Testing out the memory model of various languages

Mark Armstrong

October 2, 2019

Contents

1 Ruby

Unlike the C-like languages, in Ruby, assignment between variables never copies the value. Instead, an assignment `y = x` makes the variable `y` an *alias* (an alternate name) for `x`.

1.1 Non-mutable values

In addition, for *non-mutable* types such as integers, assignments of variables to values simply makes the variable an alias for the value.

We can see this by examining the `object_id` (Ruby's reference type) of `x` and `y` below. They both refer to the same object, `5`. *Any* way we have to "store" 5 will have the same object_id, because 5 is immutable; only one copy can exist.

```
x = 5
y = x

puts "5's object_id is #{5.object_id}"
puts "x's object_id is #{x.object_id}"
puts "y's object_id is #{y.object_id}"
```

1.2 Mutable values

For mutable values, more than one copy of the value may exist. For instance, strings are mutable, so in the below x and y refer to different instances of the same string.

```
x = "hello world"
y = "hello world"

puts "(A new) \"hello world\"'s object_id is #{"hello world".object_id}"
puts "                    x's object_id is #{x.object_id}"
puts "                    y's object_id is #{y.object_id}"
```

However, assignment between variables (assignment of the form y = x) still creates aliases.

```
x = "hello world"
y = x

puts "(A new) \"hello world\"'s object_id is #{"hello world".object_id}"
puts "                    x's object_id is #{x.object_id}"
puts "                    y's object_id is #{y.object_id}"
```

1.3 The problem with aliases

…

2 F#

2.1 Immutability is the default

As is often the case in functional languages, variables in F# are by default immutable.

```
let x = 10

printfn "x is immutable, so while in scope it will always be %d" x
```

Note that `x` might be shadowed by another declaration of `x` (though we can't redeclare it in the same scope).

2.2 The `mutable` keyword

F# provides some support for imperative programming by allowing a variable to be declared `mutable`, so its value can be updated. This update (assignment) is written using the left-facing arrow `<-`.

```
let mutable x = 10

printfn "x is mutable, so even though right now it's value is %d..." x

x <- x + 1

printfn "it's value can change to %d!" x
```

2.3 Mutability by references

F# also includes *reference* types which allow mutability.

```
let y = ref 2

printfn "The identifier y is bound to a reference to %d." !y

printfn "y is actually a record %A" y

y := 3

printfn "Now y's reference points to %d instead." !y

printfn "What happened is that y is now the record %A" y
```

3 Oz

3.1 Single assignment

The result of this code is obvious; `X` becomes the sum of `2` and `3`, so we get `5` in the browser.

```
declare X Y Z in
Y = 2
Z = 3
X = Y + Z

{Browse X}
```

If we try to run the following code, what should the result be? In the single-assignment store model, which is a kernel of Oz, the assignment `X = Y + Z` will block until we know what `Y` and `Z` are. So we never get output if we just feed this to the virtual machine, because it gets stuck. (We do get some type information; it knows that `Y` and `Z` are `char` type, since they are added, and we get a long list of potential types for `X`).

```
declare X Y Z in
X = Y + Z
Y = 2
Z = 3

{Browse X}
```

We can feed the lines `Y = 2` and `Z = 3` (the command `C-. C-l` feeds one line), and if we do so, then we get the output.

3.2 Incorporating the treading into the code

Manually feeding lines is a hassle. We can automate it away by explicitely threading our code.

```
declare X Y Z in

thread
  X = Y + Z
end

thread
  Y = 2
end

thread
  Z = 5
end
```

```
{Browse X}
```

3.3 Order doesn't matter

```
declare X Y Z in

thread
  X = Y + Z
end

thread
  {Delay 1000}
  Y = 2
end

thread
  {Delay 1000}
  Z = 5
end

{Delay 2000}
{Browse X}
```