RUBY CLOSURES

Scopes, Bindings, and Lifetimes

Thomas Countz

Ruby Closures

Scopes, Bindings, and Lifetimes

by Thomas Countz

Version 0.0.1

TABLE OF CONTENTS

WHAT IS A CLOSURE?

A closure is a function that captures and remains bound to its surrounding *lexical scope*, even when executed in a different context. This concept, borrowed from functional programming and lambda calculus, gives Ruby the ability to define higher-order functions, encapsulate and share state, and create powerful abstraction for libraries and frameworks.

Lexical Scope
Lexical scope refers to the accessibility of variables based on where they're defined in the source code. *Lexical* scope, specifically, is determined at parse/compile-time (as opposed to *dynamic scope* which changes during runtime), and is therefore also called *static scope*. This means — before execuing any code — we can determine how Ruby will manage certain variables in memory based on how/where they're defined.

What We're Building

To understand the power of closures, let's build a Ruby gem that provides an API for callbacks, similar to `before_`, `after_`, and `around_` hooks in Rails. We'll create a naive solution without using closures first and evaluate it quantitatively on what we can observe and qualitatively in terms of things like extensibility, readability, etc. Then, we'll refactor the code to use closures in order to see what there is to gain by using them.

Our Simple Gem

```ruby
def greet(name)
  -> { puts "Hello, #{name}!" }
end

hello_world = greet("World")

# Later...
hello_world.call  # => Hello, World!
```

In the example above, the `greet` method returns a closure in the form of a *lambda*. In Ruby, lambdas, procs, and blocks are all types of closures.

When we call `greet("World")`, it creates a closure (lambda) that wraps a piece of code that can be executed later by calling `#call`. Not only that, that piece of code captured a reference to the `name` variable, which means the closure to make use of it whenever its executed later. All this, despite the scope of the `#greet` method having already ended.

Once a method exits/returns in Ruby, its local variables — including its parameters — are marked as *unreachable* and are subject to garbage collection. This is because they are no longer needed. But if used in a closure, they're kept alive because there remains a reference to them.

As a quick example:

```ruby
def greet(name)
  puts "Hello, #{name}!"
end

greet("World")
```

In this example, the `name` variable is only accessible within the `#greet` method. Once the method exits, the variable is no longer accessible. If we take a look at which variables are accessible in the current scope, we'll see that `name` is not longer available:

And this makes sense. Why would we (i.e. Ruby) use up memory to keep track of a variable that will never be used again?

Lambdas
In programming, a lambda is building block of functional-style programming. It's a way to define an *anonymous functions*, which can be passed around as an argument or stored in variables. In Ruby, even if you haven't created a lambda explicitly, you've likely used them in Ruby when passing blocks to methods like `each` or `map`.

Lambdas vs. Procs
In Ruby, the `Kernel#lambda` method (and the Lambda Proc Literal, →) returns a `Proc`, just like `Proc.new`. However they have subtle differences in how they handle arguments and return values. A `lambda` enforces strict arity (argument count) and returns control to the caller, while procs are more lenient and return control to the enclosing scope. Throughout this blog post, I will be using Lambdas to demonstrate closures. See the Lambda and non-lambda semantics section of the Ruby documentation for more information.[1]

The Implementation: Behind the Scenes

Closures in Ruby involve several layers of complexity. Their implementation relies on **lexical scope analysis**, **binding environments**, and **lifetime management**. Each layer plays a critical role in making closures both powerful and practical.

1. Lexical Scope Analysis: Knowing What to Capture

Ruby determines what variables a closure needs to capture through **lexical scope analysis**, which happens during parsing. Simply put, Ruby identifies which variables are accessible based on the code's structure and where the closure is defined.

```ruby
def create_counter
  count = 0
  -> { count += 1 }
end
```

When Ruby encounters the lambda, it analyzes its surrounding context and decides that `count` must be captured because it's defined outside the closure but referenced within it.

We can visualize this process using Ruby's `Prism` parser:

```ruby
require 'prism'

code = <<~RUBY
  def create_counter
    count = 0
```

[1]https://docs.ruby-lang.org/en/3.4/Proc.html#class-Proc-label-Lambda+and+non-lambda+semantics

```
    -> { count += 1 }
  end
RUBY

puts Prism.parse(code).value
```

The parser identifies variables that need to be captured (count in this case) and marks them for inclusion in the closure's binding environment by setting the depth to 1:

```
@ LambdaNode (location: (3,2)-(3,19))
├── flags: newline
├── locals: []
├── operator_loc: (3,2)-(3,4) = "->"
├── opening_loc: (3,5)-(3,6) = "{"
├── closing_loc: (3,18)-(3,19) = "}"
├── parameters: ∅
└── body:
    @ StatementsNode (location: (3,7)-(3,17))
    ├── flags: ∅
    └── body: (length: 1)
        └── @ LocalVariableOperatorWriteNode (location: (3,7)-(3,17))
            ├── flags: newline
            ├── name_loc: (3,7)-(3,12) = "count"
            ├── binary_operator_loc: (3,13)-(3,15) = "+="
            ├── value:
            │   @ IntegerNode (location: (3,16)-(3,17))
            │   ├── flags: static_literal, decimal
            │   └── value: 1
            ├── name: :count
            ├── binary_operator: :+
            └── depth: 1  # Indicates `count` comes from one scope above
```

The `depth: 1` indicates that the `count` variable is from an outer scope (the `create_counter` method's scope). This is how Prism tracks variables that need to be captured in the closure's binding. A depth of 1 means it's looking one scope level up from the current scope.

Depth
Depth refers to the number of visible scopes that Prism has to go up to find the declaration of a local variable. Note that this follows the same scoping rules as Ruby, so a local variable is only visible in the scope it is declared in and in blocks nested in that scope.[2]

2. The Binding Environment: A Snapshot of Context

Closures don't just copy variables — they maintain *live references* to them via a binding environment as pointers to their address in memory. This allows closures to both read and modify captured variables.

```
count = 0
increment = -> { count += 1 }
get_count = -> { count }

increment.call.object_id  # => 60
count.object_id # => 60
get_count.call.object_id  # => 60
```

[2] https://github.com/ruby/prism/blob/main/docs/local_variable_depth.md

Both `increment` and `get_count` share the same `count` variable through the binding environment. Ruby ensures these references remain valid, even after the original scope exits.

You can introspect the binding environment using `binding`:

```
def binding_example
  local_var = "You can trust me, I'm a local!"
  -> { binding }
end

closure_binding = binding_example.call
puts closure_binding.local_variables  # ⇒ [:local_var]
puts closure_binding.eval("local_var")  # ⇒ "You can trust me, I'm a local!"
```

The binding environment, accessible through `binding`, provides a snapshot of the closure's context, including local variables and methods. Using it here, we can see that the closure retains access to the `local_var` variable, even though the scope in which it was defined (the `binding_example` method) has exited.

This mechanism creates the variable binding behavior as if we passed the `local_var` variable into the lambda function, i.e., the lambda function has local access the variable as if it were passed in as an argument.

```
def binding_example
  ->(local_var) { binding }
end

closure_binding = binding_example.call("You can trust me, I'm a local!")
puts closure_binding.local_variables  # ⇒ [:local_var]
puts closure_binding.eval("local_var")  # ⇒ "You can trust me, I'm a local!"
```

3. Lifetime Management: Preventing Premature Cleanup

The implication of closures retaining access to variables is that they can keep those variables alive longer than expected. This means that while local variables in a method are typically marked for garbage collection when the method exits, variables captured by a closure remain alive as long as the closure itself is accessible.

Example

```
def create_rememberer
  data = "important info"
  -> { data }  # data lives as long as this closure
end

GC.start  # Trigger garbage collection

puts create_rememberer.call
```

Ruby's GC won't clean up `data` because the closure still references it. You can confirm this behavior with `ObjectSpace`:

```
require 'objspace'

closure = create_rememberer
puts ObjectSpace.reachable_objects_from(closure)  # Shows `data` is retained
```

Practical Implications of Ruby's Closure Design

Shared State and Thread Safety

Closures with shared state can lead to subtle bugs in concurrent programs:

```ruby
def create_shared_counter
  count = 0
  increment = -> { count += 1 }
  get_count = -> { count }
  [increment, get_count]
end

increment, get_count = create_shared_counter
increment.call
puts get_count.call  # => 1
```

In multi-threaded environments, this shared state requires synchronization to avoid race conditions.

Memory Management and Efficiency

Because closures retain captured variables, they can unintentionally increase memory usage if they hold references to large objects:

```ruby
# Inefficient
def inefficient
  large_data = "x" * 1_000_000
  -> { large_data.length }
end

# Better
def efficient
  length = ("x" * 1_000_000).length
  -> { length }
end
```

Optimizing what closures capture can reduce memory overhead.

Closures enable elegant patterns like memoization and encapsulation:

```ruby
# Memoization
def memoize
  cache = {}
  ->(key) { cache[key] ||= yield(key) }
end

fib = memoize do |n|
  n < 2 ? n : fib.call(n - 1) + fib.call(n - 2)
end
```

Closures make it easy to encapsulate private state:

```ruby
def create_secure_counter
  count = 0
  secret = Object.new

  increment = ->(token) {
    raise "Invalid token" unless token.equal?(secret)
    count += 1
  }

  [increment, secret]
end
```

Performance Trade-offs

Closures involve more overhead than regular methods due to:

1. Lexical scope analysis.
2. Binding environment creation.
3. Variable indirection mechanisms.
4. Heap allocation for captured variables.

Despite this, closures' expressiveness often justifies their cost. The key is understanding when and how to use them effectively.

Conclusion

Ruby's closure implementation strikes a balance between power and practicality. By understanding the underlying mechanisms, you can:

- Write more efficient code.
- Debug closure-related issues with ease.
- Leverage closures for elegant, reusable patterns.

Remember: closures are more than just functions — they're functions **with memory**. Use them wisely, and they'll become one of your most powerful tools. s phenomenon from standard debt, we introduce the metric of **Technical Lag**.

> **Definition**
> Technical Lag refers to the temporal delta between the upstream release of a dependency and the currently deployed version within the production environment.

The accumulation of lag is not linear; it is compounding. As the delta increases, the probability of a breaking change (ΔP) approaches 1.0.

Uncertainty Principle

The uncertainty principle in dependency management states that as the technical lag (L) increases, the predictability of successful upgrades decreases exponentially. This relationship can be modeled as:

$$P(\text{success}) = e^{-k*L}$$

Where k is a constant representing the sensitivity of the dependency ecosystem.

See Dependency Management under Uncertainty for a formal treatment.

Code Example in Ruby

This is an example of a Gemfile that may incur technical lag if not regularly updated:

```
# Gemfile
gem 'rails', '~> 6.0.0'
gem 'activesupport', '~> 6.0.0'
```

As new versions of `rails` and `activesupport` are released, the lag increases, leading to potential security vulnerabilities and incompatibilities.

METRICS & MEASUREMENT

To quantify the tax, we utilize two primary scalar values. These values allow for the objective comparison of project health across the fleet.

Version Sequence Distance (VSD)

VSD represents the discrete count of releases between the current version (V_{curr}) and the ideal version (V_{ideal}).

$$\text{VSD} = \sum_{i=\text{curr}}^{\text{ideal}} \text{release}_i$$

Libyear

Libyear represents the chronological time passed between the release date of the current version and the release date of the ideal version.

Critical Warning
Failure to address lag in `rails` or `activesupport` gems results in a cascading dependency lock, rendering minor updates impossible without major refactoring.

1. Identify the constraints preventing the upgrade.
2. Isolate the “Upper Bound” dependency.
3. Execute the upgrade sequence.