8

HOW RUBY BORROWED A DECADES-OLD IDEA FROM LISP

Blocks are one of the most commonly used and powerful features of Ruby because they allow you to pass a code snippet to `Enumerable` methods, such as `each`, `detect`, or `inject`. Using the `yield` keyword, you can also write your own custom iterators or functions that call blocks for other reasons. Ruby code containing blocks is often more succinct, elegant, and expressive than equivalent code in older languages, such as C.

But don't jump to the conclusion that blocks are a new idea! In fact, blocks are not new to Ruby at all. The computer science concept behind blocks, called *closures*, was first invented by Peter J. Landin in 1964, a few years after the original version of Lisp was created by John McCarthy in 1958. Closures were later adopted by Lisp, or—more precisely—a dialect of Lisp called *Scheme*, which was invented by Gerald Sussman and Guy Steele in 1975. Sussman and Steele's use of closures in Scheme brought the idea to many programmers for the first time.

But what does the word *closure* actually mean in this context? In other words, exactly what are Ruby blocks? Are they just the snippet of Ruby code that appears between the `do` and `end` keywords? In this chapter I'll review how Ruby implements blocks internally and demonstrate how they meet the definition of *closure* used by Sussman and Steele back in 1975. I'll also show how blocks, lambdas, and procs are all different ways of looking at closures.

ROADMAP

Blocks: Closures in Ruby

Internally, Ruby represents each block using a C structure called `rb_block_t`, shown in Figure 8-1. By learning what Ruby stores in `rb_block_t`, we can find out exactly what a block is.

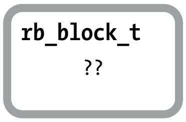


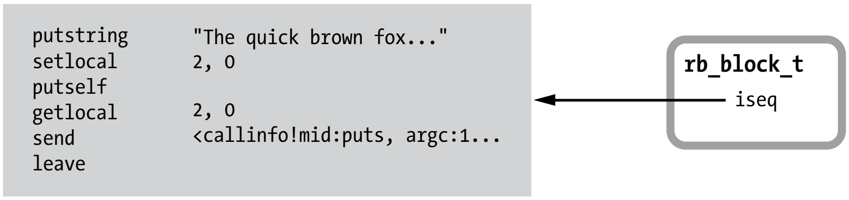
Figure 8-1: What's inside the `rb_block_t` C structure?

As we did in Chapter 5 with the RClass structure, let's deduce the contents of the rb_block_t structure based on what we know blocks can do in Ruby. We'll begin with the most obvious attribute of blocks. We know that each block must consist of a piece of Ruby code, or internally a set of compiled YARV bytecode instructions. For example, suppose we call a method and pass a block as a parameter, as shown in Listing 8-1.

```
10.times do
  str = "The quick brown fox jumps over the lazy dog."
  puts str
end
```

Listing 8-1: Superficially, a block is just a snippet of Ruby code.

When executing the 10.times call, Ruby needs to know what code to iterate over. Therefore, the rb_block_t structure must contain a pointer to that code, as shown in Figure 8-2.



```
putstring      "The quick brown fox..."
setlocal       2, 0
putself
getlocal       2, 0
send           <callinfo!mid:puts, argc:1...
leave
```

Figure 8-2: The rb_block_t structure contains a pointer to a snippet of YARV instructions.

The value iseq is a pointer to the YARV instructions for the Ruby code in the block.

Another obvious but often overlooked behavior of blocks is that they can access variables in the surrounding or parent Ruby scope, as shown in Listing 8-2.

```
❶ str = "The quick brown fox"
❷ 10.times do
❸   str2 = "jumps over the lazy dog."
❹   puts "#{str} #{str2}"
  end
```

Listing 8-2: The code inside the block accesses the variable str from the surrounding code.

Here the puts function call at ❹ refers equally to the str2 variable inside the block and the str variable defined in the surrounding code at ❶. Obviously blocks can access values from the code surrounding them. This ability is one of the things that makes blocks useful.

Blocks have in some sense a dual personality. On the one hand, they behave like separate methods: You can call them and pass them arguments just as you would any method. On the other hand, they're part of the surrounding function or method.

Stepping Through How Ruby Calls a Block

How does this work internally? Does Ruby implement blocks as separate methods or as part of the surrounding method? Let's step through Listing 8-2 to see what happens inside Ruby when you call a block.

When Ruby executes the first line of code from Listing 8-2 at ❶, `str = "The quick brown fox"`, YARV stores the local variable `str` on its internal stack. YARV tracks the location of `str` using the `EP`, or environment pointer, located in the current `rb_control_frame_t` structure, as shown in Figure 8-3.[1]

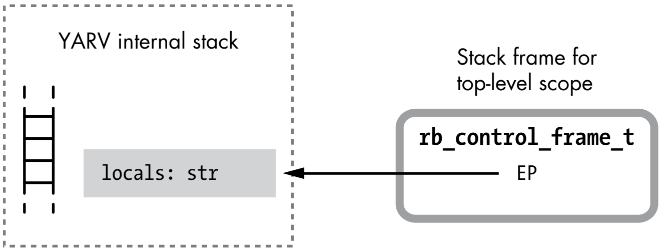


Figure 8-3: Ruby saves the local variable `str` on the stack.

Next, Ruby reaches the `10.times do` call at ❷ in Listing 8-2. Before executing the actual iteration—that is, before calling the `times` method—Ruby creates and initializes a new `rb_block_t` structure to represent the block. Ruby needs to create the block structure now because the block is really just another argument to the `times` method. Figure 8-4 shows this new `rb_block_t` structure.

When creating the new block structure, Ruby copies the current value of the `EP` into the new block. In other words, Ruby saves the location of the current stack frame in the new block.

1. If the outer code was located inside a function or method, then the `EP` would point to the stack frame as shown. But if the outer code was located in the top-level scope of your Ruby program, then Ruby would use dynamic access to save the variable in the `TOPLEVEL_BINDING` environment instead. Regardless, the `EP` will always indicate the location of the `str` variable.

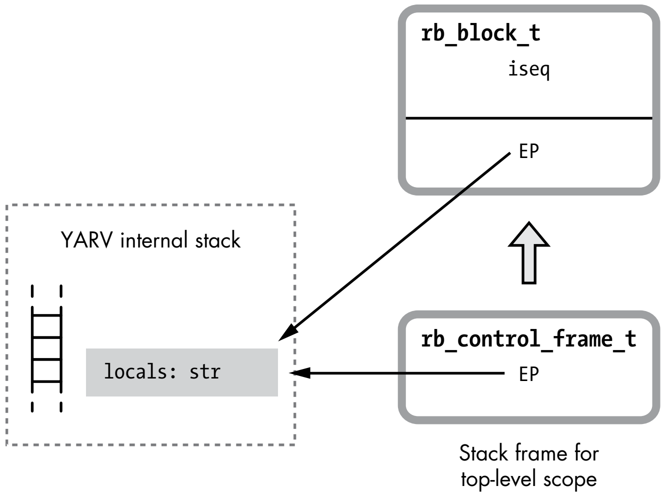


Figure 8-4: Ruby creates a new `rb_block_t` structure before calling the method and passing the block to it.

Next, Ruby calls the `times` method on the object `10`, an instance of the `Fixnum` class. While doing this, YARV creates a new frame on its internal stack. Now we have two stack frames: above, a new stack frame for the `Fixnum.times` method, and below, the original stack frame used by the top-level function (see Figure 8-5).

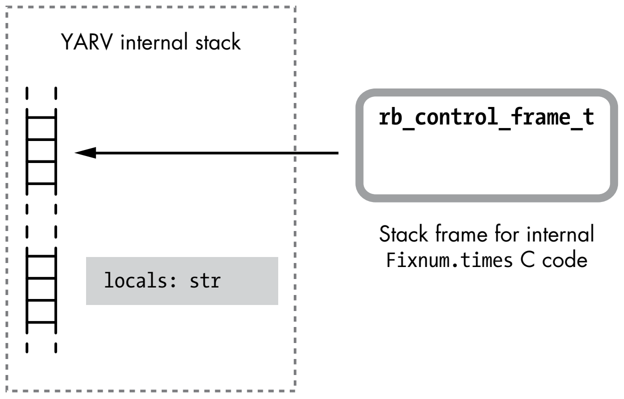


Figure 8-5: Ruby creates a new stack frame when it executes the `10.times` call.

Ruby implements the `times` method internally using its own C code. Although this is a built-in method, Ruby implements it just as you probably would. Ruby starts to iterate over the numbers 0, 1, 2, and so on, up to 9, and then it calls `yield`, calling the block once for each of these integers. Finally, the code that implements `yield` internally calls the block each time it moves through the loop, pushing a third frame onto the top of the stack for the code inside the block to use. Figure 8-6 shows this third stack frame.

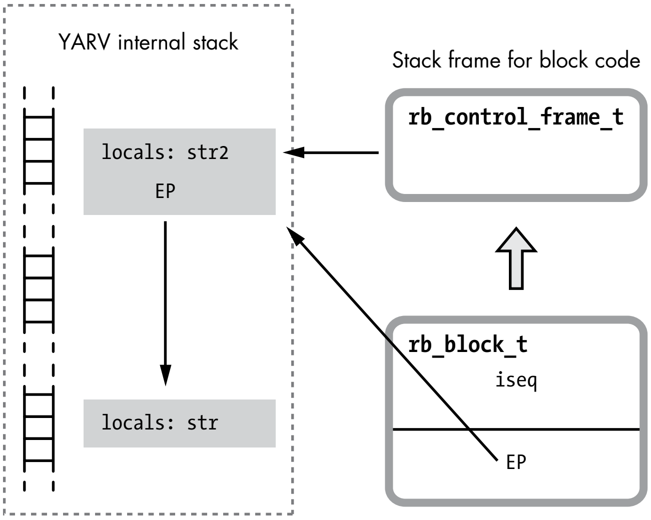


Figure 8-6: Ruby creates a third stack frame when the `10.times` method yields to the block.

On the left side of the figure, we now have three stack frames:

- On the top is the new stack frame for the block, containing the `str2` variable defined at ❸ in Listing 8-2.
- In the middle is the stack frame used by the internal C code that implements the `Fixnum#times` method.
- And at the bottom is the original function's stack frame, containing the `str` variable defined at ❶ in Listing 8-2.

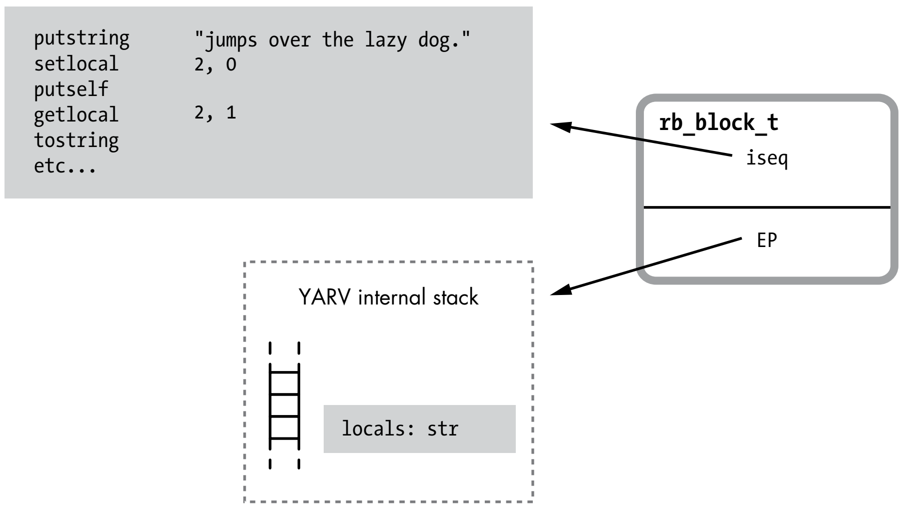
While creating the new stack frame, Ruby's internal `yield` code copies the `EP` from the block into the new stack frame. Now the code inside the block can access both its local variables, directly via the `rb_control_frame_t` structure, and the variables from the parent scope, indirectly via the `EP` pointer using dynamic variable access. Specifically, this allows the `puts` statement at ❹ in Listing 8-2 to access the `str2` variable from the parent scope.

Borrowing an Idea from 1975

So far we've seen that Ruby's `rb_block_t` structure contains two important values:

- A pointer to a snippet of YARV code instructions—the `iseq` pointer
- A pointer to a location on YARV's internal stack, the location that was at the top of the stack when the block was created—the `EP` pointer

Figure 8-7 shows these two values in the `rb_block_t` structure.



```
putstring      "jumps over the lazy dog."
setlocal       2, 0
putself
getlocal       2, 1
tostring
etc...
```

Figure 8-7: So far we've seen that Ruby blocks contain a pointer to a YARV instruction snippet and a location on the YARV stack.

The IBM 704, above, was the first computer to run Lisp, in the early 1960s. (Credit: NASA)

We also saw that Ruby uses the `EP` when a block accesses values from the surrounding code. At first, this seems like a very technical, unimportant detail. This is obviously a behavior we expect Ruby blocks to exhibit, and the `EP` seems to be a minor, uninteresting part of Ruby's internal implementation of blocks. Or is it?

The `EP` is actually a profoundly important part of Ruby internals. It's the basis for Ruby's implementation of *closures*, the computer science concept introduced in Lisp long before Ruby was created in the 1990s. Here's how Sussman and Steele defined the term *closure* in 1975:

> In order to solve this problem we introduce the notion of a closure [11, 14] which is a data structure containing a lambda expression, and an environment to be used when that lambda expression is applied to arguments.[2]

They define a closure to be the combination of the following:

- A "lambda expression"—that is, a function that takes a set of arguments
- An environment to be used when calling that lambda or function

2. Gerald J. Sussman and Guy L. Steele, Jr., "Scheme: An Interpreter for Extended Lambda Calculus" (MIT Artificial Intelligence Laboratory, AI Memo No. 349, December 1975).

Let's take another look at the internal `rb_block_t` structure, repeated for convenience in Figure 8-8.

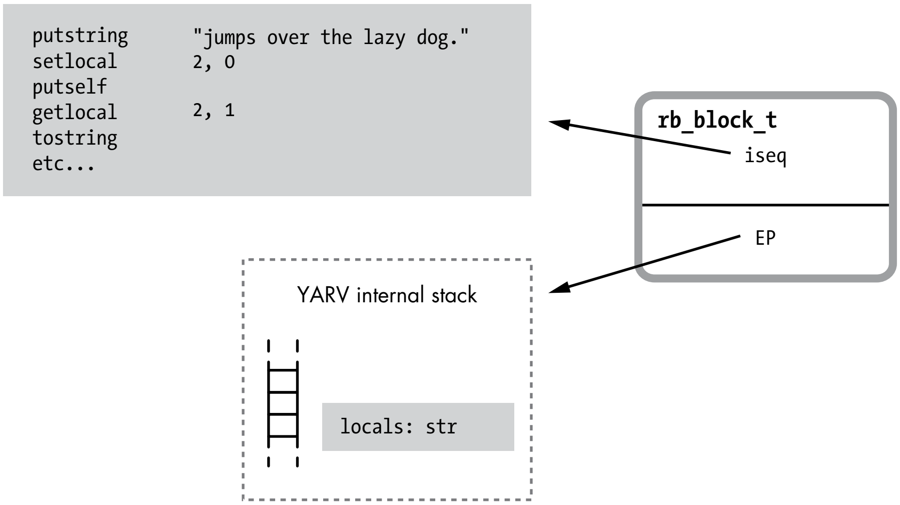


Figure 8-8: Blocks are the combination of a function and the environment to use when calling that function.

This structure meets Sussman and Steele's definition of a closure:

- `iseq` is a pointer to a lambda expression—a function or code snippet.
- `EP` is a pointer to the environment to be used when calling that lambda or function—that is, a pointer to the surrounding stack frame.

Following this train of thought, we can see that blocks are Ruby's implementation of closures. Ironically, blocks—one of the features that makes Ruby so elegant and modern—are based on research and work done at least 20 years before the birth of Ruby!

THE RB_BLOCK_T AND RB_CONTROL_FRAME_T STRUCTURES

In Ruby 1.9 and later, you'll find the definition of the `rb_block_t` structure in the *vm_core.h* file, as shown in Listing 8-3.

```
  typedef struct rb_block_struct {
❶     VALUE self;
❷     VALUE klass;
❸     VALUE *ep;
❹     rb_iseq_t *iseq;
❺     VALUE proc;
  } rb_block_t;
```

Listing 8-3: The definition of `rb_block_t` from vm_core.h

You can see the iseq ❹ and ep ❸ values described above, along with a few other values:

- self ❶: The value the self pointer had when the block was first referred to is also an important part of the closure's environment. Ruby executes block code inside the same object context that the code had outside the block.
- klass ❷: Along with self, Ruby also keeps track of the class of the current object using this pointer.
- proc ❺: Ruby uses this value when it creates a proc object from a block. As we'll see in the next section, procs and blocks are closely related.

Right above the definition of rb_block_t in *vm_core.h*, we see the definition of the rb_control_frame_t structure, as shown in Listing 8-4.

```
typedef struct rb_control_frame_struct {
    VALUE *pc;                /* cfp[0] */
    VALUE *sp;                /* cfp[1] */
    rb_iseq_t *iseq;          /* cfp[2] */
    VALUE flag;               /* cfp[3] */
❶   VALUE self;               /* cfp[4] / block[0] */
    VALUE klass;              /* cfp[5] / block[1] */
    VALUE *ep;                /* cfp[6] / block[2] */
    rb_iseq_t *block_iseq;    /* cfp[7] / block[3] */
❷   VALUE proc;               /* cfp[8] / block[4] */
    const rb_method_entry_t *me;/* cfp[9] */

#if VM_DEBUG_BP_CHECK
    VALUE *bp_check;          /* cfp[10] */
#endif
} rb_control_frame_t;
```

Listing 8-4: The definition of rb_control_frame_t *from* vm_core.h

Notice that this C structure also contains the same values as the rb_block_t structure: everything from self at ❶ to proc at ❷. The fact that these two structures share the same values is one of the interesting, but confusing, optimizations Ruby uses internally to speed things up. Whenever you first refer to a block by passing it into a method call, Ruby needs to create a new rb_block_t structure and copy values such as the EP from the current rb_control_frame_t structure into it. However, because these two structures contain the same values in the same order (rb_block_t is a subset of rb_control_frame_t), Ruby can avoid creating a new rb_block_t structure and instead set the new block pointer to the common portion of the rb_control_frame_t structure. In other words, instead of allocating new memory to hold the new rb_block_t structure, Ruby simply passes a pointer to the middle of the rb_control_frame_t structure. By doing so, Ruby avoids unnecessary calls to malloc and speeds up the process of creating blocks.

Experiment 8-1: Which Is Faster: A while Loop or Passing a Block to each?

Ruby code containing blocks is often more elegant and succinct than the equivalent code in older languages, such as C. For example, in C we would write the simple `while` loop shown in Listing 8-5 to add up the numbers 1 through 10.

```
#include <stdio.h>
main()
{
  int i, sum;
  i = 1;
  sum = 0;
  while (i <= 10) {
    sum = sum + i;
    i++;
  }
  printf("Sum: %d\n", sum);
}
```

Listing 8-5: Adding up 1 through 10 in C using a `while` loop

Listing 8-6 shows the same `while` loop in Ruby.

```
sum = 0
i = 1
while i <= 10
  sum += i
  i += 1
end
puts "Sum: #{sum}"
```

Listing 8-6: Adding up 1 through 10 in Ruby using a `while` loop

However, most Rubyists would write this code using a range object with a block, as shown in Listing 8-7.

```
sum = 0
(1..10).each do |i|
  sum += i
end
puts "Sum: #{sum}"
```

Listing 8-7: Adding up 1 through 10 in Ruby using a range object and a block

Aesthetics aside, is there any performance penalty for using a block here? Does Ruby slow down significantly in order to create the new `rb_block_t` structure, copy the `EP` value, and create new stack frames?

Well, I won't benchmark the C code because clearly it will be faster than either option using Ruby. Instead, let's measure how long it takes Ruby, using a simple `while` loop, to add up the integers 1 through 10 to obtain 55, as shown in Listing 8-8.

```
require 'benchmark'
ITERATIONS = 1000000
Benchmark.bm do |bench|
  bench.report("iterating from 1 to 10, one million times") do
    ITERATIONS.times do
      sum = 0
      i = 1
      while i <= 10
        sum += i
        i += 1
      end
    end
  end
end
```

Listing 8-8: Benchmarking the `while` loop (while.rb)

Here, I'm using the benchmark library to measure the time required to run the `while` loop one million times. Admittedly, I'm using a block to control the million iterations (`ITERATIONS.times do`), but I'll use the same block in the next test as well. Using Ruby 2.0 on my laptop, I can run through this code in just under a half second:

```
$ ruby while.rb
      user     system      total        real
      iterating from 1 to 10, one million times  0.440000   0.000000
                                                 0.440000 (  0.445757)
```

Now let's measure the time required to run the code shown in Listing 8-9, which uses `each` with a block.

```
require 'benchmark'
ITERATIONS = 1000000
Benchmark.bm do |bench|
  bench.report("iterating from 1 to 10, one million times") do
    ITERATIONS.times do
      sum = 0
      (1..10).each do |i|
        sum += i
      end
    end
  end
end
```

Listing 8-9: Benchmarking a call to a block (each.rb)

This time it takes somewhat longer to run through the loop a million times, about 0.75 seconds:

```
$ ruby each.rb
      user     system      total        real
      iterating from 1 to 10, one million times  0.760000   0.000000
                                                 0.760000 (  0.765740)
```

Ruby requires about 71 percent more time to call the block 10 times, compared to iterating through the simple `while` loop 10 times (see Figure 8-9).

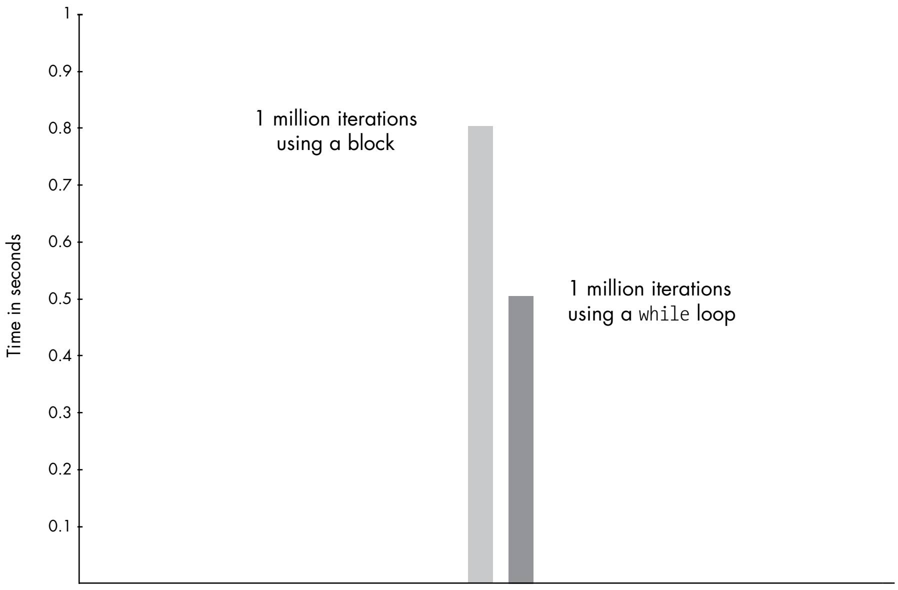


Figure 8-9: Ruby 2.0 uses 71 percent more time calling a block vs. a simple `while` loop. The graph shows the time for one million iterations (in seconds).

Using `each` is slower because internally the `Range#each` method has to call or yield to the block each time around the loop. This involves a fairly large amount of work. In order to yield to a block, Ruby first has to create a new `rb_block_t` structure for that block, setting the `EP` in the new block to the referencing environment and passing the block into the call to `each`. Then each time around the loop Ruby has to create a new stack frame on YARV's internal stack, call the block's code, and finally copy the `EP` from the block to the new stack frame. Running a simple `while` loop is faster because Ruby needs only to reset the `PC`, or program counter, each time around the loop. It never calls a method or creates a new stack frame or a new `rb_block_t` structure.

Seventy-one percent more time seems like a large performance penalty, and, depending on your work and the context of this `while` loop, it may or may not be important. If this loop were part of a time-sensitive, critical operation that your end users were waiting for, and if there weren't other expensive operations inside the loop, it might be worth writing the iteration using an old-fashioned C-style `while` loop. However, the performance of most Ruby applications, and certainly Ruby on Rails websites, is usually limited by database queries, network connections, and other factors, not by Ruby execution speed. It's rare that Ruby's execution speed has an immediate, direct impact on your application's overall performance. (Of course, if you're using a large framework, such as Ruby on Rails, then your Ruby code is a very small piece of a very large system. I imagine that Rails uses blocks and iterators many, many times while processing a simple HTTP request, apart from the Ruby code you write yourself.)

Lambdas and Procs: Treating a Function as a First-Class Citizen

Now to look at a more convoluted way of printing the "quick brown fox" string to the console. Listing 8-10 shows an example of using `lambda`.

```
❶ def message_function
❷   str = "The quick brown fox"
❸   lambda do |animal|
❹     puts "#{str} jumps over the lazy #{animal}."
    end
  end
❺ function_value = message_function
❻ function_value.call('dog')
```

Listing 8-10: Using `lambda` in Ruby

Let's step through this code carefully. First, at ❶ we define a method called `message_function`. Inside `message_function`, we create a local variable at ❷ called `str`. Next, at ❸ we call `lambda`, and pass it a block. Inside this block, at ❹, we print the "quick brown fox" string again. However, `message_function` won't immediately display the string because it doesn't actually call the block at ❸. Instead, `lambda` returns the block we give it as a data value, which in turn is returned by `message_function`.

This is an example of "treating a function as a first-class citizen," to paraphrase a commonly used computer science expression. Once the block is returned from `message_function`, we save it in the local variable `function_value` at ❺ and then call it explicitly, using the `call` method at ❻. With the `lambda` keyword—or the equivalent `proc` keyword—Ruby allows you to convert a block into a data value in this way.

I have lots of questions about Listing 8-10. What happens when we call `lambda`? How does Ruby convert the block into a data value, and what does it mean to treat this block as a first-class citizen? Does `message_function` return an `rb_block_t` structure directly, or does it return an `rb_lambda_t` structure? And what information would `rb_lambda_t` contain (see Figure 8-10)?

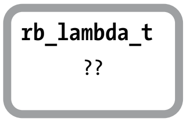


Figure 8-10: Does Ruby use an `rb_lambda_t` C structure? And if so, what would it contain?

Stack vs. Heap Memory

Before we can answer these questions, we need to take a closer look at how Ruby saves your data. Internally, Ruby saves your data in two places: on the *stack* or in the *heap*.

We've seen the *stack* before. This is where Ruby saves local variables, return values, and arguments for each of the methods in your program. Values on the stack are valid only for as long as that method is running. When a method returns, YARV deletes its stack frame and all the values inside it.

Ruby uses the *heap* to save information that you might need for a while, even after a particular method returns. Each value in the heap remains valid for as long as there is a reference to it. Once a value is no longer referred to by any variable or object in your program, Ruby's garbage collection system deletes it, freeing its memory for other uses.

This scheme is not unique to Ruby. In fact, it's used by many other programming languages, including Lisp and C. And remember, Ruby itself is a C program. YARV's stack design is based on the way C programs use the stack, and Ruby's heap uses the underlying C heap implementation.

The stack and heap differ in one other important aspect. Ruby saves only references to data on the stack—that is, the `VALUE` pointers. For simple integer values, symbols, and constants such as `nil`, `true`, or `false`, the reference is the actual value. However, for all other data types, the `VALUE` is a pointer to a C structure containing the actual data, such as `RObject`. If only the `VALUE` references go on the stack, where does Ruby save the structures? In the heap. Let's look at an example to understand this better.

A Closer Look at How Ruby Saves a String Value

Let's look in detail at how Ruby handles the string value `str` from Listing 8-10. First, imagine YARV has a stack frame for the outer scope but has yet to call `message_function`. Figure 8-11 shows this initial stack frame.

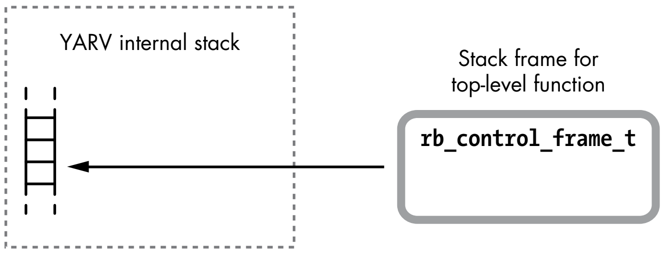


Figure 8-11: To execute the code in Listing 8-11, Ruby starts with an initial stack frame.

In this figure you can see YARV's internal stack on the left and the `rb_control_frame_t` structure on the right. Now suppose Ruby executes the `message_function` function call shown at ❺ in Listing 8-10. Figure 8-12 shows what happens next.

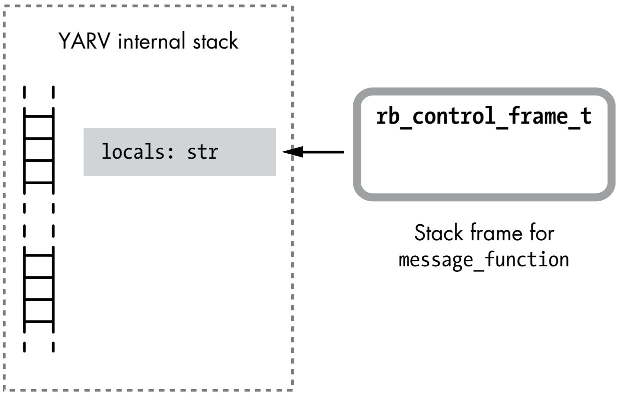


Figure 8-12: Ruby creates a second stack frame when calling `message_function`*.*

Ruby saves the str local variable in the new stack frame used by `message_function`. Let's take a closer look at that `str` variable and how Ruby stores the "quick brown fox" string into it. Ruby stores each of your objects in a C structure called `RObject`, each of your arrays in a structure called `RArray`, each of your strings in a structure called `RString`, and so on. Figure 8-13 shows the "quick brown fox" string saved with `RString`.

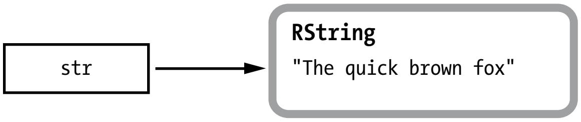


Figure 8-13: Ruby uses the `RString` *C structure to save string values.*

The actual string structure is shown on the right side of the figure, and a reference, or pointer, to the string is shown on the left. When Ruby saves a string value (or any object) onto the YARV stack, it actually places only the reference to the string on the stack. The actual string structure is saved in the heap instead, as shown in Figure 8-14 on the next page.

Once there are no longer any pointers referencing a particular object or value in the heap, Ruby frees that object or value during the next run of the garbage collection system. To demonstrate, suppose that my example code didn't call `lambda` at all but rather immediately returned `nil` after saving the `str` variable, as shown in Listing 8-11.

```
def message_function
  str = "The quick brown fox"
  nil
end
```

Listing 8-11: This code doesn't call `lambda`*.*

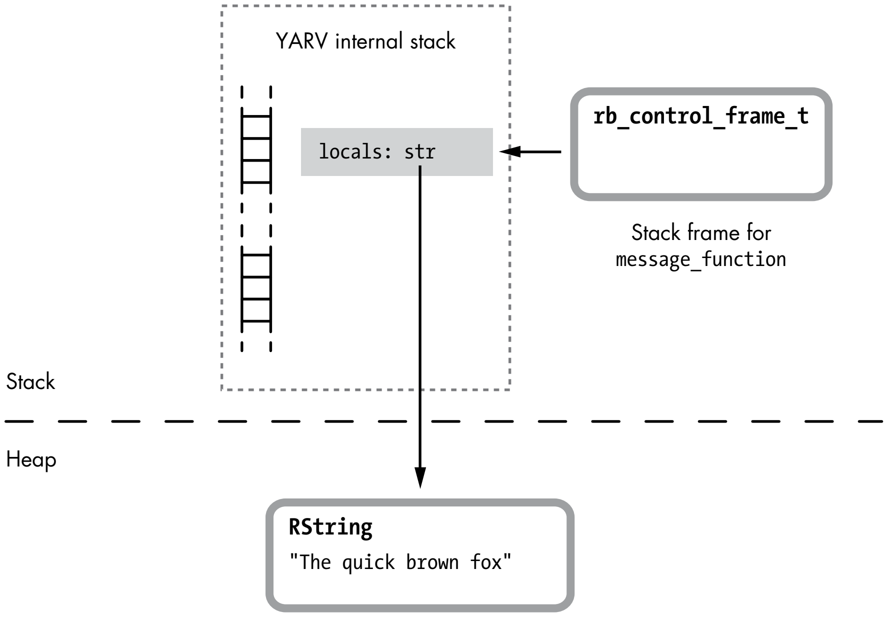


Figure 8-14: The `str` value on the stack is a reference to the `RString` structure saved in the heap.

Once this call to `message_function` finishes, YARV simply pops the `str` value off the stack (as well as any other temporary values saved there) and returns to the original stack frame, as shown in Figure 8-15.

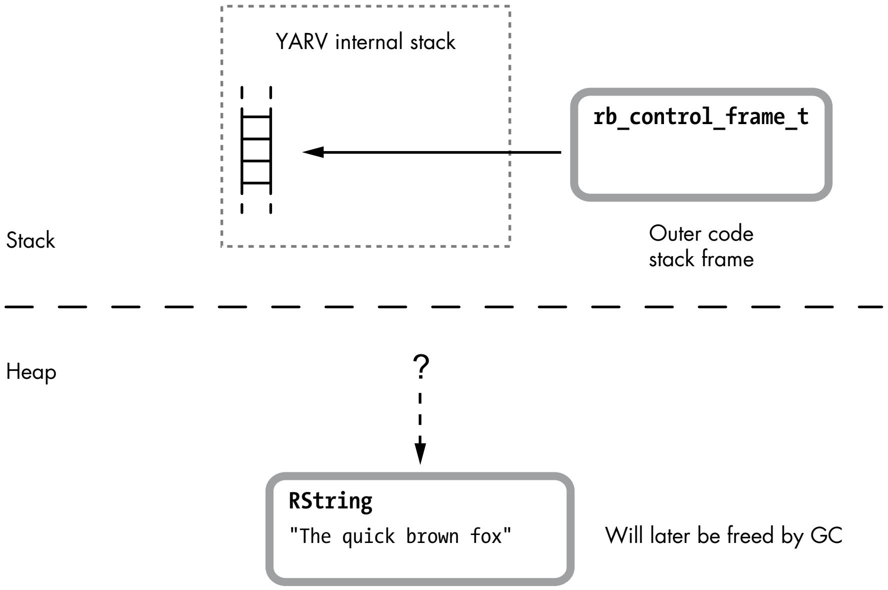


Figure 8-15: Now there is no longer a reference to the `RString` structure.

As you can see in the figure, there is no longer a reference to the RString structure containing the "quick brown fox" string. Ruby's garbage collection system is designed to identify values in the heap that don't have any references to them, like the "quick brown fox" string here. After it identifies them, the GC system will free those orphaned values, returning that memory to the heap.

How Ruby Creates a Lambda

Now that we understand a bit more about the heap and how Ruby uses it, we're ready to learn more about lambdas. Earlier when I used the phrase "treating a function as a first-class citizen," I meant that Ruby allows you to treat functions or code as a data value, saving them into variables, passing them as arguments, and so on. Ruby implements this idea using blocks.

The `lambda` (or `proc`) keyword converts a block into a data value. But remember, blocks are Ruby's implementation of closures. This means the new data value must somehow contain both the block's code and referencing environment.

To see what I mean, let's return to Listing 8-10, repeated here in Listing 8-12 with an eye toward its use of `lambda`.

```
  def message_function
❶   str = "The quick brown fox"
❷   lambda do |animal|
❸     puts "#{str} jumps over the lazy #{animal}."
    end
  end
  function_value = message_function
❹ function_value.call('dog')
```

Listing 8-12: Using `lambda` in Ruby (repeated from Listing 8-10)

Notice at ❹ that when we call the lambda (the block), the `puts` statement inside the block at ❸ can access the `str` string variable defined at ❶ inside `message_function`. How can this be? We've just seen how the `str` reference to the `RString` structure is popped off the stack when `message_function` returns! Obviously, after calling `lambda`, the value of `str` lives on so that the block can access it later.

When you call `lambda`, Ruby copies the entire contents of the current YARV stack frame into the heap, where the `RString` structure is located. For example, Figure 8-16 shows how the YARV stack looks just after the `message_function` starts at ❶ in Listing 8-12. (To keep things simple, I'm not showing the `RString` structure, but remember that the `RString` structure will also be saved in the heap.)

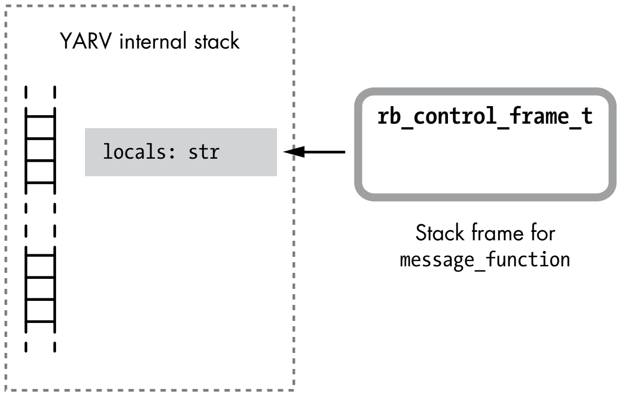


Figure 8-16: Ruby creates a second stack frame when calling `message_function`*.*

Next, Listing 8-12 calls `lambda` at ❷. Figure 8-17 shows what happens in Ruby when you call `lambda`.

The horizontal stack icon below the dotted line shows that Ruby creates a new copy of the stack frame for `message_function` in the heap. Now there is a second reference to the `str` `RString` structure, which means that Ruby won't free it when `message_function` returns.

In fact, along with the copy of the stack frame, Ruby creates two other new objects in the heap:

- An internal environment object, represented by the `rb_env_t` C structure at the lower left of the figure. It's essentially a wrapper for the heap copy of the stack. As we'll see in Chapter 9, you can access this environment object indirectly in your programs using the `Binding` class.
- A Ruby proc object, represented by the `rb_proc_t` structure. This is the actual return value from the `lambda` keyword; it's what the `message_function` function returns.

Note that the new proc object, the `rb_proc_t` structure, contains an `rb_block_t` structure, including the `iseq` and `EP` pointers. Think of a proc as a kind of Ruby object that wraps up a block. As with a normal block, these keep track of the block's code and the referencing environment for its closure. Ruby sets the `EP` in this block to point to the new heap copy of the stack frame.

Also, notice that the proc object contains an internal value called `is_lambda`. This is set to `true` for this example because we used the `lambda` keyword to create the proc. If I had instead created the proc using the `proc` keyword, or simply by calling `Proc.new`, then `is_lambda` would have been set to `false`. Ruby uses this flag to produce the slight behavior differences between procs and lambdas, though it's best to think of procs and lambdas as essentially the same.

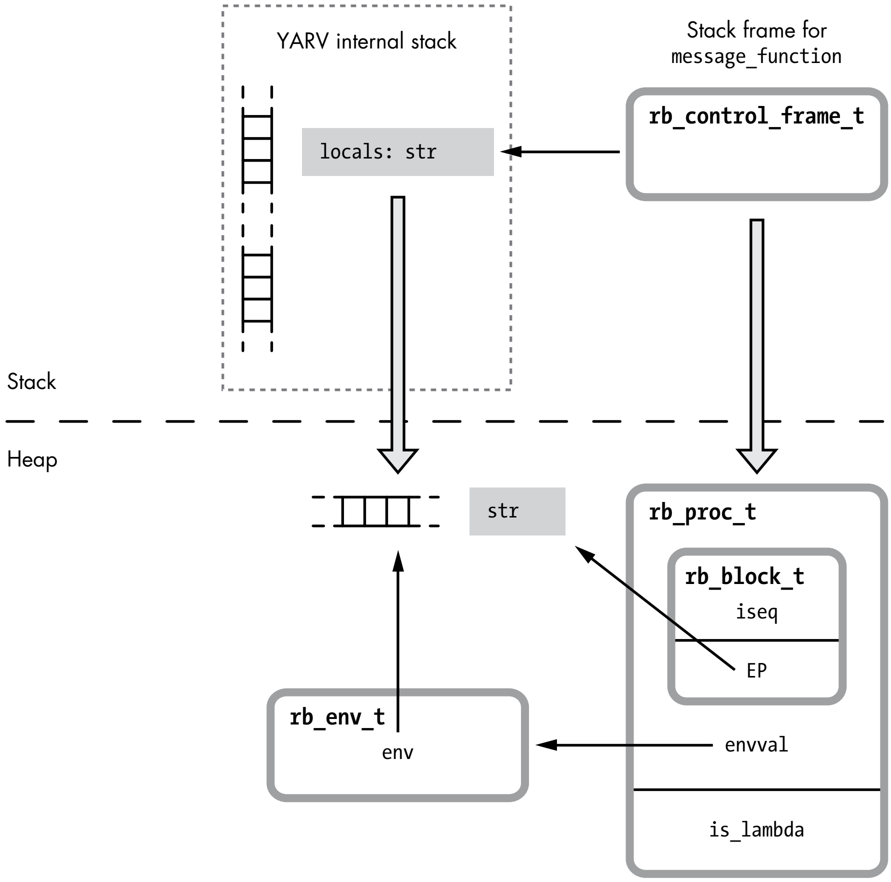


Figure 8-17: When you call `lambda`, Ruby copies the current stack frame to the heap.

How Ruby Calls a Lambda

Let's go back to our lambda example in Listing 8-13.

```
def message_function
  str = "The quick brown fox"
  lambda do |animal|
    puts "#{str} jumps over the lazy #{animal}."
  end
end
❶ function_value = message_function
❷ function_value.call('dog')
```

Listing 8-13: Using `lambda` in Ruby (repeated again from Listing 8-10)

What happens when message_function returns at ❶? Because the lambda or proc object is its return value, a reference to the lambda is saved in the stack frame for the outer scope in the function_value local variable. This prevents Ruby from freeing the proc, the internal environment object, and the str variable, and there are now pointers referring to all of these values in the heap (see Figure 8-18).

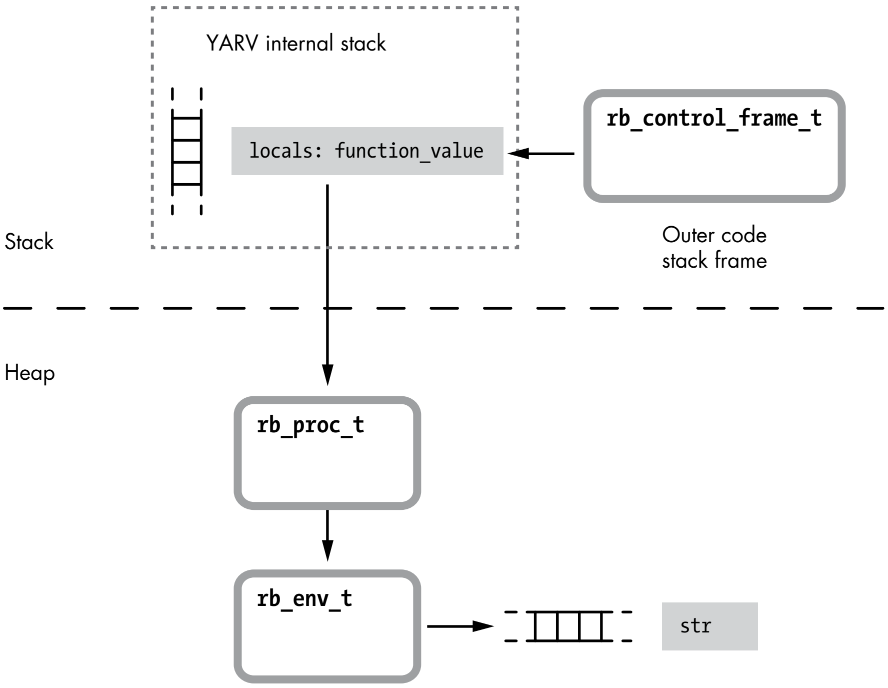


Figure 8-18: Once message_function returns, the surrounding code holds a reference to the proc object.

When Ruby executes the call method on the proc object at ❷, it executes its block as well. Figure 8-19 shows what happens in Ruby when you use the call method on a lambda or proc.

As with any block, when Ruby calls the block inside a proc object it creates a new stack frame and sets the EP to the block's referencing environment. However, that environment is a copy of a stack frame previously copied into the heap; the new stack frame contains an EP that points to the heap. This EP allows the block's call to puts to access the str value defined in message_function. Figure 8-19 shows the argument to the proc, animal, saved in the new stack frame, like any other method or block argument.

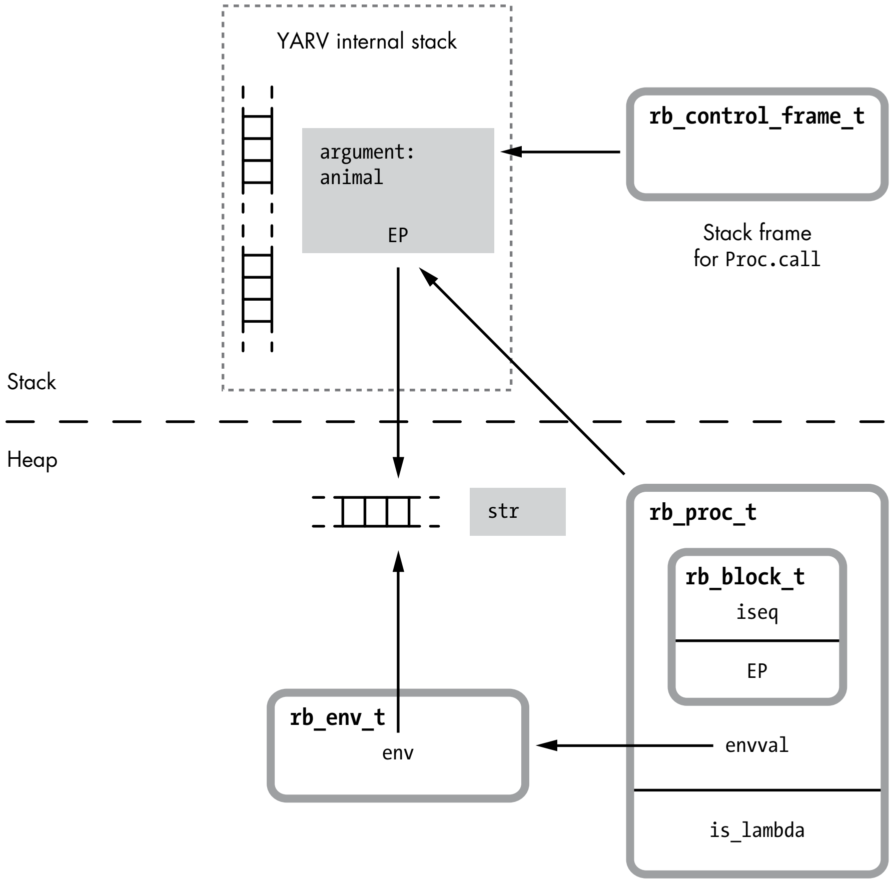


Figure 8-19: Calling a proc object creates a new stack frame as usual and sets the EP *to point to the heap's referencing environment.*

The Proc Object

We've seen that Ruby really has no structure called `rb_lambda_t`. In other words, the structure shown in Figure 8-20 doesn't actually exist.

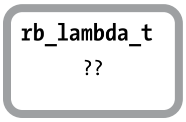
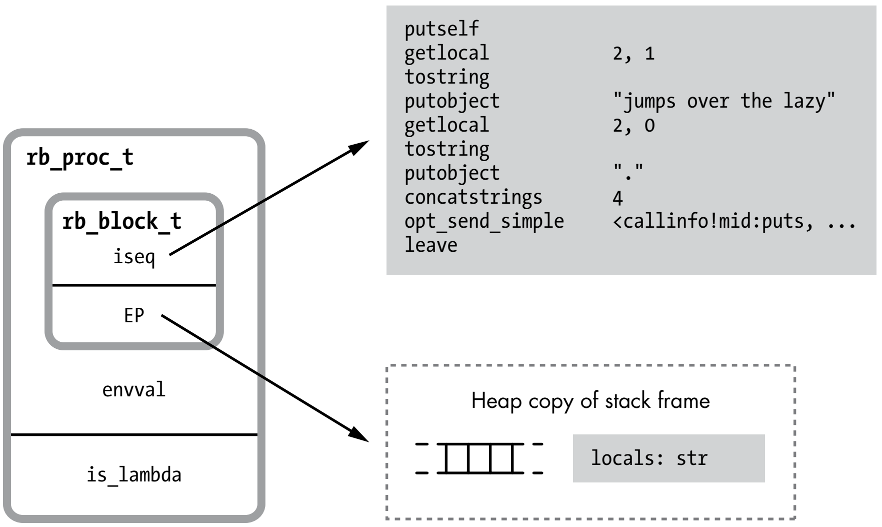


Figure 8-20: Ruby doesn't actually use a structure called `rb_lambda_t`*.*

Instead, in this example, Ruby's lambda keyword created a proc object—really, a wrapper for the block we passed to the lambda or proc keyword. Ruby represents procs using an rb_proc_t C structure, as you can see in Figure 8-21.



```
putself
getlocal          2, 1
tostring
putobject         "jumps over the lazy"
getlocal          2, 0
tostring
putobject         "."
concatstrings     4
opt_send_simple   <callinfo!mid:puts, ...
leave
```

Figure 8-21: Ruby procs are closures; they contain pointers to a function and a referencing environment.

This is a closure: It contains a function along with the environment that function was referred to or created in. The environment is a persistent copy of the stack frame saved in the heap.

A proc is a Ruby object. It contains the same information as other objects, including the RBasic structure. To save its object-related information, Ruby uses a structure called RTypedData, along with rb_proc_t, to represent instances of the proc object. Figure 8-22 shows how these structures work together.

You might think of RTypedData as a kind of trick that Ruby's C code uses to create a Ruby object wrapper around a C data structure. In this case, Ruby uses RTypedData to create an instance of the Proc Ruby class that represents a single copy of the rb_proc_t structure. The RTypedData structure contains the same RBasic information as all Ruby objects:

flags Certain internal technical information Ruby needs to track

klass A pointer to the Ruby class that the object is an instance of; the Proc class in this example

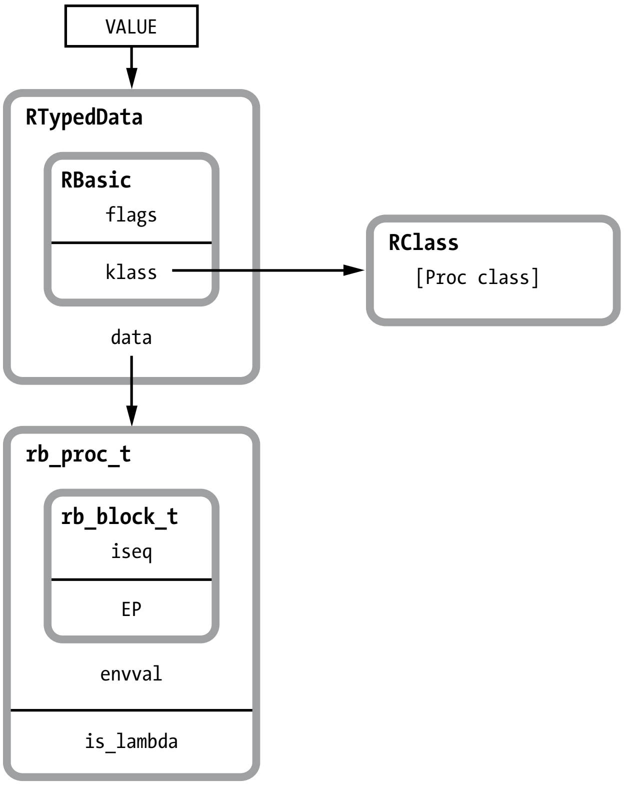


Figure 8-22: Ruby saves the object-related information about proc objects in the `RTypedData` structure.

Figure 8-23 takes another look at how Ruby represents a proc object. The proc object is on the right next to an `RString` structure.

Notice that Ruby handles the string value and the proc similarly. As with strings, procs can be saved into variables or passed as arguments to a function call. Ruby uses the `VALUE` pointer to the proc whenever you refer to one or save one into a variable.

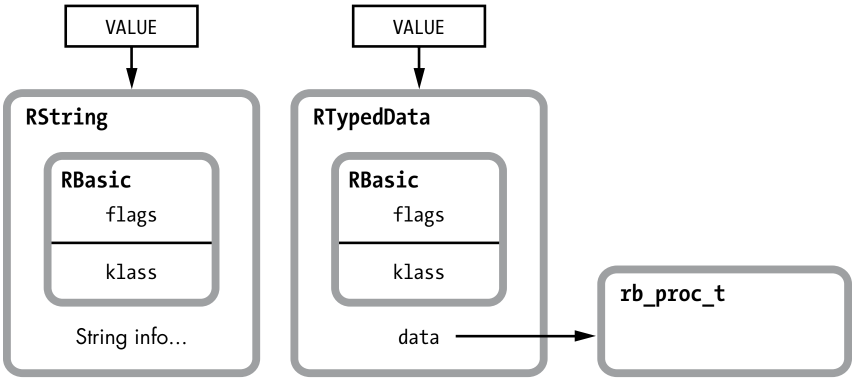


Figure 8-23: Comparing a Ruby string with a proc

Experiment 8-2: Changing Local Variables After Calling lambda

Listings 8-10 through 8-13 show how calling `lambda` copies the current stack frame in the heap. Now for a slightly different example. Listing 8-14 is basically the same, except that the line at ❷ changes `str` after calling `lambda`.

```
  def message_function
    str = "The quick brown fox"
❶   func = lambda do |animal|
      puts "#{str} jumps over the lazy #{animal}."
    end
❷   str = "The sly brown fox"
    func
  end
  function_value = message_function
❸ function_value.call('dog')
```

Listing 8-14: Which version of `str` will `lambda` copy to the heap (modify_after_lambda.rb)?

Because we call `lambda` at ❶ before changing `str` to `The sly brown fox` at ❷, Ruby should have copied the stack frame to the heap, including the original value of `str`. That means that when we call the lambda at ❸, we should see the original "quick brown fox" string. However, running the code, we get the following:

```
$ ruby modify_after_lambda.rb
The sly brown fox jumps over the lazy dog.
```

What happened? Ruby somehow copied the new value of `str`, `The sly brown fox`, to the heap so we could access it when we called the lambda at ❸.

To find out how Ruby did this, let's look more closely at what happens when you call `lambda`. Figure 8-24 shows how Ruby copies the stack frame to the heap, including the value `str` from Listing 8-14.

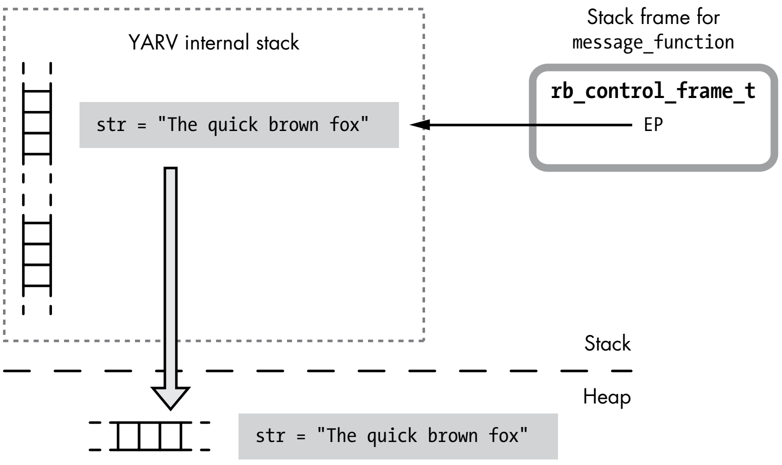
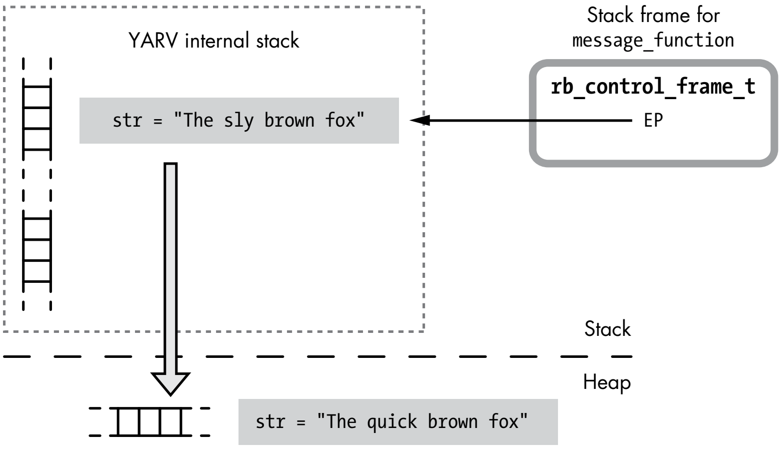

YARV internal stack
Stack frame for message_function
rb_control_frame_t
EP
str = "The quick brown fox"
Stack
Heap
str = "The quick brown fox"

Figure 8-24: When you call `lambda`, Ruby copies the stack frame to the heap.

Once this copy is made, the code at ❷ in Listing 8-14 changes `str` to the "sly fox" string:

```
str = "The sly brown fox"
```

Because Ruby copied the stack frame when we called `lambda`, we should be modifying the original copy of `str`, not the new lambda copy (see Figure 8-25).

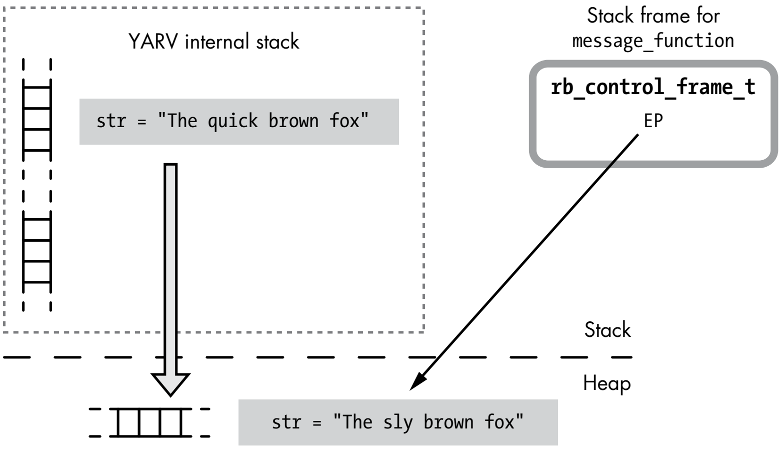


YARV internal stack
Stack frame for message_function
rb_control_frame_t
EP
str = "The sly brown fox"
Stack
Heap
str = "The quick brown fox"

Figure 8-25: Does Ruby continue to use the original stack frame after making a heap copy?

The new heap copy of the string should have remained unmodified, and calling the lambda later should have given the original "quick fox" string, not the modified "sly fox" one. How does Ruby allow us to modify the new persistent copy of the stack once it's been created by `lambda`?

As it turns out, once Ruby creates the new heap copy of the stack (the new `rb_env_t` structure or internal environment object), it resets the `EP` in the `rb_control_frame_t` structure to point to the copy. Figure 8-26 shows how Ruby resets the `EP` after creating a persistent heap copy of a stack frame.



YARV internal stack
Stack frame for message_function
rb_control_frame_t
EP
str = "The quick brown fox"
Stack
Heap
str = "The sly brown fox"

Figure 8-26: Ruby resets the `EP` after creating a persistent heap copy of a stack frame.

The difference here is that the `EP` now points down to the heap. Now when we call `str = "The sly brown fox"` at ❷ in Listing 8-14, Ruby will use the new `EP` and access the value in the heap, not the original value on the stack. Notice `The sly brown fox` appears in the heap at the bottom of Figure 8-26.

Calling lambda More Than Once in the Same Scope

Another interesting behavior of the `lambda` keyword is that Ruby avoids making copies of the stack frame more than once, as you can see in Listing 8-15.

```
i = 0
increment_function = lambda do
  puts "Incrementing from #{i} to #{i+1}"
  i += 1
end
decrement_function = lambda do
  i -= 1
  puts "Decrementing from #{i+1} to #{i}"
end
```

Listing 8-15: Calling `lambda` twice in the same scope

This code expects both lambda functions to operate on the local variable `i` in the main scope.

But if Ruby made a separate copy of the stack frame for each call to `lambda`, each function would operate on a separate copy of `i`. Look at the following example in Listing 8-16.

```
increment_function.call
decrement_function.call
increment_function.call
increment_function.call
decrement_function.call
```

Listing 8-16: Calling the lambdas created in Listing 8-15

If Ruby used a separate copy of `i` for each lambda function, the previous listing would generate the output shown in Listing 8-17.

```
Incrementing from 0 to 1
Decrementing from 0 to -1
Incrementing from 1 to 2
Incrementing from 2 to 3
Decrementing from -1 to -2
```

Listing 8-17: The output we would expect if each call to `lambda` created its own copy of the stack frame

But we actually see the output shown in Listing 8-18.

```
Incrementing from 0 to 1
Decrementing from 1 to 0
Incrementing from 0 to 1
Incrementing from 1 to 2
Decrementing from 2 to 1
```

Listing 8-18: Because the lambda functions share the same heap copy of the stack, running Listing 8-16 generates this output.

Usually this is what you expect: Each block you pass to the lambdas accesses the same variable in the parent scope. Ruby achieves this by checking whether the `EP` already points to the heap. If so, as with the second call to `lambda` in Listing 8-15, Ruby won't create a second copy; it will simply reuse the same `rb_env_t` structure in the second `rb_proc_t` structure. Ultimately, both lambdas use the same heap copy of the stack.

Summary

In Chapter 3 we saw how YARV creates a new stack frame whenever you call a block, just as it does when you call a method. At first glance, Ruby blocks appear to be a special kind of method that you can call and pass arguments to. However, as we've seen in this chapter, there's more to blocks than meets the eye.

Looking closely at the `rb_block_t` structure, we saw how blocks implement the computer science concept of *closure* in Ruby. Blocks are the combination of a function and an environment to use when calling that function. We learned that blocks have a curious dual personality in Ruby: They are similar to methods, but they also become part of the method that you call them from. The simplicity with which Ruby's syntax allows for this dual role is one of the language's most beautiful and elegant features.

Later we saw how Ruby allows you to treat functions or code as first-class citizens using the `lambda` keyword, which converts a block into a data value that you can pass, save, and reuse. After reviewing the differences between stack and heap memory, we explored the way that Ruby implements lambdas and procs, and we saw that Ruby copies the stack frame to the heap when you call `lambda` or `proc` and reuses it when you call the lambda's block. Finally, we saw how the proc object represents code as a data object in Ruby.