9f. Reductions

Martin Alfaro
PhD in Economics

INTRODUCTION

Reductions are a strategy of computation for **operations that take collections as input and return a single element**. Such operations arise naturally in a wide range of contexts, such as when computing summary statistics (e.g., averages, variances, or maxima of collections).

The reduction process works by iteratively applying an operation to pairs of elements, accumulating the results at each step until the final output is obtained. For example, to compute `sum(x)` for a vector `x`, a reduction would start by adding the first two elements, then add the third element to the partial total, and continue this cumulative process until all elements have been summed.

The method is particularly convenient when we need to transform a vector's elements prior to aggregating the result. By operating on scalars, **reductions sidestep the need to materialize intermediate outputs**, thus reducing memory allocations. This means that, if for example you have to compute `sum(log.(x))`, a reduction would avoid the creation of the intermediate vector `log.(x)`.

INTUITION

Reductions are implemented using a for-loop, with an operator applied to pairs of elements and the resulting output updated in each iteration. A classic example is the summation of all numeric elements in a vector. This involves applying the addition operator `+` to pairs of elements, iteratively updating the accumulated sum. This process is demonstrated below.

```julia
x = rand(100)

foo(x) = sum(x)
```

```
julia> foo(x)
48.447
```

```
x = rand(100)

function foo(x)
    output = 0.

    for i in eachindex(x)
        output = output + x[i]
    end

    return output
end
```

```
julia> foo(x)
48.447
```

```
x = rand(100)

function foo(x)
    output = 0.

    for i in eachindex(x)
        output += x[i]
    end

    return output
end
```

```
julia> foo(x)
48.447
```

In reductions, it's common to see implementations like the last tab, which are based on update operators. Recall that they turn an expression like `x = x + a` into `x += a`.

IMPLEMENTING REDUCTIONS

The implementation of reductions relies on either a binary operator or a two-argument function during each iteration. An example of a binary operator is `+`, which we used above. However, we could've also used its two-argument function form, replacing `output = output + x[i]` with `output = +(output, x[i])`.

The use of two-argument functions expands the scope of reductions, enabling us to compute more complex operations. For instance, it allows us to compute the maximum value of a vector `x`. This requires the function `max`, where `max(a,b)` returns the maximum of the scalars `a` and `b`.

Formally, a reduction requires the binary operation to satisfy **two mathematical properties**:

- **Associativity**: the way in which operations are grouped must not change the result. For example, addition is associative because `(a + b) + c = a + (b + c)`.

- **Existence of an identity element**: there exists an element that, when combined with any other element through a binary operation, leaves that element unchanged. For example, the identity element of addition is `0` because `a + 0 = a`.

The following list indicates the identity elements of each operation.

Operation	**Identity Element**
Sum	0
Product	1
Maximum	-Inf
Minimum	Inf

Identity elements constitute the initial values of the iterative process. Based on them, we next implement reductions for several operations. The examples define the function `foo1` to show the desired outcome, while `foo2` provides the same output via a reduction.

```
x = rand(100)

foo1(x) = sum(x)

function foo2(x)
    output = 0.

    for i in eachindex(x)
        output += x[i]
    end

    return output
end
```

```
x = rand(100)

foo1(x) = prod(x)

function foo2(x)
    output = 1.

    for i in eachindex(x)
        output *= x[i]
    end

    return output
end
```

```julia
x = rand(100)

foo1(x) = maximum(x)

function foo2(x)
    output = -Inf

    for i in eachindex(x)
        output = max(output, x[i])
    end

    return output
end
```

```julia
x = rand(100)

foo1(x) = minimum(x)

function foo2(x)
    output = Inf

    for i in eachindex(x)
        output = min(output, x[i])
    end

    return output
end
```

Below, we visually illustrate how reductions operate in these examples.

REDUCTION 1: sum of [1,2,3,4]

Initial Value: 0

Iteration 1: $[\boxed{0+1}, 2, 3, 4]$

Iteration 2: $[\boxed{1+2}, 3, 4]$

Iteration 3: $[\boxed{3+3}, 4]$

Iteration 4: $\boxed{[6+4]}$

Final Output: $\boxed{10}$

REDUCTION 2: maximum of [1,4,2,8]

Initial Value: $-\text{Inf}$

Iteration 1: $[\boxed{max\{-\text{Inf}, 1\}},\ 4,\ 2,\ 8]$

Iteration 2: $[\boxed{max\{1, 4\}},\ 2,\ 8]$

Iteration 3: $[\boxed{max\{4, 2\}},\ 8]$

Iteration 4: $\boxed{max\{4, 8\}}$

Final Output: $\boxed{8}$

AVOIDING MEMORY ALLOCATIONS VIA REDUCTIONS

One of the primary advantages of reductions is that they avoid the memory allocation of intermediate results.

To illustrate, consider the operation `sum(log.(x))` for a vector `x`. Its computation involves transforming `x` into `log.(x)`, and then summing the transformed elements. By default, broadcasting internally creates a new vector to store the result of `log.(x)`, thereby allocating memory. However, in many cases like this one, we're only interested in the final sum, not the intermediate result. Therefore, an approach that bypasses the allocation of `log.(x)` is beneficial. Reductions accomplish this by defining a scalar `output`, which is iteratively updated by summing the transformed values of `x`. [1]

```julia
x = rand(100)

foo1(x) = sum(log.(x))

function foo2(x)
    output = 0.

    for i in eachindex(x)
        output += log(x[i])
    end

    return output
end
```

```
julia> @btime foo1($x)
  315.584 ns (1 allocation: 896 bytes)

julia> @btime foo2($x)
  296.119 ns (0 allocations: 0 bytes)
```

```julia
x = rand(100)

foo1(x) = prod(log.(x))

function foo2(x)
    output = 1.

    for i in eachindex(x)
        output *= log(x[i])
    end

    return output
end
```

```
julia> @btime foo1($x)
  311.840 ns (1 allocation: 896 bytes)

julia> @btime foo2($x)
  296.061 ns (0 allocations: 0 bytes)
```

```julia
x = rand(100)

foo1(x) = maximum(log.(x))

function foo2(x)
    output = -Inf

    for i in eachindex(x)
        output = max(output, log(x[i]))
    end

    return output
end
```

```
julia> @btime foo1($x)
  482.602 ns (1 allocation: 896 bytes)

julia> @btime foo2($x)
  374.961 ns (0 allocations: 0 bytes)
```

```julia
x = rand(100)

foo1(x) = minimum(log.(x))

function foo2(x)
    output = Inf

    for i in eachindex(x)
        output = min(output, log(x[i]))
    end

    return output
end
```

```
julia> @btime foo1($x)
  487.156 ns (1 allocation: 896 bytes)
julia> @btime foo2($x)
  368.502 ns (0 allocations: 0 bytes)
```

REDUCTIONS VIA BUILT-IN FUNCTIONS

In the previous examples, reductions were implemented manually through explicit for-loops. While this approach makes the underlying mechanics transparent, it also introduces considerable verbosity. In real-world scenarios, for-loops tend to compromise code readability. To address this issue, Julia provides several streamlined alternatives for expressing reductions in a more concise and idiomatic way.

In particular, several functions come with an additional method for applying reductions to a transformed version of the input vector. Such methods are present for functions like `sum`, `prod`, `maximum`, and `minimum`. The syntax is given by `foo(<transforming function>, x)`, where `foo` is one of the functions mentioned and `x` is the vector to be transformed. For instance, the following examples consider reductions for a log transformation.

```julia
x      = rand(100)

foo(x) = sum(log, x)          #same output as sum(log.(x))
```

```
julia> @btime foo($x)
  294.889 ns (0 allocations: 0 bytes)
```

```julia
x      = rand(100)

foo(x) = prod(log, x)         #same output as prod(log.(x))
```

```
julia> @btime foo($x)
  294.763 ns (0 allocations: 0 bytes)
```

```
x      = rand(100)

foo(x) = maximum(log, x)     #same output as maximum(log.(x))
```

```
julia> @btime foo($x)
  579.940 ns (0 allocations: 0 bytes)
```

```
x      = rand(100)

foo(x) = minimum(log, x)     #same output as minimum(log.(x))
```

```
julia> @btime foo($x)
  577.516 ns (0 allocations: 0 bytes)
```

To keep matters simple, we've used the built-in function `log` for transforming `x`. However, the approach supports any form of function, including anonymous ones.

```
x      = rand(100)

foo(x) = sum(a -> 2 * a, x)        #same output as sum(2 .* x)
```

```
julia> @btime foo($x)
  6.493 ns (0 allocations: 0 bytes)
```

```
x      = rand(100)

foo(x) = prod(a -> 2 * a, x)       #same output as prod(2 .* x)
```

```
julia> @btime foo($x)
  6.741 ns (0 allocations: 0 bytes)
```

```
x      = rand(100)

foo(x) = maximum(a -> 2 * a, x)    #same output as maximum(2 .* x)
```

```
julia> @btime foo($x)
  172.547 ns (0 allocations: 0 bytes)
```

```
x      = rand(100)

foo(x) = minimum(a -> 2 * a, x)    #same output as minimum(2 .* x)
```

```
julia> @btime foo($x)
  171.490 ns (0 allocations: 0 bytes)
```

All these functions additionally accept transforming functions that require multiple arguments. To incorporate this possibility, it's necessary to call the multiple variables using `zip`, and referring to each variable through indexes. We illustrate this below, where the transforming function is `x .* y`.

```julia
x = rand(100); y = rand(100)

foo(x,y) = sum(a -> a[1] * a[2], zip(x,y))          #same output as sum(x .* y)
```

```
julia> @btime foo($x)
  29.127 ns (0 allocations: 0 bytes)
```

```julia
x = rand(100); y = rand(100)

foo(x,y) = prod(a -> a[1] * a[2], zip(x,y))         #same output as prod(x .* y)
```

```
julia> @btime foo($x)
  48.031 ns (0 allocations: 0 bytes)
```

```julia
x = rand(100); y = rand(100)

foo(x,y) = maximum(a -> a[1] * a[2], zip(x,y))      #same output as maximum(x .* y)
```

```
julia> @btime foo($x)
  172.580 ns (0 allocations: 0 bytes)
```

```julia
x = rand(100); y = rand(100)

foo(x,y) = minimum(a -> a[1] * a[2], zip(x,y))      #same output as minimum(x .* y)
```

```
julia> @btime foo($x)
  166.969 ns (0 allocations: 0 bytes)
```

THE "REDUCE" AND "MAPREDUCE" FUNCTIONS

Beyond the specific functions discussed, reductions can be applied to any operation that meets the requirements of a reduction. In Julia, this generality is provided through the functions `reduce` and `mapreduce`. The function `reduce` applies a binary operation directly to the elements of a collection, combining them into a single result. By contrast, `mapreduce` first transforms each element of the collection and then applies the reduction, thereby unifying the roles of `map` and `reduce` in a single step.

It's worth remarking that reductions involving `sum`, `prod`, `max`, and `min` should still be done via their dedicated functions. These specialized functions have been carefully optimized for their respective tasks and typically outperform the more general alternatives. Consequently, our primary use case of `reduce` and `mapreduce` is for other types of reductions not covered or when packages provide their own implementations for reductions. [2]

FUNCTION "REDUCE"

The function `reduce` uses the syntax `reduce(<function>,x)`, where `<function>` is a two-argument function. The following example demonstrates its use.

```julia
x      = rand(100)

foo(x) = reduce(+, x)            #same output as sum(x)
```

```
julia> @btime foo($x)
  6.168 ns (0 allocations: 0 bytes)
```

```julia
x      = rand(100)

foo(x) = reduce(*, x)            #same output as prod(x)
```

```
julia> @btime foo($x)
  6.176 ns (0 allocations: 0 bytes)
```

```julia
x      = rand(100)

foo(x) = reduce(max, x)          #same output as maximum(x)
```

```
julia> @btime foo($x)
  167.905 ns (0 allocations: 0 bytes)
```

```julia
x      = rand(100)

foo(x) = reduce(min, x)          #same output as minimum(x)
```

```
julia> @btime foo($x)
  167.440 ns (0 allocations: 0 bytes)
```

Note all the examples presented could've been implemented as we did previously, where we directly applied `sum`, `prod`, `maximum` and `minimum`.

FUNCTION "MAPREDUCE"

The function `mapreduce` combines the functions `map` and `reduce`: before applying the reduction, `mapreduce` transforms vectors via the function `map`. Recall that `map(foo,x)` transforms each element of the collection `x` by applying `foo` element-wise. Thus, `mapreduce(<transformation>, <reduction>,x)` first transforms `x`'s elements through `map`, and then applies a reduction to the resulting output.

To illustrate its use, we make use of a `log` transformation.

```julia
x      = rand(100)

foo(x) = mapreduce(log, +, x)        #same output as sum(log.(x))
```

```
julia> @btime foo($x)
  294.805 ns (0 allocations: 0 bytes)
```

```julia
x      = rand(100)

foo(x) = mapreduce(log, *, x)       #same output as prod(log.(x))
```

```
julia> @btime foo($x)
  294.618 ns (0 allocations: 0 bytes)
```

```julia
x      = rand(100)

foo(x) = mapreduce(log, max, x)     #same output as maximum(log.(x))
```

```
julia> @btime foo($x)
  579.808 ns (0 allocations: 0 bytes)
```

```julia
x      = rand(100)

foo(x) = mapreduce(log, min, x)     #same output as minimum(log.(x))
```

```
julia> @btime foo($x)
  577.505 ns (0 allocations: 0 bytes)
```

Just like with `reduce`, note that the examples could've been implemented directly as [we did previously](), through the functions `sum`, `prod`, `maximum`, and `minimum`.

`mapreduce` can also be used with anonymous functions and transformations requiring multiple arguments. Below, we illustrate both possibilities, whose implementations are the same as with `sum`, `prod`, `maximum`, and `minimum`.

```julia
x = rand(100); y = rand(100)

foo(x,y) = mapreduce(a -> a[1] * a[2], +, zip(x,y))       #same output as sum(x .* y)
```

```
julia> @btime foo($x)
  29.165 ns (0 allocations: 0 bytes)
```

```julia
x = rand(100); y = rand(100)

foo(x,y) = mapreduce(a -> a[1] * a[2], *, zip(x,y))       #same output as prod(x .* y)
```

```
julia> @btime foo($x)
  48.221 ns (0 allocations: 0 bytes)
```

```julia
x = rand(100); y = rand(100)

foo(x,y) = mapreduce(a -> a[1] * a[2], max, zip(x,y))     #same output as maximum(x .* y)
```

```
julia> @btime foo($x)
  175.634 ns (0 allocations: 0 bytes)
```

```julia
x = rand(100); y = rand(100)

foo(x,y) = mapreduce(a -> a[1] * a[2], min, zip(x,y))      #same output as minimum(x .* y)
```

```
julia> @btime foo($x)
  166.995 ns (0 allocations: 0 bytes)
```

REDUCE OR MAPREDUCE?

In principle, `reduce` could be considered as a special case of `mapreduce`, where the latter transforms `x` through the identity function `identity(x)`. Likewise, `mapreduce(<transformation>, <operator>,x)` produces the same result as `reduce(<operator>, map(<transformation>,x))`.

Nonetheless, `mapreduce` should be employed in those cases where the vector input must be transformed. The reason is that `mapreduce` avoids the internal memory allocations of the transformed vector. This is demonstrated below, where we compute `sum(2 .* x)` through a reduction.

```julia
x = rand(100)

foo(x) = mapreduce(a -> 2 * a, +, x)
```

```
julia> @btime foo($x)
  6.372 ns (0 allocations: 0 bytes)
```

```julia
x = rand(100)

foo(x) = reduce(+, map(a -> 2 * a, x))
```

```
julia> @btime foo($x)
  43.476 ns (1 allocation: 896 bytes)
```

FOOTNOTES

1. In the section Lazy Operations, we'll explore an alternative based on broadcasting that doesn't materialize intermediate results either.

2. For instance, the package `Folds` provides a parallelized version of both `map` and `mapreduce`, enabling the utilization of all available CPU cores. Its syntax is identical to Julia's built-in functions.