeff)
                newtrees[f′′] = get(newtrees, f′′, zero(coeff)) + coeff
            end
        end
    end
    return newtrees
end

# trace two neighbouring indices of a single fusion tree
function elementary_trace(f::FusionTree{I, N}, i) where {I<:Sector, N}
    (N > 1 && 1 <= i <= N) ||
        throw(ArgumentError("Cannot trace outputs i=$i and i+1 out of only $N
outputs"))
```

```
    i < N || f.coupled == one(I) ||
        throw(ArgumentError("Cannot trace outputs i=$N and 1 of fusion tree that
couples to non-trivial sector"))

    u = one(I)
    T = typeof(Fsymbol(u,u,u,u,u,u)[1,1,1,1])
    F = fusiontreetype(I, N-2)
    newtrees = FusionTreeDict{F,T}()

    j = mod1(i+1, N)
    b = f.uncoupled[i]
    b′ = f.uncoupled[j]
    # if trace is zero, return empty dict
    (b == dual(b′) && f.isdual[i] != f.isdual[j]) || return newtrees
    if i < N
        a = i == 1 ? one(I) : (i == 2 ? f.uncoupled[1] : f.innerlines[i-2])
        d = i == N-1 ? f.coupled : f.innerlines[i]
        a == d || return newtrees
        uncoupled′ = TupleTools.deleteat(TupleTools.deleteat(f.uncoupled, i+1), i)
        isdual′ = TupleTools.deleteat(TupleTools.deleteat(f.isdual, i+1), i)
        coupled′ = f.coupled
        if N <= 4
            inner′ = ()
        else
            inner′ = i <= 2 ? Base.tail(Base.tail(f.innerlines)) :
                        TupleTools.deleteat(TupleTools.deleteat(f.innerlines,
i-1), i-2)
        end
        if N <= 3
            vertices′ = ()
        else
            vertices′ = i <= 2 ? Base.tail(Base.tail(f.vertices)) :
                        TupleTools.deleteat(TupleTools.deleteat(f.vertices, i),
i-1)
        end
        f′ = FusionTree{I}(uncoupled′, coupled′, isdual′, inner′, vertices′)
        coeff = sqrtdim(b)
        if i > 1
            c = f.innerlines[i-1]
            if FusionStyle(I) isa MultiplicityFreeFusion
                coeff *= Fsymbol(a, b, dual(b), a, c, one(I))
            else
                μ = f.vertices[i-1]
                ν = f.vertices[i]
                coeff *= Fsymbol(a, b, dual(b), a, c, one(I))[μ, ν, 1, 1]
            end
        end
        if f.isdual[i]
            coeff *= frobeniusschur(b)
        end
        push!(newtrees, f′ => coeff)
        return newtrees
    else # i == N
        if N == 2
```

```julia
            f′ = FusionTree{I}((), one(I), (), (), ())
            coeff = sqrtdim(b)
            if !(f.isdual[N])
                coeff *= conj(frobeniusschur(b))
            end
            push!(newtrees, f′ => coeff)
            return newtrees
        end
        uncoupled_ = Base.front(f.uncoupled)
        inner_ = Base.front(f.innerlines)
        coupled_ = f.innerlines[end]
        @assert coupled_ == dual(b)
        isdual_ = Base.front(f.isdual)
        vertices_ = Base.front(f.vertices)
        f_ = FusionTree(uncoupled_, coupled_, isdual_, inner_, vertices_)
        fs = FusionTree((b,), b, (!f.isdual[1],), (), ())
        for (f_′, coeff) = merge(fs, f_, one(I), 1)
            f_′.innerlines[1] == one(I) || continue
            uncoupled′ = Base.tail(Base.tail(f_′.uncoupled))
            isdual′ = Base.tail(Base.tail(f_′.isdual))
            inner′ = N <= 4 ? () : Base.tail(Base.tail(f_′.innerlines))
            vertices′ = N <= 3 ? () : Base.tail(Base.tail(f_′.vertices))
            f′ = FusionTree(uncoupled′, one(I), isdual′, inner′, vertices′)
            coeff *= sqrtdim(b)
            if !(f.isdual[N])
                coeff *= conj(frobeniusschur(b))
            end
            newtrees[f′] = get(newtrees, f′, zero(coeff)) + coeff
        end
        return newtrees
    end
end

# BRAIDING MANIPULATIONS:
#-----------------------------------------------
# -> manipulations that depend on a braiding
# -> requires both Fsymbol and Rsymbol
"""
    artin_braid(f::FusionTree, i; inv::Bool = false) -> <:AbstractDict{typeof(f),
<:Number}

Perform an elementary braid (Artin generator) of neighbouring uncoupled indices
`i` and
`i+1` on a fusion tree `f`, and returns the result as a dictionary of output trees
and
corresponding coefficients.

The keyword `inv` determines whether index `i` will braid above or below index
`i+1`, i.e.
applying `artin_braid(f′, i; inv = true)` to all the outputs `f′` of
`artin_braid(f, i; inv = false)` and collecting the results should yield a single
fusion
tree with non-zero coefficient, namely `f` with coefficient `1`. This keyword has
no effect
```

```
if `BraidingStyle(sectortype(f)) isa SymmetricBraiding`.
"""
function artin_braid(f::FusionTree{I, N}, i; inv::Bool = false) where {I<:Sector,
N}
    1 <= i < N ||
        throw(ArgumentError("Cannot swap outputs i=$i and i+1 out of only $N
outputs"))
    uncoupled = f.uncoupled
    coupled′ = f.coupled
    isdual′ = TupleTools.setindex(f.isdual, f.isdual[i], i+1)
    isdual′ = TupleTools.setindex(isdual′, f.isdual[i+1], i)
    inner = f.innerlines
    vertices = f.vertices
    u = one(I)
    oneT = one(eltype(Rsymbol(u,u,u))) * one(eltype(Fsymbol(u,u,u,u,u,u)))
    if i == 1
        a, b = uncoupled[1], uncoupled[2]
        c = N > 2 ? inner[1] : coupled′
        uncoupled′ = TupleTools.setindex(uncoupled, b, 1)
        uncoupled′ = TupleTools.setindex(uncoupled′, a, 2)
        if FusionStyle(I) isa MultiplicityFreeFusion
            R = oftype(oneT, (inv ? conj(Rsymbol(b, a, c)) : Rsymbol(a, b, c)))
            f′ = FusionTree{I}(uncoupled′, coupled′, isdual′, inner, vertices)
            return fusiontreedict(I)(f′ => R)
        else # GenericFusion
            μ = vertices[1]
            Rmat = inv ? Rsymbol(b, a, c)' : Rsymbol(a, b, c)
            local newtrees
            for ν = 1:size(Rmat, 2)
                R = oftype(oneT, Rmat[μ,ν])
                iszero(R) && continue
                vertices′ = TupleTools.setindex(vertices, ν, 1)
                f′ = FusionTree{I}(uncoupled′, coupled′, isdual′, inner, vertices′)
                if (@isdefined newtrees)
                    push!(newtrees, f′ => R)
                else
                    newtrees = fusiontreedict(I)(f′ => R)
                end
            end
            return newtrees
        end
    end
    # case i > 1:
    b = uncoupled[i]
    d = uncoupled[i+1]
    a = i == 2 ? uncoupled[1] : inner[i-2]
    c = inner[i-1]
    e = i == N-1 ? coupled′ : inner[i]
    uncoupled′ = TupleTools.setindex(uncoupled, d, i)
    uncoupled′ = TupleTools.setindex(uncoupled′, b, i+1)
    if FusionStyle(I) isa UniqueFusion
        inner′ = TupleTools.setindex(inner, first(a ⊗ d), i-1)
        bd = first(b ⊗ d)
        R = oftype(oneT, inv ? conj(Rsymbol(d, b, bd)) : Rsymbol(b, d, bd))
```

```
        f′ = FusionTree{I}(uncoupled′, coupled′, isdual′, inner′)
        return fusiontreedict(I)(f′ => R)
    elseif FusionStyle(I) isa SimpleFusion
        local newtrees
        for c′ in intersect(a ⊗ d, e ⊗ conj(b)) # c′ is f in the figure
            coeff = oftype(oneT, if inv
                    conj(Rsymbol(d, c, e))*conj(Fsymbol(d, a, b, e, c′,
c))*Rsymbol(d, a, c′)
                else
                    Rsymbol(c, d, e)*conj(Fsymbol(d, a, b, e, c′,
c))*conj(Rsymbol(a, d, c′))
                end)
            iszero(coeff) && continue
            inner′ = TupleTools.setindex(inner, c′, i-1)
            f′ = FusionTree{I}(uncoupled′, coupled′, isdual′, inner′)
            if (@isdefined newtrees)
                push!(newtrees, f′ => coeff)
            else
                newtrees = fusiontreedict(I)(f′ => coeff)
            end
        end
        return newtrees
    else # GenericFusion
        local newtrees
        for c′ in intersect(a ⊗ d, e ⊗ conj(b))
            Rmat1 = inv ? Rsymbol(d, c, e)' : Rsymbol(c, d, e)
            Rmat2 = inv ? Rsymbol(d, a, c′)' : Rsymbol(a, d, c′) # There's still
                problem in Jutho's codes
            Fmat = Fsymbol(d, a, b, e, c′, c)
            μ = vertices[i-1]
            ν = vertices[i]
            for σ = 1:Nsymbol(a, d, c′)
                for λ = 1:Nsymbol(c′, b, e)
                    coeff = zero(oneT)
                    for ρ = 1:Nsymbol(d, c, e), κ = 1:Nsymbol(d, a, c′)
                        coeff += Rmat1[ν,ρ]*conj(Fmat[κ,λ,μ,ρ])*conj(Rmat2[σ,κ])
                    end
                    iszero(coeff) && continue
                    vertices′ = TupleTools.setindex(vertices, σ, i-1)
                    vertices′ = TupleTools.setindex(vertices′, λ, i)
                    inner′ = TupleTools.setindex(inner, c′, i-1)
                    f′ = FusionTree{I}(uncoupled′, coupled′, isdual′, inner′,
vertices′)
                    if (@isdefined newtrees)
                        push!(newtrees, f′ => coeff)
                    else
                        newtrees = fusiontreedict(I)(f′ => coeff)
                    end
                end
            end
        end
        return newtrees
    end
end
```

```julia
# braid fusion tree
"""
    braid(f::FusionTree{<:Sector, N}, levels::NTuple{N, Int}, p::NTuple{N, Int})
    -> <:AbstractDict{typeof(t), <:Number}

Perform a braiding of the uncoupled indices of the fusion tree `f` and return the
result as
a `<:AbstractDict` of output trees and corresponding coefficients. The braiding is
determined by specifying that the new sector at position `k` corresponds to the
sector that
was originally at the position `i = p[k]`, and assigning to every index `i` of the
original
fusion tree a distinct level or depth `levels[i]`. This permutation is then
decomposed into
elementary swaps between neighbouring indices, where the swaps are applied as
braids such
that if `i` and `j` cross, ``τ_{i,j}`` is applied if `levels[i] < levels[j]` and
``τ_{j,i}^{-1}`` if `levels[i] > levels[j]`. This does not allow to encode the
most general
braid, but a general braid can be obtained by combining such operations.
"""
function braid(f::FusionTree{I, N},
                levels::NTuple{N, Int},
                p::NTuple{N, Int}) where {I<:Sector, N}
    TupleTools.isperm(p) || throw(ArgumentError("not a valid permutation: $p"))
    if FusionStyle(I) isa UniqueFusion && BraidingStyle(I) isa SymmetricBraiding
        coeff = Rsymbol(one(I), one(I), one(I))
        for i = 2:N
            for j = 1:i-1
                if p[j] > p[i]
                    a, b = f.uncoupled[p[j]], f.uncoupled[p[i]]
                    coeff *= Rsymbol(a, b, first(a ⊗ b))
                end
            end
        end
        uncoupled′ = TupleTools._permute(f.uncoupled, p)
        coupled′ = f.coupled
        isdual′ = TupleTools._permute(f.isdual, p)
        f′ = FusionTree{I}(uncoupled′, coupled′, isdual′)
        return fusiontreedict(I)(f′ => coeff)
    else
        coeff = Rsymbol(one(I), one(I), one(I))[1,1]
        trees = FusionTreeDict(f => coeff)
        newtrees = empty(trees)
        for s in permutation2swaps(p)
            inv = levels[s] > levels[s+1]
            for (f, c) in trees
                for (f′, c′) in artin_braid(f, s; inv = inv)
                    newtrees[f′] = get(newtrees, f′, zero(coeff)) + c*c′
                end
            end
            l = levels[s]
            levels = TupleTools.setindex(levels, levels[s+1], s)
```

```
            levels = TupleTools.setindex(levels, l, s+1)
            trees, newtrees = newtrees, trees
            empty!(newtrees)
        end
        return trees
    end
end

# permute fusion tree
"""
    permute(f::FusionTree, p::NTuple{N, Int}) -> <:AbstractDict{typeof(t),
<:Number}

Perform a permutation of the uncoupled indices of the fusion tree `f` and returns
the result
as a `<:AbstractDict` of output trees and corresponding coefficients; this
requires that
`BraidingStyle(sectortype(f)) isa SymmetricBraiding`.
"""
function permute(f::FusionTree{I, N}, p::NTuple{N, Int}) where {I<:Sector, N}
    @assert BraidingStyle(I) isa SymmetricBraiding
    return braid(f, ntuple(identity, Val(N)), p)
end

# braid double fusion tree
const braidcache = LRU{Any, Any}(; maxsize = 10^5)
const usebraidcache_abelian = Ref{Bool}(false)
const usebraidcache_nonabelian = Ref{Bool}(true)

"""
    braid(f1::FusionTree{I}, f2::FusionTree{I},
            levels1::IndexTuple, levels2::IndexTuple,
            p1::IndexTuple{N₁}, p2::IndexTuple{N₂}) where {I<:Sector, N₁, N₂}
    -> <:AbstractDict{Tuple{FusionTree{I, N₁}, FusionTree{I, N₂}}, <:Number}

Input is a fusion-splitting tree pair that describes the fusion of a set of
incoming
uncoupled sectors to a set of outgoing uncoupled sectors, represented using the
splitting
tree `f1` and fusion tree `f2`, such that the incoming sectors `f2.uncoupled` are
fused to
`f1.coupled == f2.coupled` and then to the outgoing sectors `f1.uncoupled`.
Compute new
trees and corresponding coefficients obtained from repartitioning and braiding the
tree such
that sectors `p1` become outgoing and sectors `p2` become incoming. The uncoupled
indices in
splitting tree `f1` and fusion tree `f2` have levels (or depths) `levels1` and
`levels2`
respectively, which determines how indices braid. In particular, if `i` and `j`
cross,
``τ_{i,j}`` is applied if `levels[i] < levels[j]` and ``τ_{j,i}^{-1}`` if
`levels[i] >
levels[j]`. This does not allow to encode the most general braid, but a general
```

```
braid can
be obtained by combining such operations.
"""
function braid(f1::FusionTree{I}, f2::FusionTree{I},
                levels1::IndexTuple, levels2::IndexTuple,
                p1::IndexTuple{N₁}, p2::IndexTuple{N₂}) where {I<:Sector, N₁, N₂}
    @assert length(f1) + length(f2) == N₁ + N₂
    @assert length(f1) == length(levels1) && length(f2) == length(levels2)
    @assert TupleTools.isperm((p1..., p2...))
    if FusionStyle(f1) isa UniqueFusion &&
        BraidingStyle(f1) isa SymmetricBraiding
        if usebraidcache_abelian[]
            u = one(I)
            T = Int
            F₁ = fusiontreetype(I, N₁)
            F₂ = fusiontreetype(I, N₂)
            D = SingletonDict{Tuple{F₁, F₂}, T}
            return _get_braid(D, (f1, f2, levels1, levels2, p1, p2))
        else
            return _braid((f1, f2, levels1, levels2, p1, p2))
        end
    else
        if usebraidcache_nonabelian[]
            u = one(I)
            T = typeof(sqrtdim(u)*Fsymbol(u, u, u, u, u, u)[1,1,1,1]*Rsymbol(u, u,
u)[1,1])
            F₁ = fusiontreetype(I, N₁)
            F₂ = fusiontreetype(I, N₂)
            D = FusionTreeDict{Tuple{F₁, F₂}, T}
            return _get_braid(D, (f1, f2, levels1, levels2, p1, p2))
        else
            return _braid((f1, f2, levels1, levels2, p1, p2))
        end
    end
end

@noinline function _get_braid(::Type{D}, @nospecialize(key)) where D
    d::D = get!(braidcache, key) do
        _braid(key)
    end
    return d
end

const BraidKey{I<:Sector, N₁, N₂} = Tuple{<:FusionTree{I}, <:FusionTree{I},
                                        IndexTuple, IndexTuple,
                                        IndexTuple{N₁}, IndexTuple{N₂}}

function _braid((f1, f2, l1, l2, p1, p2)::BraidKey{I, N₁, N₂}) where {I<:Sector,
N₁, N₂}
    p = linearizepermutation(p1, p2, length(f1), length(f2))
    levels = (l1..., reverse(l2)...)
    local newtrees
    for ((f, f0), coeff1) in repartition(f1, f2, N₁ + N₂)
        for (f′, coeff2) in braid(f, levels, p)
```

```
            for ((f1′, f2′), coeff3) in repartition(f′, f0, N₁)
                if @isdefined newtrees
                    newtrees[(f1′, f2′)] = get(newtrees, (f1′, f2′), zero(coeff3))
+
                        coeff1*coeff2*coeff3
                else
                    newtrees = fusiontreedict(I)( (f1′, f2′) =>
coeff1*coeff2*coeff3 )
                end
            end
        end
    end
    return newtrees
end

"""
    permute(f1::FusionTree{I}, f2::FusionTree{I},
            p1::NTuple{N₁, Int}, p2::NTuple{N₂, Int}) where {I, N₁, N₂}
    -> <:AbstractDict{Tuple{FusionTree{I, N₁}, FusionTree{I, N₂}}, <:Number}

Input is a double fusion tree that describes the fusion of a set of incoming
uncoupled
sectors to a set of outgoing uncoupled sectors, represented using the individual
trees of
outgoing (`t1`) and incoming sectors (`t2`) respectively (with identical coupled
sector
`t1.coupled == t2.coupled`). Computes new trees and corresponding coefficients
obtained from
repartitioning and permuting the tree such that sectors `p1` become outgoing and
sectors
`p2` become incoming.
"""
function permute(f1::FusionTree{I}, f2::FusionTree{I},
                    p1::IndexTuple{N₁}, p2::IndexTuple{N₂}) where {I<:Sector, N₁,
N₂}
    @assert BraidingStyle(I) isa SymmetricBraiding
    levels1 = ntuple(identity, length(f1))
    levels2 = length(f1) .+ ntuple(identity, length(f2))
    return braid(f1, f2, levels1, levels2, p1, p2)
end
```