9e. Reductions

Martin Alfaro
PhD in Economics

INTRODUCTION

Reductions are a computational technique for **operations taking a collection as input and returning 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 underlying technique 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. It arrives at the final result by applying the addition operator `+` to pairs of elements, iteratively updating the accumulated sum. This case is illustrated below.

```
x = rand(100)

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

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

```
x = rand(100)

function foo(x)
    output = 0.0

    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 = 0.0

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

    return output
end
```

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

The last tab implements the reduction via an update operator. These operators are frequently used in reductions because they streamline notation, rewriting expressions like `x = x + a` in the more compact form `x += a`.

The main benefit of reductions is that, by operating on scalars, **they don't create memory allocations**. This is particularly convenient when a vector must be transformed prior to aggregating the result. For instance, if you have to compute `sum(log.(x))`, a reduction would avoid the allocations of the intermediate vector `log.(x)`.

IMPLEMENTING REDUCTIONS

Technically, reductions iteratively apply a *binary operation* to pairs of values, ultimately producing a scalar result. For a reduction to be valid, the binary operation must satisfy **two mathematical properties**:

- **Associativity**: the grouping of operations doesn't affect the outcome. For example, scalar 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 scalar addition is 0 because $a + 0 = a$.

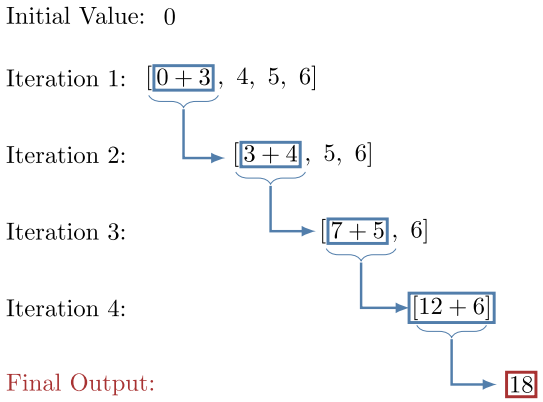
The identity element serves as the initial value of the accumulator variable. [1] Each operation has its own identity element, as shown in the table below.

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

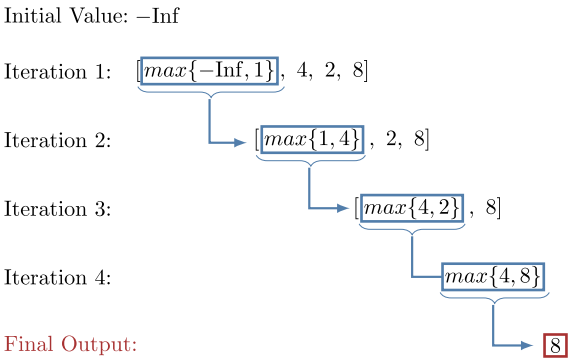
Remarkably, binary operations can be expressed either as binary operators or as two-argument functions. For instance, the symbol `+` can be employed in either form, making `output = output + x[i]` equivalent to `output = +(output, x[i])`. The possibility of using functions for reductions expands their scope. For instance, it enables the use of the `max` function to find the maximum value in a vector `x`, where `max(a,b)` returns the larger of the two scalars `a` and `b`.

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

REDUCTION via OPERATOR: sum of [3,4,5,6]



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



Manual implementations of reductions are done via for-loops. To illustrate its formulation, below we present `foo1` to identify the desired outcome, with `foo2` providing the same result through a reduction.

```
x       = rand(100)

foo1(x) = sum(x)

function foo2(x)
    output = 0.0

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

    return output
end
```

```
x       = rand(100)

foo1(x) = prod(x)

function foo2(x)
    output = 1.0

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

    return output
end
```

```
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
```

```
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 avoidance of memory allocation, in particular for intermediate results.

To illustrate, 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 the internal creation of a new vector to store the values

of `log.(x)`. Consequently, the step results in memory allocations.

In many cases, however, only the scalar output matters, with the intermediate result being 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 that 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. [2]

```
x       = rand(100)

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

function foo2(x)
    output = 0.0

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

    return output
end
```

```
julia> @btime foo1($x)
  437.594 ns (2 allocations: 928 bytes)
julia> @btime foo2($x)
  427.652 ns (0 allocations: 0 bytes)
```

```
x       = rand(100)

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

function foo2(x)
    output = 1.0

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

    return output
end
```

```
julia> @btime foo1($x)
  436.706 ns (2 allocations: 928 bytes)
julia> @btime foo2($x)
  427.406 ns (0 allocations: 0 bytes)
```

```
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($x)
  673.357 ns (2 allocations: 928 bytes)
julia> @btime foo2($x)
  538.000 ns (0 allocations: 0 bytes)
```

```
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($x)
  680.628 ns (2 allocations: 928 bytes)
julia> @btime foo2($x)
  526.982 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 inconvenience, Julia provides several streamlined alternatives for expressing reductions. One such alternative is through specialized methods of common reduction functions, including `sum`, `prod`, `maximum`, and `minimum`. These function 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` is 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.

```
x      = rand(100)

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

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

```
x      = rand(100)

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

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

```
x      = rand(100)

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

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

```
x      = rand(100)

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

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

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

```
x      = rand(100)

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

```
julia> @btime foo($x)
  9.750 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)
  12.222 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)
  237.689 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)
  235.365 ns (0 allocations: 0 bytes)
```

The methods also accept transforming functions with multiple arguments. In this case, the arguments must be paired using `zip`, with indices corresponding to each argument within the transforming function. This is illustrated below, with the transforming operation `x .* y`.

```
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)
  40.502 ns (0 allocations: 0 bytes)
```

```
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)
  64.310 ns (0 allocations: 0 bytes)
```

```
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)
  224.798 ns (0 allocations: 0 bytes)
```

```
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)
  231.638 ns (0 allocations: 0 bytes)
```

THE "REDUCE" AND "MAPREDUCE" FUNCTIONS

Beyond the specific cases discussed, reductions are applicable whenever a binary operation meets the necessary conditions for their implementation. To accommodate this generality, Julia provides the functions `reduce` and `mapreduce`.

The `reduce` function applies a binary operation directly to the elements of a collection, combining them into a single result. By contrast, `mapreduce` first transforms each element before applying the reduction.

It's worth remarking that reductions for sums, products, maximum, and minimum should still be implemented via their dedicated functions. This is because the methods in `sum`, `prod`, `maximum`, and `minimum` have been carefully optimized for their respective tasks, typically outperforming the general functions `reduce` and `mapreduce`. [3]

FUNCTION "REDUCE"

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

```
x      = rand(100)

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

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

```
x      = rand(100)

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

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

```
x      = rand(100)

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

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

```
x      = rand(100)

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

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

Note that all the examples above could've been implemented as before, where we directly applied `sum`, `prod`, `maximum` and `minimum`.

FUNCTION "MAPREDUCE"

The `mapreduce` function integrates `map` and `reduce` into a unified operation (hence its name). Thus, it applies a transformation via the function `map` before doing the reduction. Its syntax is `mapreduce(<transformation>,<reduction>,x)`. To illustrate its use, we make use of a `log` transformation.

```
x      = rand(100)

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

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

```
x      = rand(100)

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

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

```
x      = rand(100)

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

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

```
x      = rand(100)

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

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

Note that, again, the examples could've been implemented directly through the functions `sum`, `prod`, `maximum`, and `minimum` as we did previously.

`mapreduce` can also 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)
  40.503 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)
  64.309 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)
  224.946 ns (0 allocations: 0 bytes)
```

```
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)
  231.547 ns (0 allocations: 0 bytes)
```

REDUCE OR MAPREDUCE?

`mapreduce(<transformation>,<operator>,x)` yields the same result as `reduce(<operator>,map(<transformation>,x))`. Despite this, `mapreduce` is preferred if the vector input must be transformed beforehand. 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.

```
x      = rand(100)

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

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

```
x      = rand(100)

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

```
julia> @btime foo($x)
  43.327 ns (2 allocations: 928 bytes)
```

FOOTNOTES

[1.] Strictly speaking, an identity element isn't always required. In many cases, no natural identity exists, yet the first element of the collection can serve as the initial value. For simplicity, however, we assume the existence of an identity element as a requirement.

[2.] In the section Lazy Operations, we'll explore an alternative approach. This is based on broadcasting and also avoids materializing intermediate results.

[3.] The functions `reduce` and `mapreduce` are also convenient for packages that implement specialized versions of reductions. By simply defining these two functions, the package can then cover all possible reductions. For instance, the package `Folds` provides a multithreaded version of reductions via `map` and `mapreduce`.