```
"""
    artin_braid(f::FusionTree, i; inv::Bool = false) ->
<:AbstractDict{typeof(t),<: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{G,N}, i; inv::Bool = false) where {G<: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
    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)
        R = inv ? conj(Rsymbol(b,a,c)) : Rsymbol(a,b,c)
        f′ = FusionTree{G}(uncoupled′, coupled′, isdual′, inner, vertices)
        if FusionStyle(G) isa Abelian
            return SingletonDict(f′ => R)
        elseif FusionStyle(G) isa SimpleNonAbelian
            return FusionTreeDict(f′ => R)
        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(G) isa Abelian
        inner′ = TupleTools.setindex(inner, first(a ⊗ d), i-1)
        R = inv ? conj(Rsymbol(d, b, first(b ⊗ d))) : Rsymbol(b, d, first(b ⊗ d))
        return SingletonDict(FusionTree{G}(uncoupled′, coupled′, isdual′, inner′)
```

```
=> R)
    elseif FusionStyle(G) isa SimpleNonAbelian
        local newtrees
        for c′ in a ⊗ d
            coeff = if inv
                    Rsymbol(a,b,c)*Fsymbol(b,a,d,e,c,c′)*conj(Rsymbol(c′,b,e))
                else
                    conj(Rsymbol(b,a,c))*Fsymbol(b,a,d,e,c,c′)*Rsymbol(b,c′,e)
                end
            inner′ = TupleTools.setindex(inner, c′, i-1)
            f′ = FusionTree{G}(uncoupled′, coupled′, isdual′, inner′)
            if coeff != zero(coeff)
                if @isdefined newtrees
                    newtrees[f′] = coeff
                else
                    newtrees = FusionTreeDict(f′ => coeff)
                end
            end
        end
        return newtrees
    else
        # TODO: implement DegenerateNonAbelian case
        throw(MethodError(artin_braid, (f, i)))
    end
end

# 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 returns the
result
as a `<:AbstractDict` of output trees and corresponding coefficients. The braiding
is
specified by specifying that index `i` goes to position `perm[i]` and assinging to
every
index 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{G,N},
                levels::NTuple{N,Int},
                p::NTuple{N,Int}) where {G<:Sector, N}
    TupleTools.isperm(p) || throw(ArgumentError("not a valid permutation: $p"))
    if FusionStyle(G) isa Abelian && BraidingStyle(G) isa SymmetricBraiding
        coeff = Rsymbol(one(G), one(G), one(G))
        for i = 1:N
            for j = 1:i-1
```

```julia
                if p[j] > p[i]
                    a, b = f.uncoupled[j], f.uncoupled[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{G}(uncoupled′, coupled′, isdual′)
        return SingletonDict(f′=>coeff)
    else
        coeff = Rsymbol(one(G), one(G), one(G))
        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{G,N}, p::NTuple{N,Int}) where {G<:Sector, N}
    @assert BraidingStyle(G) isa SymmetricBraiding
    return braid(f, ntuple(identity, Val(N)), p)
end

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

Split a fusion tree with the first M outgoing indices, and an incoming index
corresponding
to the internal fusion tree index between outgoing indices N and N+1 of the
```

```julia
original tree
`f`; and a second fusion tree whose first outgoing index is that same internal
index. Its
remaining outgoing indices are the N-M outgoing indices of the original tree `f`,
and also
the incoming index is the same. This is in the inverse of `insertat` in the sense
that if
`f1, f2 = split(t, StaticLength(M)) ⇒ f == insertat(f2, 1, f1)`.
"""
split(f::FusionTree{G,N}, ::StaticLength{N}) where {G,N} =
    (f, FusionTree{G}((f.coupled,), f.coupled, (false,), (), ()))
split(f::FusionTree{G,N}, ::StaticLength{1}) where {G,N} =
    (FusionTree{G}((f.uncoupled[1],), f.uncoupled[1], (false,), (), ()), f)
function split(f::FusionTree{G,N}, ::StaticLength{0}) where {G,N}
    f1 = FusionTree{G}((), one(G), (), ())
    uncoupled2 = (one(G), f.uncoupled...)
    coupled2 = f.coupled
    isdual2 = (false, f.isdual...)
    innerlines2 = N >= 2 ? (f.uncoupled[1], f.innerlines...) : ()
    if FusionStyle(G) isa DegenerateNonAbelian
        vertices2 = (1, f.vertices...)
        return f1, FusionTree(uncoupled2, coupled2, isdual2, innerlines2,
vertices2)
    else
        return f1, FusionTree(uncoupled2, coupled2, isdual2, innerlines2)
    end
end
function split(f::FusionTree{G,N}, ::StaticLength{M}) where {G,N,M}
    @assert 1 < M < N
    uncoupled1 = ntuple(n->f.uncoupled[n], Val(M))
    isdual1 = ntuple(n->f.isdual[n], Val(M))
    innerlines1 = M>2 ? ntuple(n->f.innerlines[n], Val(M-2)) : ()
    coupled1 = f.innerlines[M-1]
    vertices1 = ntuple(n->f.vertices[n], Val(M-1))
    f1 = FusionTree(uncoupled1, coupled1, isdual1, innerlines1, vertices1)

    uncoupled2 = (coupled1, ntuple(n->f.uncoupled[M+n], Val(N-M))...)
    isdual2 = (false, ntuple(n->f.isdual[M+n], Val(N-M))...)
    innerlines2 = ntuple(n->f.innerlines[M-1+n], Val(N-M-1))
    coupled2 = f.coupled
    vertices2 = ntuple(n->f.vertices[M-1+n], Val(N-M))
    f2 = FusionTree(uncoupled2, coupled2, isdual2, innerlines2, vertices2)
    return f1, f2
end

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

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
```

```julia
`FusionStyle(G) == DegenerateNonAbelian()`, 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{G,N₁}, f2::FusionTree{G,N₂},
                    c::G, μ = nothing) where {G,N₁,N₂}
    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{G,0}, f2::FusionTree{G,0}, c::G, μ =nothing) where
{G}
    c == one(G) ||
        throw(SectorMismatch("cannot fuse sectors $(f1.coupled) and $(f2.coupled)
to $c"))
    return SingletonDict(f1=>Fsymbol(one(G),one(G),one(G),one(G),one(G),one(G)))
end

"""
    insertat(f::FusionTree{G,N₁}, i, f2::FusionTree{G,N₂})
    -> <:AbstractDict{<:FusionTree{G,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{G}, i, f2::FusionTree{G,0}) where {G}
    # 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(G), one(G), one(G), one(G), one(G), one(G))

    uncoupled = TupleTools.deleteat(f1.uncoupled, i)
    coupled = f1.coupled
    isdual = TupleTools.deleteat(f1.isdual, i)
    inner = TupleTools.deleteat(f1.innerlines, max(1,i-2))
    vertices = TupleTools.deleteat(t1.vertices, max(1, i-1))
    f = FusionTree(uncoupled, coupled, isdual, inner, vertices)
    if FusionStyle(G) isa Abelian
        return SingletonDict(f => coeff)
    elseif FusionStyle(G) isa SimpleNonAbelian
        return FusionTreeDict(f => coeff)
    end
end
function insertat(f1::FusionTree{G}, i, f2::FusionTree{G,1}) where {G}
    # identity operation
    (f1.uncoupled[i] == f2.coupled && !f1.isdual[i]) ||
```

```
        throw(SectorMismatch("cannot connect $(f2.uncoupled) to
$(f1.uncoupled[i])"))
    coeff = Fsymbol(one(G), one(G), one(G), one(G), one(G), one(G))
    isdual′ = TupleTools.setindex(f1.isdual, f2.isdual[1], i)
    f = FusionTree{G}(f1.uncoupled, f1.coupled, isdual′, f1.innerlines,
f1.vertices)
    if FusionStyle(G) isa Abelian
        return SingletonDict(f => coeff)
    elseif FusionStyle(G) isa SimpleNonAbelian
        return FusionTreeDict(f => coeff)
    end
end
function insertat(f1::FusionTree{G}, i, f2::FusionTree{G,2}) where {G}
    # 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(G), one(G), one(G), one(G), one(G), one(G))
        f′ = FusionTree(uncoupled′, coupled, isdual′, inner′, vertices′)
        if FusionStyle(G) isa Abelian
            return SingletonDict(f′ => coeff)
        elseif FusionStyle(G) isa SimpleNonAbelian
            return FusionTreeDict(f′ => coeff)
        end
    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(G) isa Abelian
        e = first(a ⊗ b)
        inner′ = TupleTools.insertafter(inner, i-2, (e,))
        f′ = FusionTree(uncoupled′, coupled, isdual′, inner′)
        coeff = conj(Fsymbol(a,b,c,d,e,e′))
        return SingletonDict(f′ => coeff)
    elseif FusionStyle(G) isa SimpleNonAbelian
        local newtrees
        for e in a ⊗ b
            inner′ = TupleTools.insertafter(inner, i-2, (e,))
            f′ = FusionTree(uncoupled′, coupled, isdual′, inner′)
```

```
            coeff = conj(Fsymbol(a,b,c,d,e,e′))
            if coeff != zero(coeff)
                if @isdefined newtrees
                    newtrees[f′] = coeff
                else
                    newtrees = FusionTreeDict(f′ => coeff)
                end
            end
        end
        return newtrees
    else
        # TODO: implement DegenerateNonAbelian case
        throw(MethodError(insertat, (f1, i, f2)))
    end
end
function insertat(f1::FusionTree{G}, i, f2::FusionTree{G}) where {G}
    (f1.uncoupled[i] == f2.coupled && !f1.isdual[i]) ||
        throw(SectorMismatch("cannot connect $(f2.uncoupled) to
$(f1.uncoupled[i])"))
    if length(f1) == 1
        coeff = Fsymbol(one(G), one(G), one(G), one(G), one(G), one(G))
        if FusionStyle(G) isa Abelian
            return SingletonDict(f2 => coeff)
        elseif FusionStyle(G) isa SimpleNonAbelian
            return FusionTreeDict(f2 => coeff)
        end
    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)
        coeff = Fsymbol(one(G), one(G), one(G), one(G), one(G), one(G))
        if FusionStyle(G) isa Abelian
            return SingletonDict(f′ => coeff)
        elseif FusionStyle(G) isa SimpleNonAbelian
            return FusionTreeDict(f′ => coeff)
        end
    else # recursive definition
        N2 = length(f2)
        f2′, f2′′ = split(f2, StaticLength(N2) - StaticLength(1))
        if FusionStyle(G) isa Abelian
            f, coeff = first(insertat(f1, i, f2′′))
            f′, coeff′ = first(insertat(f, i, f2′))
            return SingletonDict(f′=>coeff*coeff′)
        else
            local newtrees
            for (f, coeff) in insertat(f1, i, f2′′)
                if @isdefined newtrees
                    for (f′, coeff′) in insertat(f, i, f2′)
                        newtrees[f′] = get(newtrees, f′, zero(coeff′)) +
coeff*coeff′
```

```julia
                    end
                else
                    newtrees = insertat(f, i, f2′)
                    for (f′, coeff′) in newtrees
                        newtrees[f′] = coeff*coeff′
                    end
                end
            end
            return newtrees
        end
    end
end

# repartition double fusion tree
"""
    repartition(f1::FusionTree{G,N₁}, f2::FusionTree{G,N₂},
                ::StaticLength{N}) where {G,N₁,N₂,N}
    -> <:AbstractDict{Tuple{FusionTree{G,N}, FusionTree{G,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.
"""
function repartition(f1::FusionTree{G,N₁},
                        f2::FusionTree{G,N₂},
                        V::StaticLength{N}) where {G<:Sector, N₁, N₂, N}
    f1.coupled == f2.coupled || throw(SectorMismatch())
    @assert 0 <= N <= N₁+N₂
    V1 = V
    V2 = StaticLength(N₁)+StaticLength(N₂)-V

    if FusionStyle(f1) isa Abelian || FusionStyle(f1) isa SimpleNonAbelian
        coeff = sqrt(dim(one(G)))*Bsymbol(one(G), one(G), one(G))
        uncoupled = (f1.uncoupled..., map(dual, reverse(f2.uncoupled))...)
        isdual = (f1.isdual..., map(!, reverse(f2.isdual))...)
        inner1ext = isa(StaticLength(N₁), StaticLength{0}) ? () :
                        (isa(StaticLength(N₁), StaticLength{1}) ? (one(G),) :
                            (one(G), first(uncoupled), f1.innerlines...))
        inner2ext = isa(StaticLength(N₂), StaticLength{0}) ? () :
                        (isa(StaticLength(N₂), StaticLength{1}) ? (one(G),) :
                            (one(G), dual(last(uncoupled)), f2.innerlines...))
        innerext = (inner1ext..., f1.coupled, reverse(inner2ext)...) # length
            N₁+N₂+1
        for n = N₁+1:N
             # map fusion vertex c<-(a,b) to splitting vertex (c,dual(b))<-a
            b = dual(uncoupled[n])
```

```
            a = innerext[n+1]
            c = innerext[n]
            coeff *= sqrt(dim(c)/dim(a))*conj(Bsymbol(a,b,c))
            if !isdual[n]
                coeff *= frobeniusschur(dual(b))
            end
        end
        for n = N₁:-1:N+1
            # map splitting vertex (a,b)<-c to fusion vertex a<-(c,dual(b))
            b = uncoupled[n]
            a = innerext[n]
            c = innerext[n+1]
            coeff *= sqrt(dim(c)/dim(a))*Bsymbol(a,b,c)
            if isdual[n]
                coeff *= conj(frobeniusschur(dual(b)))
            end
        end
        uncoupled1 = TupleTools.getindices(uncoupled, ntuple(n->n, V1))
        uncoupled2 = TupleTools.getindices(map(dual,uncoupled),
ntuple(n->N₁+N₂+1-n, V2))
        isdual1 = TupleTools.getindices(isdual, ntuple(n->n, V1))
        isdual2 = TupleTools.getindices(map(!,isdual), ntuple(n->N₁+N₂+1-n, V2))
        innerlines1 = TupleTools.getindices(innerext, ntuple(n->n+2, V1 -
StaticLength(2)))
        innerlines2 = TupleTools.getindices(innerext, ntuple(n->N₁+N₂-n, V2 -
StaticLength(2)))
        c = innerext[N+1]
        f1′ = FusionTree{G}(uncoupled1, c, isdual1, innerlines1)
        f2′ = FusionTree{G}(uncoupled2, c, isdual2, innerlines2)
        return SingletonDict((f1′, f2′)=>coeff)
    else
        # TODO: implement DegenerateNonAbelian case
        throw(MethodError(repartition, (f1, f2, V)))
    end
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{G}, f2::FusionTree{G},
            levels1::IndexTuple, levels2::IndexTuple,
            p1::IndexTuple{N₁}, p2::IndexTuple{N₂}) where {G<:Sector,N₁,N₂}
    -> <:AbstractDict{Tuple{FusionTree{G,N₁}, FusionTree{G,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{G}, f2::FusionTree{G},
                levels1::IndexTuple, levels2::IndexTuple,
                p1::IndexTuple{N₁}, p2::IndexTuple{N₂}) where {G<: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 Abelian &&
        BraidingStyle(f1) isa SymmetricBraiding
        if usebraidcache_abelian[]
            u = one(G)
            T = typeof(sqrt(dim(u))*Fsymbol(u,u,u,u,u,u)*Rsymbol(u,u,u))
            F₁ = fusiontreetype(G, StaticLength(N₁))
            F₂ = fusiontreetype(G, StaticLength(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(G)
            T = typeof(sqrt(dim(u))*Fsymbol(u,u,u,u,u,u)*Rsymbol(u,u,u))
            F₁ = fusiontreetype(G, StaticLength(N₁))
            F₂ = fusiontreetype(G, StaticLength(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
```

```julia
const BraidKey{G<:Sector,N₁,N₂} = Tuple{<:FusionTree{G}, <:FusionTree{G},
                                        IndexTuple, IndexTuple,
                                        IndexTuple{N₁}, IndexTuple{N₂}}

function _braid((f1, f2, l1, l2, p1, p2)::BraidKey{G,N₁,N₂}) where
{G<:Sector,N₁,N₂}
    p = linearizepermutation(p1, p2, length(f1), length(f2))
    levels = (l1..., reverse(l2)...)
    if FusionStyle(f1) isa Abelian
        (f,f0), coeff1 = first(repartition(f1, f2, StaticLength(N₁) +
StaticLength(N₂)))
        f, coeff2 = first(braid(f, levels, p))
        (f1′,f2′), coeff3 = first(repartition(f, f0, StaticLength(N₁)))
        return SingletonDict((f1′,f2′)=>coeff1*coeff2*coeff3)
    elseif FusionStyle(f1) isa SimpleNonAbelian
        (f,f0), coeff1 = first(repartition(f1, f2, StaticLength(N₁) +
StaticLength(N₂)))
        local newtrees
        for (f, coeff2) in braid(f, levels, p)
            (f1′, f2′), coeff3 = first(repartition(f, f0, StaticLength(N₁)))
            if @isdefined newtrees
                newtrees[(f1′,f2′)] = coeff1*coeff2*coeff3
            else
                newtrees = FusionTreeDict((f1′,f2′)=>coeff1*coeff2*coeff3)
            end
        end
        return newtrees
    else
        # TODO: implement DegenerateNonAbelian case
        throw(MethodError(braid, (f1, f2, l1, l2, p1, p2)))
    end
end

"""
    permute(f1::FusionTree{G}, f2::FusionTree{G},
            p1::NTuple{N₁,Int}, p2::NTuple{N₂,Int}) where {G,N₁,N₂}
    -> <:AbstractDict{Tuple{FusionTree{G,N₁}, FusionTree{G,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{G}, f2::FusionTree{G},
                    p1::IndexTuple{N₁}, p2::IndexTuple{N₂}) where {G<:Sector,N₁,N₂}
    @assert BraidingStyle(G) isa SymmetricBraiding
    levels1 = ntuple(identity, length(f1))
```

```
    levels2 = length(f1) .+ ntuple(identity, length(f2))
    return braid(f1, f2, levels1, levels2, p1, p2)
end
```