Functional Programming

Anthony Stephenson

12/1/2020

Contents

Functional Programming

In (pure) functional programming, the fundamental building block is the function, where here "function" explicitly means a function in the mathematical sense; i.e. a machine that takes an input and deterministically calculates and returns an output. Here we will briefly consider functional programming through the structure of *category theory*, in the context of R.

Category theory

The underlying mathematical structure that provides the basis for functional programming is category theory. Delving beyond the surface of category theory is beyond the scope of this document, but I will provide a brief introduction to some salient points with respect to functional programming, in particular, in R. For (much) more detail, see Milewski (2018).

Composition

In category theory, a category is a collection of objects with morphisms between them. The morphisms can be composed and this composition is associative. Each object has an associated identity morphism, that returns itself. In R, this is very simple, in part due to its dynamic typing:

```
id <- function(x) x
```

We can define a function, `compose` (in a new package `functools`) to do what we want, or actually we can use the pipe operator `%>%`

```
library(devtools)
```

```
## Loading required package: usethis
```

```
load_all("functools")
```

```
## Loading functools
```

```
library(magrittr)
g <- compose(log, sum, exp)
c(2,2) %>% exp %>% sum %>% log
```

```
## [1] 2.693147
```

```
g(c(2,2))
```

```
## [1] 2.693147
```

```
log(sum(exp(c(2,2))))
```

```
## [1] 2.693147
```

Types

Types define distinct objects with specific (different) properties. In computing the basic data types tend to include `int`s, `char`s, `bool`s etc and reflect the use of sets to denote the "type" of a variable in a mathematical function, e.g. consider a function $f : \mathbb{Z} \to \mathbb{R}$ that takes a natural number (i.e. an `int`) to a real number (i.e. a `float`) and in an imagined static-typed version of R might look something like `f <- float function(int x) 1.5^x` (the actual computation is irrelevant apart from to generate a real number).

Static typing can be quite helpful in reasoning about how a function will behave and more importantly, how *compositions* of functions might behave, since you can immediately see the input and output types of a bunch of functions to be called sequentially and therefore whether or not they are compatible. Static typing also catches such errors before run-time and potentially makes debugging more straightforward. Dynamic typing has its own benefits in terms of simplicity of writing and makes polymorphic implementations implicit rather than explicit.

Monoids

A monoid is a set with an associative binary operation and an identity element. We can easily define some monoids in R, starting with the most obvious, that of addition of numeric scalars with identity element 0. We can also define marginally more interesting ones, for example, for row-based concatenation of `data.frame`s:

```
library(dplyr)
```

```
##
## Attaching package: 'dplyr'
```

```
## The following object is masked _by_ '.GlobalEnv':
##
##     id
```

```
## The following objects are masked from 'package:stats':
##
##     filter, lag
```

```
## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union
```

```
`%rcat%` <- bind_rows

A <- data.frame(x = 1:10, y = 21:30)
B <- data.frame(z = 31:40)
C <- data.frame(q = 501:510, r = 401:410)
```

```
# Associativity:
v1 <- (A %rcat% B) %rcat% C
v2 <- A %rcat% (B %rcat% C)

all.equal(v1, v2)
```

```
## [1] TRUE
```

```
# Identity
v1 <- A %rcat% data.frame()
v2 <- data.frame() %rcat% A

all.equal(v1,v2)
```

```
## [1] TRUE
```

This a relatively trivial example, but actually, functions under composition form a monoid. In R we can (approximate) this by using the pipe operator `%>%` although strictly speaking it does not preserve types, since we can start and end a sequence of pipes with objects rather than functions.

Note one useful fallout from monoidal structure is parallelisation. Since monoidal operations are associative by definition, their order does not matter, so performing such an operation on a large dataset can instead be broken into smaller chunks and then combined at the end using the same operation.

For a more detailed look at monoids in R, see ("Monoids for Data Scientists," n.d.).

Currying

Currying is the technique of converting a function of multiple arguments into a series of single-argument functions. *Uncurrying* is effectively the inverse operation.

Curry: $f(x, y, z) \to g(x)(y)(z)$

There are various packages that implement more or less the same (slightly incorrect) version of currying as partial calls. For example the `purrr::partial` function. Given a partial function, we can implement a simple currying operation by composing partial calls, using the `purrr::pmap` function.

```
library(purrr)
```

```
##
## Attaching package: 'purrr'
```

```
## The following object is masked from 'package:magrittr':
##
##     set_names
```

```
## The following object is masked from 'package:functools':
##
##     compose
```

```
f <- function(a, b) a + b
g <- partial(f, 1)
g(2)
```

```
## [1] 3
```

```
partial(partial(f, 1), 2)()
```

```
## [1] 3
```

```
# implement currying by passing all arguments using pmap but deferring evaluation with partial.
simple_curry <- function(f, ...) {
  args <- list(...)
  g <- partial(pmap, args, f)
  return(g)
}

g <- simple_curry(f, 1, 2)
g()
```

```
## [[1]]
## [1] 3
```

We can define a limited version of partial functionality with the `%>%` operator that allows the conversion of a function of multiple variables to a unary function with use of the `.` placeholder:

```
f <- function(a, b) a + b
g <- . %>% f(1, .)
g()
```

```
## Error in freduce(value, `_function_list`): argument "value" is missing, with no default
```

```
g(2)
```

```
## [1] 3
```

Monads

Monads are *type constructors* with two associated operations: `return` or `unit` and `join` or `bind`.

`unit` converts a basic type T into an augmented type MT that has additional properties, i.e.

$$unit : T \rightarrow MT$$

For example, we could define a type called "Optional[T]" that takes some base type T and can return either that same type T or some form of `NULL` value. In order to retain function composition with this augmented type MT, we need to define a further operation

`bind` unwraps a monadic value (MT) in order to pass it to another function that returns a monadic value, i.e.

$$bind : M(MT) \rightarrow MT$$

Effectively, it allows us to transform an operation that acts on the underlying type to act on our augmented type. A more informative name might be "composition operator".

The general motivation for monads is to implement functionality that traditionally requires the use of *impure* functions (i.e. ones with side effects that change global state) with only pure functions. For example when we directly require side effects (i.e. IO) or want to propagate exceptions. This is achieved by embellishing the output of a function in some way, which is achieved with the `unit` function above, where we convert a type to a new *monadic* type (that is in some way useful for us) and introducing a binary operator (`bind`) that allows us to compose functions with embellished outputs together. (Again, for a *much* more detailed account with examples, see Milewski (2018)).

As it happens, the `magrittr` pipe `%>%` (as `bind`) combined with the `unit <- id` act monadically. We can verify this by checking that the `unit` acts as a right and left identity and the `bind` is associative:

```
f <- function(x) x^2
g <- function(x) x^3 + x
a <- c(1,3,5,24)
```

```r
b <- c(12,24,22,5)

# left identity
v1 <- id(a) %>% f
all.equal(v1, f(a))
```

```
## [1] TRUE
```

```r
# right identity
v2 <- a %>% f %>% id
all.equal(v1, v2)
```

```
## [1] TRUE
```

```r
# associativity
v3 <-  (a %>% f) %>% g
v4 <- a %>% (function(x) f(x) %>% g)
all.equal(v3, v4)
```

```
## [1] TRUE
```

Maybe

Returning to our suggested example "Optional", we can define a slightly more generic version that can be used more flexibly: a *Maybe* type-class which takes on values of either a given type, or *nothing*. This can allow you to, for example, handle errors in a pipeline of composed functions, by either returning the desired output of the function, or "nothing" along with an error message if an exception is thrown by the (wrapped) function.

Define the Maybe monad, i.e. the constructor functions and the composition operator.

```r
just <- function(x) {
  out <- list(
        type = "Just",
        content = x
    )
  class(out) <- append(class(out), "maybe")
  return(out)
}

nothing <- function(errorString) {
  out <- list(
    type = "Nothing",
    content = errorString
  )
  class(out) <- "maybe"
  return(out)
}

`%>=%` <- function(ma, f) {
  if(!is(ma, "maybe")) stop("Provide a maybe value left of '%>=%' !")
  if(ma[[1]]=="Nothing") {
    return(ma) # If Nothing, just pass on the Nothing
  } else {
    out <- f(ma[[2]])
    # Check if the function returns a maybe value
    if(is(out, "maybe")) {
```

```
      return(out)
    } else {
      stop("RHS function must return a Maybe.")
    }
  }
}
```

Now test the Maybe monad as a wrapper for Cholesky decomposition that prevents the function `chol` from being called if the matrix passed is not positive definite. This is really redundant since `chol` fails fairly gracefully in this instance, but it provides an example of the *kind* of use it might have and does provide a slightly more informative error message.

```
m <- matrix(rnorm(25), nrow=5, ncol=5)
chol(m)
```

```
## Error in chol.default(m): the leading minor of order 2 is not positive definite
```

```
safe_chol <- function(x, varName) {
  e <- eigen(x, only.values=TRUE)
  valid <- !any(sapply(e$values, is.complex))
  if (valid) {
    valid <- valid & prod(e$values) > 0
  } else {
    return(nothing("safe_chol: complex eigenvalues present"))
  }
    if(valid) {
        # Return the variable
        return(just(chol(x)))
    } else {
        return(nothing("safe_chol: matrix not positive definite"))
    }
}

m %>=% safe_chol
```

```
## Error in m %>=% safe_chol: Provide a maybe value left of '%>=%' !
```

```
just(m) %>=% safe_chol
```

```
## $type
## [1] "Nothing"
##
## $content
## [1] "safe_chol: complex eigenvalues present"
##
## attr(,"class")
## [1] "maybe"
```

(In the above example, which fails due to the presence of complex eigenvalues, we can imagine implementing another monad to deal with complex numbers, too).

```
# Demonstrate that it works when you can safely calculate
just(m + t(m) + diag(rep(10,5))) %>=% safe_chol
```

```
## $type
## [1] "Just"
##
```

```
## $content
##          [,1]       [,2]       [,3]        [,4]       [,5]
## [1,] 3.329741 -0.5440846  0.3865703 -0.81093918  0.4987488
## [2,] 0.000000  3.3557700 -0.4636812 -0.08658399 -0.4794628
## [3,] 0.000000  0.0000000  3.3902572 -0.35970832 -0.1656793
## [4,] 0.000000  0.0000000  0.0000000  2.67009235  0.9019669
## [5,] 0.000000  0.0000000  0.0000000  0.00000000  3.1881374
##
## attr(,"class")
## [1] "list"  "maybe"
```

Note: the credit for most of the above code on the maybe monad comes from ("Maybe Monad in R," n.d.)

There is also a package, `rmonad`, that implements infix monadic pipe operators.

Automatic-differentiation

To further examine monads, I will construct one for automatic differentiaton from scratch.

In order to design a monad for automatic differentiation, we need to define a wrapper function to create automatically differentiable functions and a composition operator that implements the chain rule. We can also define a product on `ad` objects (monadic values) that implements the product rule.

```
# implement "unit" for ad
# diff[1] is length of grad
# diff[2] is element of grad to set to 1. Set this to 0 for constants
ad <- function(x, diff = c(1,1)) {
  grad <- rep(0, diff[1])
  if (diff[2]>0 && diff[2]<=diff[1]) {
    grad[diff[2]] <- 1
  }
  attr(x, "grad") <- grad
  class(x) <- append(class(x), "ad")
  return(x)
}

# implement the sine function, but with additional gradient attr, cos(x)
sinad <- function(x) {
  out <- sin(x)
  attr(out, "grad") <- cos(x)
  class(out) <- append(class(out), "ad")
  return(out)
}

# implement the exp function, but with additional gradient attr, exp
expad <- function(x) {
  out <- exp(x)
  attr(out, "grad") <- exp(x)
  class(out) <- append(class(out), "ad")
  return(out)
}

logad <- function(x) {
  out <- log(x)
  attr(out, "grad") <- 1/x
  class(out) <- append(class(out), "ad")
```

```
  return(out)
}

# implement the composition operator for ad monads, so that the chain rule is applied
`%>ad%` <- function(ad, f) {
  # check input value
  if(!is(ad, "ad")) stop("Provide a ad value left of '%>ad%' !")
  # get the gradient of the lhs ad input
  grad <- attr(ad, "grad")
  # get the gradient and function evaluation for the rhs monadic-type function
  f_out <- f(as.numeric(ad))
  gradf <- attr(f_out, "grad")
  # set the output
  out <- f_out
  attr(out, "grad") <- grad * gradf
  class(out) <- append(class(out), "ad")
  return(out)
}

"*.ad" <- function(f,g) { ## ad multiplication
  gradf <- attr(f,"grad")
  gradg <- attr(g,"grad")
  f <- as.numeric(f)
  g <- as.numeric(g)
  out <- f*g ## evaluation
  attr(out,"grad") <- f * gradg + g * gradf ## product rule
  class(out) <- append(class(out), "ad")
  return(out)
}

"^.ad" <- function(f,g) { ## ad power
  out <- (g * (f %>ad% logad)) %>ad% expad
  return(out)
}
```

Test that the implementation obeys the requirements for a very simple case:

```
x <- pi/3

# check identity
y <- ad(x) %>ad% sinad
stopifnot(y == sin(x))

# check composition
res <- x %>%
  expad %>ad%
  sinad

identical(res[1], sin(exp(x)))
```

```
## [1] TRUE
```

```
identical(attr(res, "grad"), cos(exp(x))*exp(x))
```

```
## [1] TRUE
```

```
# check product
prod_res <- sinad(x) * expad(x)

identical(prod_res[1], sin(x) * exp(x))
```

```
## [1] TRUE
```

```
identical(attr(prod_res, "grad"), (sin(x) + cos(x)) * exp(x))
```

```
## [1] TRUE
```

```
# check power
pow_res <- ad(x) ^ sinad(x)
identical(pow_res[1], x ^ sin(x))
```

```
## [1] TRUE
```

```
identical(attr(pow_res, "grad"), (cos(x) * log(x) + sin(x)/x) * exp(sin(x) * log(x)))
```

```
## [1] TRUE
```

```
# check standard powers work
const <- ad(2, c(1,0))
x2 <- ad(x) ^ const
identical(x2[1], x^2)
```

```
## [1] TRUE
```

```
identical(attr(x2, "grad"), 2 * x)
```

```
## [1] TRUE
```

This implementation isn't particularly useful and would require manually defining all functions one wanted to differentiate with a clunky naming convention. Using an OOP approach might fare better since then the names of the functions could be overloaded when dealing with `"ad"` objects, whereas in the monadic implementation here, the functions take bare objects and so R will not recognise that it is running an ad operation.

Immutability

To preserve the notion that functional programming does not modify global states, we want to ensure that our variables cannot be modified, i.e. are immutable. R implements a "copy-on-modify" mechanism that preserves immutability in general for objects in R, e.g. ((“Immutability in R,” n.d.))

```
{
  x <- list(1, 2, 3)
  y <- x
  print(pryr::address(x))
  print(pryr::address(y)) # x and y point to the same object
  x[[2]] <- 1 # change the second element of x
  print(x)   # x is changed
  print(y)   # y is not changed
  print(pryr::address(x)) # x points to a new object
  print(pryr::address(y)) # y still points to the original object
}
```

```
## Registered S3 method overwritten by 'pryr':
##   method      from
##   print.bytes Rcpp
```

```
## [1] "0x7fc372e29408"
## [1] "0x7fc372e29408"
## [[1]]
## [1] 1
##
## [[2]]
## [1] 1
##
## [[3]]
## [1] 3
##
## [[1]]
## [1] 1
##
## [[2]]
## [1] 2
##
## [[3]]
## [1] 3
##
## [1] "0x7fc37131aa48"
## [1] "0x7fc372e29408"
```

In this example, we can see that after assignment of `y`, the two variables share the same address (i.e. `y` points to `x` in memory). After we modify an element of `x` however, R copies the variable and modifies that, so that `x` now points to a new object in memory, whilst `y` points to the original still.

In comparison, if no other variables exist that point to the same underlying object, then R will directly mutate the variable:

```
{
  rm(x)
  x <- list(1, 2, 3)
  print(pryr::address(x))
  x[[2]] <- 1
  x
  print(pryr::address(x))
}
```

```
## [1] "0x7fc370f8a9b8"
## [1] "0x7fc370f8a9b8"
```

References

"Immutability in R." n.d. https:https://masterr.org/r/RFP-part5-immutability/.

"Maybe Monad in R." n.d. https://www.r-bloggers.com/2019/05/maybe-monad-in-r/.

Milewski, B. 2018. *Category Theory for Programmers*. Blurb, Incorporated. https://books.google.co.uk/books?id=ZaP-swEACAAJ.

"Monoids for Data Scientists." n.d. https://cfhammill.github.io/posts/monoids.html.