9f. Lazy Operations

Martin Alfaro
PhD in Economics

INTRODUCTION

Computational approaches can be broadly classified into "lazy" and "eager" paradigms. **Eager operations** are executed immediately upon definition, providing instant access to the results. This is the default behavior in most programming contexts, including the majority of operations on this website up to this point.

In contrast, **lazy operations** define the code to be executed, deferring computation until the results are actually needed. The approach is particularly valuable for operations involving heavy intermediate computations, as lazy evaluation can **sidestep unnecessary memory allocations**: by fusing operations, it becomes possible to perform intermediate calculations on the fly and feed them directly into the final calculation.

This section provides various implementations for lazy computations. The first approach presented is based on the so-called **generators**, which are the lazy analogue of array comprehensions. After this, we'll introduce several functions from the package **Iterators**, which provides lazy implementations of functions such as `map` and `filter`.

GENERATORS

Array comprehensions offer a convenient technique for creating vectors, using a similar syntax to for-loops to define their elements. These elements are computed and stored right away, making array comprehensions an eager operation. For their part, **generators** represent **the lazy counterpart of array comprehensions**, deferring the creation of elements until they're actually needed.

In terms of syntax, generators are identical to array comprehensions, with the sole difference that they're enclosed in parentheses `( )` instead of square brackets `[ ]`. Just like array comprehensions, generators also retain the ability to add conditions and simultaneously iterate over multiple collections.

```
x = [a for a in 1:10]

y = [a for a in 1:10 if a > 5]
```

```
julia> x
10-element Vector{Int64}:
  1
  2
  ⋮
  9
 10
julia> y
5-element Vector{Int64}:
  6
  7
  8
  9
 10
```

```
x = (a for a in 1:10)

y = (a for a in 1:10 if a > 5)
```

```
julia> x
Base.Generator{UnitRange{Int64}, typeof(identity)}(identity, 1:10)
```

The examples show that array comprehensions compute all their elements at the moment of defining the vector, giving immediate access to them. In contrast, generators formally define an object with type `Base.Generator`, where operations are described, but no output is materialized.

This characteristic of generators makes them particularly useful for computing reductions with transformed values. By producing values on-demand and fusing them with the reduction function, generators avoid the materialization of temporary vectors, thus reducing memory allocations.

To illustrate the performance benefits this entails, let's compute the sum of all elements in a vector `y`. In particular, `y` is obtained by doubling each element of a vector `x`. One way to compute this operation is by first creating the vector `y` and then summing all its elements. Alternatively, we can describe the transformation through a generator, which bypasses the storage of the intermediate output `y` and instead feeds the transformation directly into the `sum` function. This allows the compiler to perform the addition as a cumulative operation on scalars, thereby reducing memory usage.

```
x = rand(100)

function foo(x)
    y = [a * 2 for a in x]                  # 1 allocation (same as y = x .* 2)

    sum(y)
end
```

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

```
x = rand(100)

function foo(x)
    y = (a * 2 for a in x)                  # 0 allocations

    sum(y)
end
```

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

```
x = rand(100)


foo(x) = sum(a * 2 for a in x)              # 0 allocations
```

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

The last tab shows that generators can be incorporated directly as a function argument, resulting in a compact syntax. Remarkably, this syntax is applicable to *any* function that accepts a collection as its input.

ITERATORS

Iterators are formally defined as lazy objects that create sequential values on demand, rather than storing them all in memory upfront. Throughout the website, we've already encountered numerous scenarios involving iterators. A typical example of an iterator is a range, such as `1:length(x)`, which defines a sequence of numbers to be generated on the fly. Their lazy evaluation explains why the function `collect` is needed when we want to materialize the entire sequence into a vector. Without `collect`, iterators merely describe the numbers to be created, without actually creating and storing them in memory.

Beyond simple ranges, we've also covered other types of iterators that offer more specialized functionality. They included `eachindex` for accessing array indices, `enumerate` for pairing elements with their positions, and `zip` for combining multiple sequences.

The lazy nature of iterators makes them particularly efficient in for-loops: by generating each value as the for-loop progresses, we eliminate unnecessary memory allocations that would arise from materializing the list being iterated over.

```
x = 1:10
```

```
julia> x
1:10
```

```
x = collect(1:10)
```

```
julia> x
10-element Vector{Int64}:
  1
  2
  ⋮
  9
 10
```

The built-in package `Iterators`, which is automatically "imported" in every Julia session, provides multiple functions for generating lazy sequences. Additionally, it offers lazy counterparts of various functions such as `filter` and `map`, which can be accessed as `Iterators.filter` and `Iterators.map`. [1]

The following example demonstrates the use of these functions to avoid memory allocations of intermediate computations.

```
x = collect(1:100)

function foo(x)
    y = filter(a -> a > 50, x)              # 1 allocation

    sum(y)
end
```

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

```
x = collect(1:100)

function foo(x)
    y = Iterators.filter(a -> a > 50, x)    # 0 allocations

    sum(y)
end
```

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

```
x = rand(100)

function foo(x)
    y = map(a -> a * 2, x)                  # 1 allocation

    sum(y)
end
```

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

```
x = rand(100)

function foo(x)
    y = Iterators.map(a -> a * 2, x)        # 0 allocations

    sum(y)
end
```

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

FOOTNOTES

[1] The `IterTools` package further extends the functionality of `Iterators`, offering even more tools for working with lazy sequences.