9f. Reductions

Martin Alfaro
PhD in Economics

INTRODUCTION

Reductions are a computational technique applied to **operations that take a collection as input and return a single element as output**. 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 process involves iteratively applying an operation to pairs of elements, accumulating the results at each step until the final output is obtained. A classic example of reduction 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. The process is demonstrated below.

```julia
x = rand(100)

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

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

```julia
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
```

```julia
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
```

The last tab implements the reduction via update operators. They're commonly applied in reductions, as they keep compact notation by transforming expressions such as `x = x + a` into `x += a`.

Regarding memory allocations, reduction are particularly convenient when a vector's elements need to be transformed prior to aggregating the result. By operating on scalars, **reductions sidestep the need to materialize intermediate outputs**, thus reducing allocations. For instance, if you have to compute `sum(log.(x))`, a reduction would avoid the allocations of the intermediate vector `log.(x)`.

IMPLEMENTING REDUCTIONS

At a formal level, reductions combine a sequence of values iteratively through a *binary operation*, collapsing into a single scalar. For a reduction to be valid, the binary operation must satisfy **two mathematical properties**:

- **Associativity**: the way in which operations are grouped doesn't 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`.

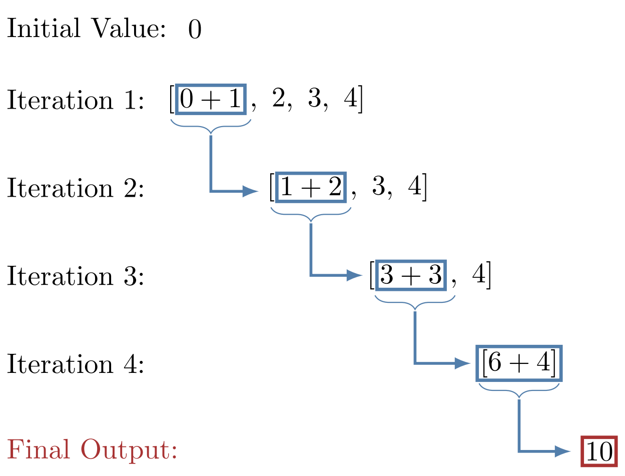
The identity element is crucial in implementation, as it provides the initial value of the iterative process. Specifically, it serves as the initialization value of the variable that accumulates the final result. Each operation has its own identity element, as summarized in the table below.

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

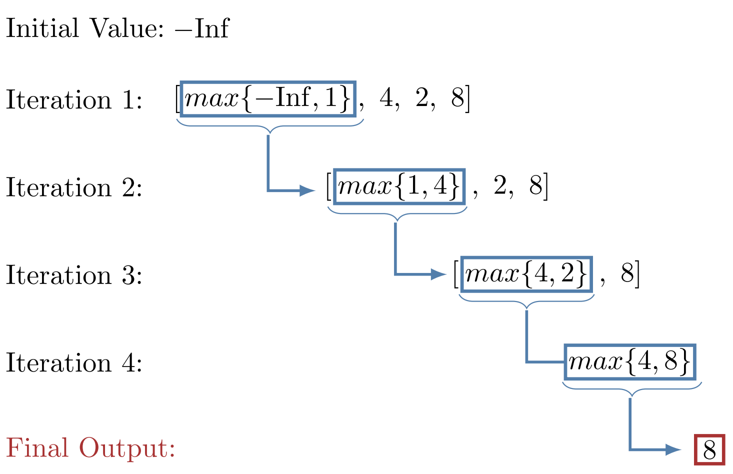
The implementation of reductions requires binary operations, which can be expressed using binary operators or two-argument functions. For instance, the symbol `+` can be employed in either form, making `output = output + x[i]` equivalent to `output = +(output, x[i])`. The possibility of functions expands the scope of reductions, enabling operations not directly supported by operators. For instance, the maximum value within a vector `x` can be found by the `max` function, where `max(a,b)` returns the maximum of the scalars `a` and `b`.

The figures below illustrate reductions implemented with a binary operator and with a two-argument function.

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



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


Initial Value: −Inf
Iteration 1: [max{−Inf, 1}, 4, 2, 8]
Iteration 2: [max{1, 4}, 2, 8]
Iteration 3: [max{4, 2}, 8]
Iteration 4: max{4, 8}
Final Output:
8

The following examples demonstrate how to formulate reductions using for-loops. We present `foo1` to show the desired outcome explicitly, while `foo2` provides the same result through a reduction.

```julia
x = rand(100)

foo1(x) = sum(x)

function foo2(x)
    output = 0.

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

    return output
end
```

```julia
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
```

AVOIDING MEMORY ALLOCATIONS THROUGH REDUCTIONS

One of the primary advantages of reductions is their ability to avoid memory allocation for intermediate results.

To illustrate this capability, consider the operation `sum(log.(x))` for a vector `x`. Its computation involves two steps: transforming `x` into `log.(x)`, and then summing the transformed elements. By default, broadcasting materializes its results, implying that a new vector is internally created to store the values of `log.(x)`. Consequently, the first step results in memory allocations.

In many cases like this one, however, only the scalar output matters, and the intermediate result is of no interest. In this context, computational strategies that obtain the final output while bypassing the allocation of `log.(x)` are preferred. Reductions make this possible by defining a scalar `output`, which is iteratively updated by summing the transformed values of `x`. This means each element of `x` is transformed by the logarithm, and the result is then immediately added to the accumulator. In this way, the storage of the intermediate vector `log.(x)` is entirely avoided. [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

So far, reductions with intermediate transformations have been implemented manually through explicit for-loops. While this approach makes the underlying mechanics transparent, it also introduces considerable verbosity.

To address this limitation, Julia provides several streamlined alternatives for expressing reductions in a more concise and idiomatic way. One such alternative is through specialized methods for common reduction functions, including `sum`, `prod`, `maximum`, and `minimum`. These methods accept a transforming function as their first argument, followed by the collection to be reduced. The general syntax is `foo(<transforming function>, x)`, where `foo` denotes the reduction function and `x` is the vector to be transformed and reduced.

The following examples illustrate this approach by applying a logarithmic transformation prior to the reduction.

```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)
```

```julia
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)
```

```julia
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)
```

These specialized methods are commonly applied using anonymous functions, as shown below.

```julia
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)
```

```julia
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)
```

```julia
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)
```

```julia
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)
```

The specialized methods also accept transforming functions with multiple arguments. In this case, the arguments must be combined by using `zip`, referring each argument within the transforming function through indices. This is illustrated below, with the transforming operation `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 cases discussed, reductions can be applied as long as the operation meets their requirements. To accommodate this generality, Julia provides 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 for sums, products, maximum, and minimum should still be implemented via their dedicated functions. Specifically, `sum`, `prod`, `maximum`, and `minimum` have been carefully optimized for their respective tasks, typically outperforming the general functions `reduce` and `mapreduce`. This is why `reduce` and `mapreduce` should be restricted to reductions not covered by specialized methods. [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)
```

```
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)
```

```
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)
```

```
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 through the functions `sum`, `prod`, `maximum`, and `minimum`, as we did previously.

Moreover, `mapreduce` can be used with anonymous functions and functions with multiple arguments. Below, we illustrate these possibilities.

```
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)
```

```
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)
```

```
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?

`mapreduce(<transformation>,<operator>,x)` produces the same result as `reduce(<operator>, map(<transformation>,x))`. Despite this, `mapreduce` is preferred for scenarios where the vector input must be transformed first. The reason is that `mapreduce` avoids the internal memory allocations of the transformed vector, while `map` doesn't. This aspect is demonstrated below, where `sum(2 .* x)` is computed 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 approach. This is based on broadcasting and also avoids materializing intermediate results.

[2] The functions `reduce` and `mapreduce` are also added to packages for implementing reductions in a specific way. 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.