```julia
# implementation/indices.jl
#
# Implements the index calculations, i.e. converting the tensor labels into
# indices specifying the operations

# Auxiliary tools to manipulate tuples
#-------------------------------------
import Base.tail

# tuple setdiff, assumes b is completely contained in a
tsetdiff(a::Tuple, b::Tuple{}) = a
tsetdiff(a::Tuple{Any}, b::Tuple{Any}) = () #a[1] == b[1] ? () : tseterror()
tsetdiff(a::Tuple, b::Tuple{Any}) = a[1] == b[1] ? tail(a) : (a[1],
tsetdiff(tail(a), b)...)
tsetdiff(a::Tuple, b::Tuple) = tsetdiff(tsetdiff(a, (b[1],)), tail(b))

@noinline tseterror() = throw(ArgumentError("tuples did not meet requirements"))

# tuple unique: assumes that every element appears exactly twice
tunique(src::Tuple) = tunique(src, ())
tunique(src::NTuple{N,Any}, dst::NTuple{N,Any}) where {N} = dst
tunique(src::Tuple, dst::Tuple) = src[1] in dst ? tunique((tail(src)..., src[1]),
dst) : tunique(tail(src), (dst..., src[1]))

# type stable find, returns zero if not found
_findfirst(args...) = (i = findfirst(args...); i === nothing ? 0 : i)
_findnext(args...) = (i = findnext(args...); i === nothing ? 0 : i)
_findlast(args...) = (i = findlast(args...); i === nothing ? 0 : i)

# Auxiliary method to analyze trace indices
#-----------------------------------------
"""
    unique2(itr)

Returns an array containing only those elements that appear exactly once in itr,
and without any elements that appear more than once.
"""
function unique2(itr)
    out = reshape(collect(itr),length(itr))
    i = 1
    while i < length(out)
        inext = _findnext(isequal(out[i]), out, i+1)
        if inext == nothing || inext == 0
            i += 1
            continue
        end
        while !(inext == nothing || inext == 0)
            deleteat!(out,inext)
            inext = _findnext(isequal(out[i]), out, i+1)
        end
        deleteat!(out,i)
    end
    out
end
```

```julia
# Extract index information
#----------------------------
function add_indices(IA::NTuple{NA,Any}, IC::NTuple{NC,Any}) where {NA,NC}
    indCinA = map(l->_findfirst(isequal(l), IA), IC)
    (NA == NC && isperm(indCinA)) || throw(IndexError("invalid index specification:
$IA to $IC"))
    return indCinA
end

function trace_indices(IA::NTuple{NA,Any}, IC::NTuple{NC,Any}) where {NA,NC}
    # trace indices
    isodd(length(IA)-length(IC)) && throw(IndexError("invalid trace specification:
$IA to $IC"))
    Itrace = tunique(tsetdiff(IA, IC))

    cindA1 = map(l->_findfirst(isequal(l), IA), Itrace)
    cindA2 = map(l->_findnext(isequal(l), IA, _findfirst(isequal(l), IA)+1), Itrace)
    indCinA = map(l->_findfirst(isequal(l), IA), IC)

    pA = (indCinA..., cindA1..., cindA2...)
    (isperm(pA) && length(pA) == NA) || throw(IndexError("invalid trace
specification: $IA to $IC"))
    return indCinA, cindA1, cindA2
end

function contract_indices(IA::NTuple{NA,Any}, IB::NTuple{NB,Any},
IC::NTuple{NC,Any}) where {NA,NB,NC}
    # labels
    IAB = (IA..., IB...)
    isodd(length(IAB)-length(IC)) && throw(IndexError("invalid contraction pattern:
$IA and $IB to $IC"))
    Icontract = tunique(tsetdiff(IAB, IC))
    IopenA = tsetdiff(IA, Icontract)
    IopenB = tsetdiff(IB, Icontract)

    # to indices
    cindA = map(l->_findfirst(isequal(l), IA), Icontract)
    cindB = map(l->_findfirst(isequal(l), IB), Icontract)
    oindA = map(l->_findfirst(isequal(l), IA), IopenA)
    oindB = map(l->_findfirst(isequal(l), IB), IopenB)
    indCinoAB = map(l->_findfirst(isequal(l), (IopenA..., IopenB...)), IC)

    if !isperm((oindA..., cindA...)) || !isperm((oindB..., cindB...)) ||
!isperm(indCinoAB)
        throw(IndexError("invalid contraction pattern: $IA and $IB to $IC"))
    end

    return oindA, cindA, oindB, cindB, indCinoAB
end
```