Domain Specific Languages in R

Thomas Mailund

Table of Contents

# Pattern matching

In languages such as ML or Haskell, you can define data types by specifying functions you will use to construct values of any given type. In itself, that is not that interesting, but combined with a pattern matching feature of these languages, you can write very succinct functions for transforming data structures.

In my book on *Functional Data Structures in R* (Mailund 2017a), I describe several algorithms that depend on the transformation of various trees based on their structure. Such transformations involve figuring out the current structure of a tree—does it have a left sub-tree? Is that tree a leaf? If it is a red-black search tree, what is the colour of the tree? And the colour of its right sub-tree? In the algorithms, I presented in that book, most of the functions contained tens of lines of code just for matching such tree structure.

With the language we implement in this chapter, we will make writing such transformation functions vastly more efficient. We will write two main constructions. The first for defining a data structure, which we can use to define red-black search trees like this:

colour := R | B  
rb\_tree := E | T(col : colour, left : rb\_tree, value, right : rb\_tree)

The second constructing is used to match values of such types and then perform actions accordingly. A balancing function for red-black search trees can be implemented succinctly like this:

balance <- function(tree) {  
 match(tree,  
 T(B,T(R,a,x,T(R,b,y,c)),z,d) -> T(R,T(B,a,x,b),y,T(B,c,z,d)),  
 T(B,T(R,T(R,a,x,b),y,c),z,d) -> T(R,T(B,a,x,b),y,T(B,c,z,d)),  
 T(B,a,x,T(R,b,y,T(R,c,z,d))) -> T(R,T(B,a,x,b),y,T(B,c,z,d)),  
 T(B,a,x,T(R,T(R,b,y,c),z,d)) -> T(R,T(B,a,x,b),y,T(B,c,z,d)),  
 otherwise -> tree)  
}

This function is mere eight lines, compared to the 42 lines of code used in *Functional Data Structures in R*, where I also use some pattern matching tricks but not a domain-specific language uniquely designed for it. Such a language is what we will implement in this chapter.

For the chapter, we need to will use the following packages:

library(rlang)  
library(magrittr)  
library(dplyr)

We will also need the make\_args\_list function we defined in Chapter **¿sec:lambda?**.

make\_args\_list <- function(args) {  
 res <- replicate(length(args), substitute())  
 names(res) <- args  
 as.pairlist(res)  
}

## Constructors

The first construction we must implement is for defining data types. Here, we will use the := operator. An assignment has the lowest precedence, which means that whatever we write to the left or right of this operator will be arguments to the function. We do not have to worry about an expression in our language being translated into some call object of a different type. We cannot override the other assignment operators, <-, -> and =, so we have to use :=. Since this is also traditionally used to mean “defined to be equal to”, it works quite well.

The approach we take in implementing this part of the pattern matching DSL is different from the examples we have seen earlier. We do not create a data structure that we can analyse nor do we evaluate expressions directly from expressions in our new language. Instead, we combine parsing expressions with code generation—we generate new functions and objects while we parse the specification. We add these functions, and other objects for constants, to the environment in which we call :=. Adding these objects to this environment allows us to use the constructors after we have defined them with no further coding, but it does mean that calling := will have side-effects.

The construction function will expect a type name as its left-hand-side parameter and an expression describing the different ways of constructing elements of the type on its right-hand side. We will translate the left-hand side into a quosure because we want to get its associated environment. The right-hand side we will turn into an expression. For the construction specification, we do not want to evaluate any of the elements (unless the user invokes quasi-quotations). The left-hand side—the type we are defining—is just treated as a string, since that is how the S3 system deals with types, so we will make sure that it is a single symbol and then get the string representation of it. For this, we can use the quo\_name function from rlang. The right-hand side we have to parse, but we delegate this to a separate function that we define below. Finally, we specify a function for pretty-printing elements of the new type we define.

`:=` <- function(data\_type, constructors) {  
 data\_type <- enquo(data\_type)  
 constructors <- enexpr(constructors)  
  
 stopifnot(quo\_is\_symbol(data\_type))  
 data\_type\_name <- quo\_name(data\_type)  
 process\_alternatives(  
 constructors,   
 data\_type\_name,   
 get\_env(data\_type)  
 )  
  
 assign(paste0("toString.", data\_type\_name),  
 deparse\_construction, envir = get\_env(data\_type))  
 assign(paste0("print.", data\_type\_name),  
 construction\_printer, envir = get\_env(data\_type))  
}

The last two statements in this function, the calls to assign, creates functions for printing elements of the type we are creating. We will implement the deparse\_construction and construction\_printer functions below. They extract information about values from meta-information we will store in objects of the new type, and we can use the same functions for all types we define in our language. We use them to specialise the toString and print functions for this specific type. The paste0 calls create the names of the specialisations of the generic toString and print functions. The assign function then stores deparse\_construction and construction\_printer under the appropriate names in the environment we get from get\_env(data\_type), i.e., the environment where we define the type.

The expression on the right-hand side of := defines how we construct elements of the new type. We allow there to be more than one way to do this, and we separate the various choices using the or-operator |. This approach resembles how we describe different alternatives when we specify a grammar, so it is a natural choice. To process the right-hand side, we use the function process\_alternatives.

process\_alternatives <- function(constructors,  
 data\_type\_name,  
 env) {  
 if (is\_lang(constructors) && constructors[[1]] == "|") {  
 process\_alternatives(  
 constructors[[2]],  
 data\_type\_name,  
 env  
 )  
 process\_alternatives(  
 constructors[[3]],  
 data\_type\_name,  
 env  
 )  
 } else {  
 process\_constructor(  
 constructors,  
 data\_type\_name,  
 env  
 )  
 }  
}

In addition to the constructor expression, we pass the name of the type and the environment we are defining it in as parameters. We do not use these directly in this function but merely pass them along. We will use them later when we create the actual constructors.

The process\_alternatives function recursively parse the expression to get all alternatives separated by |. The actual constructors will be either a function or a symbol, so the constructor specifications will not have higher precedence than the *or* operator. The first time we see something that isn’t a call to |, then, we have a constructor. We handle those using the process\_constructor function.

process\_constructor <- function(constructor,   
 data\_type\_name,  
 env) {  
 if (is\_lang(constructor))  
 process\_constructor\_function(  
 constructor,  
 data\_type\_name,  
 env  
 )  
 else  
 process\_constructor\_constant(  
 constructor,  
 data\_type\_name,  
 env  
 )  
}

This function figures out if what we are looking at is a function constructor or a constant, i.e., a symbol. We use the is\_lang function to test if we are looking at a function. It does the same as is.call from the base package; I just prefer the rlang functions for this chapter.

Constant constructors are the simplest. They are merely symbols so to make them available for programmers; we need to define a value for each such symbol. We will use NA as the value of these variables and store some meta-information with them. We set the class, so the construction\_printer function will be called when we try to print the object, and we set the attribute constructor\_constant that we will later need for pattern matching.

process\_constructor\_constant <- function(constructor,  
 data\_type\_name,   
 env) {  
 stopifnot(is\_symbol(constructor))  
 constructor\_name <- as\_string(constructor)  
 constructor\_object <- structure(  
 NA,  
 constructor\_constant = constructor\_name,  
 class = data\_type\_name  
 )  
 assign(constructor\_name, constructor\_object, envir = env)  
}

For the function constructors that we need to create, you guessed it, functions. We analyse the arguments given to the constructor specification and build a function out of that, and this function we then store in the environment where the constructor is defined. We permit two kinds of parameters to a constructor: either a symbol or a symbol with a type. For the latter, we use the : operator. If a parameter is a : call, then we consider the left-hand side the parameter and the right-hand side the type. We use the types to guarantee that values we construct are of the expected kind. If there is no type specified, we will allow a parameter to hold any value. We use the following function to translate the list of parameters from a function constructor expression into a data frame where the first column holds the argument names and the second column holds their type. We use NA to indicate that we allow any type. The function works by first translating the arguments—that are in the form of a call object—into a list. We have to use the base as.list function for this, rather than the rlang as\_list, since the latter will not translate call objects into lists. Once we have the arguments as a list, we map the process\_arg function over the elements. This function creates a row for the data frame per element, and we combine the rows using the bind\_rows function from dplyr.

process\_arguments <- function(constructor\_arguments) {  
 process\_arg <- function(argument) {  
 if (is\_lang(argument)) {  
 stopifnot(argument[[1]] == ":")  
 arg <- quo\_name(argument[[2]])  
 type <- quo\_name(argument[[3]])  
 tibble::tibble(arg = arg, type = type)  
 } else {  
 arg <- quo\_name(argument)  
 tibble::tibble(arg = arg, type = NA)  
 }  
 }  
 constructor\_arguments %>%   
 as.list %>%   
 purrr::map(process\_arg) %>%   
 bind\_rows  
}

The process\_constructor\_function translates a function construction specification into a function. The first element of the specifications, which is a call object, is the name of the function. The remaining elements, we translate into a data frame using the function we just saw. After that, we need to create the function that will work as the constructor. Here, we create a closure without arguments and then add formal parameters afterwards, as we did in the previous chapter, and we get the actual parameters that the closure is called with using the as\_list(environment()) trick.

The value we return from the closure is just the list of arguments that are provided to it but tagged with a constructor attribute we can use for pattern matching and a class set to the type we are defining, something we use for type checking. The type checking is the chief part of the constructor. Here, we check that we get the right number of arguments and that they have the right type if a type was specified.

Once we have created the closure and set its formal arguments, we also update its class, so it is both a constructor and a function. Giving constructor functions the class constructor is also something we will need when pattern matching. Then we assign it to the environment associated with the specification to make it available to the programmer.

process\_constructor\_function <- function(constructor,  
 data\_type\_name,  
 env) {  
 stopifnot(is\_lang(constructor))  
 constructor\_name <- quo\_name(constructor[[1]])  
 constructor\_arguments <- process\_arguments(constructor[-1])  
  
 # Create the constructor function  
 constructor <- function() {  
 args <- as\_list(environment())  
  
 # Type check!  
 stopifnot(length(args) == length(constructor\_arguments$arg))  
 for (i in seq\_along(args)) {  
 arg <- args[[constructor\_arguments$arg[i]]]  
 type <- constructor\_arguments$type[i]  
 stopifnot(is\_na(type) || inherits(arg, type))  
 }  
  
 structure(args,  
 constructor = constructor\_name,  
 class = data\_type\_name)  
 }  
 formals(constructor) <- make\_args\_list(constructor\_arguments$arg)  
  
 # Set meta information about the constructor  
 class(constructor) <- c("constructor", "function")  
  
 # Put the constructor in the binding scope  
 assign(constructor\_name, constructor, envir = env)  
}

The only remaining function to write for the constructors is the function for printing them. Here, we write a function that translates a constructed object into a string; this function we can then use recursively to translate any constructed element into a string. We then just call this function in construction\_printer which is the function that is assigned to the specialised print function for any type we define.

There is nothing complicated in the function. We first check if the object has an attribute “constructor”. Strictly speaking, only the function constructors have this—the constant constructors have the attribute “constructor\_constant”, but the attire function will pick an attribute if it gets a unique prefix, so we also get that. If we do not have a “constructor” attribute, then it isn’t an element constructed from something we have defined from our language, so it must be a value of some other type—we just convert it into a string and return this. We use the generic toString function for this. This function converts any object into a string. Not necessarily a beautiful representation of the object, but you can specialise it if you need to.

If the object we have *is* a constructor, it is either a constant or the result of a constructor function call. If the latter, it will be a list. If it is a list, then we must convert all the elements in the list into strings and paste them together. Otherwise, the name of the constructor is the string representation of the object.

deparse\_construction <- function(object) {  
 constructor\_name <- attr(object, "constructor")  
 if (is\_null(constructor\_name)) {  
 # This is not a constructor, so just get the value  
 return(toString(object))  
 }  
  
 if (is\_list(object)) {  
 components <- names(object)  
 values <- as\_list(object) %>% purrr::map(deparse\_construction)  
  
 print\_args <- vector("character", length = length(components))  
 for (i in seq\_along(components)) {  
 print\_args[i] <- paste0(components[i], "=", values[i])  
 }  
 print\_args <- paste0(print\_args, collapse = ", ")  
 paste0(constructor\_name, "(", print\_args, ")")  
  
 } else {  
 constructor\_name  
 }  
}  
construction\_printer <- function(x, ...) {  
 cat(deparse\_construction(x), "\n")  
}

As an example of using the construction language we can define a binary tree as either a tree with a left and right sub-tree or a leaf:

tree := T(left : tree, right : tree) | L(value : numeric)

We can use the constructors to create a tree:

x <- T(T(L(1),L(2)),L(3))  
x

## T(left=T(left=L(value=1), right=L(value=2)), right=L(value=3))

Values we create using these constructors can be accessed just as lists—which, in fact, they are—using the variable names we used in the type specification:

x$left$left$value

## [1] 1

x$left$right$value

## [1] 2

x$right$value

## [1] 3

The type checking is rather strict, however. We demand that the values we pass to the constructor functions are of the types we give in the specification—in the sense that they must inherit the class from the specification—and this can be a problem in some cases where R would otherwise ordinarily just convert values. In the specification for the L constructor, for example, we require that the argument is numeric. We will get an error if we give it an integer:

L(1L)

## Error: is\_na(type) || inherits(arg, type) is not TRUE

This situation is where we can use the variant of parameters without a type:

tree := T(left : tree, right : tree) | L(value)  
L(1L)

## L(value=1)

An alternative solution could be to specify more than one type in the specification. If you are interested, you can play with that. I will just leave it here and move on to pattern matching.

## Pattern matching

We want to implement pattern matching such that an expression like this

match(L(1),  
 L(v) -> v,  
 T(L(v), L(w)) -> v + w,  
 otherwise -> 5)

should return 1, since the pattern L(v) matches the value L(1) and we return v, which we expect to be bound to 1. Likewise, we want this expression to return nine since v should be bound to 4 and w to 5 and we return the result of evaluating v + w.

match(T(L(4), L(5)),  
 L(v) -> v,  
 T(L(v), L(w)) -> v + w,  
 otherwise -> 5)

## Error in as.list.environment(x): object 'v' not found

We want the otherwise keyword to mean anything at all and use it as a default pattern, so in this expression, we want to return five.

match(T(L(1), T(L(4), L(5))),  
 L(v) -> v,  
 T(L(v), L(w)) -> v + w,  
 otherwise -> 5)

The syntax for pattern matching uses the right-arrow operator. This operator is usually an assignment. We cannot specialise arrow assignments, but we can still use them in a meta-programming function. We use an assignment operator for the same reasons as we had for using the := operator for defining types. Since assignment operators have the lowest precedence, we don’t have to worry about how tight the operators to the left and right of the operator binds. We could also have used that operator here, but I like the arrow more for this function. It shows us what different patterns map to. You need to be careful with the -> operator, though, since it is syntactic sugar for <-. This means that once we have an expression that uses ->, we will actually see a call to <- and the left- and right-hand sides will be switched.

The match function will take a variable number of arguments. The first is the expression we match against, and the rest are captured by the three-dots operator. The expressions there should not be evaluated directly, so we capture them as quosures. We then iterate through them, split them into left-hand and right-hand sides, and test the left-hand side against the expression. The function we use for testing the pattern will return an environment that contains bound variables if it matches, and NULL otherwise. If we have a match, we evaluate the right-hand side in the quosure environment over-scoped by the environment we get from matching the pattern.

match <- function(expr, ...) {  
 matchings <- quos(...)  
  
 for (i in seq\_along(matchings)) {  
 eval\_env <- get\_env(matchings[[i]])  
 match\_expr <- quo\_expr(matchings[[i]])  
 stopifnot(match\_expr[[1]] == "<-")  
  
 test\_expr <- match\_expr[[3]]  
 result\_expr <- match\_expr[[2]]  
  
 match <- test\_pattern(expr, test\_expr, eval\_env)  
 if (!is\_null(match))  
 return(eval\_tidy(result\_expr, data = match, env = eval\_env))  
 }  
  
 stop("No matching pattern!")  
}

In the test\_pattern function we create the environment where we will bind matched variables. If the pattern is otherwise, we return the empty environment—no variables are bound there. Otherwise, we need to explore both pattern and expression recursively.

We use the function test\_pattern\_rec to do this, but we do not call it directly. Instead, we use a function called callCC. The name stands for *call with current continuation*, and it is a function that sometimes causes some confusion for people not intimately familiar with functional programming. There is no need for this confusions, however, because all the function does is provide us with a way to return to the point where we called callCC.

We wrap the test\_pattern\_rec function in a closure, tester, that is called with a function that we call escape. This is the function that callCC will provide. If, at any point, we call the function escape, it will terminate whatever we are doing and return to the point where we called callCC. This means that we can use escape to get out of deep recursions if we find out at some point that the pattern doesn’t match the expression. We do not need to propagate a failed match up the call stack through the recursive function calls. As soon as we call escape we are taken back to the end of test\_pattern. Whatever we called escape with—it is a function of a single parameter—will be the return value of the callCC call. So, if we find that a pattern doesn’t match, we will call escape with NULL. This will then be the result of test\_pattern. If we never call escape, but instead return normally from test\_pattern\_rec, then what we return from that function will also be the return value of the callCC call. So, if we match the pattern and return an environment from test\_pattern\_rec, this will also be the return value of the test\_pattern call.

test\_pattern <- function(expr, test\_expr, eval\_env) {  
 # Environment in which to store matched variables  
 match\_env <- env()  
  
 if (test\_expr == quote(otherwise))  
 return(match\_env)  
  
 # Test pattern  
 tester <- function(escape)  
 test\_pattern\_rec(escape, expr, test\_expr,   
 eval\_env, match\_env)  
 callCC(tester)  
}

It is in test\_pattern\_rec the real work is done. It analyses the pattern expression, stored in the test\_expr variable, and matches it against the value stored in the expr variable. It also takes two environments as parameters, one is the environment where the expression and pattern are defined—it needs this environment to look up variables to check what they are—and the other is the environment in which it should bind variables from the pattern. It, of course, also knows the escape function that it can use if it finds out that the pattern isn’t matching.

test\_pattern\_rec <- function(escape, expr, test\_expr,  
 eval\_env, match\_env) {  
  
 # Is this a function-constructor?  
 if (is\_lang(test\_expr)) {  
 func <- get(as\_string(test\_expr[[1]]), eval\_env)  
   
 if (inherits(func, "constructor")) {  
 # This is a constructor.  
 # Check if it is the right kind  
 constructor <- as\_string(test\_expr[[1]])  
 expr\_constructor <- attr(expr, "constructor")  
 if (is\_null(expr\_constructor) ||   
 constructor != expr\_constructor)  
 escape(NULL) # wrong type  
  
 # Now check recursively  
 for (i in seq\_along(expr)) {  
 test\_pattern\_rec(  
 escape,   
 expr[[i]], test\_expr[[i+1]],  
 eval\_env, match\_env  
 )  
 }  
  
 # If we get here, the matching was successfull  
 return(match\_env)  
 }  
 }  
  
 # Is this a constant-constructor?  
 if (is\_symbol(test\_expr) &&   
 exists(as\_string(test\_expr), eval\_env)) {  
 constructor <- as\_string(test\_expr)  
 val <- get(constructor, eval\_env)  
 val\_constructor <- attr(val, "constructor\_constant")  
 if (!is\_null(val\_constructor)) {  
 expr\_constructor <- attr(expr, "constructor")  
 if (is\_null(expr) || constructor != expr\_constructor)  
 escape(NULL) # wrong type  
 else  
 return(match\_env) # Successfull match  
 }  
 }  
  
 # Not a constructor.  
 # Must be a value to compare with or a variable to bind to  
 if (is\_symbol(test\_expr)) {  
 assign(as\_string(test\_expr), expr, match\_env)  
 } else {  
 value <- eval\_tidy(test\_expr, eval\_env)  
 if (expr != value) escape(NULL)  
 }  
  
 match\_env  
}

There are three cases to consider: The test\_expr is a constructor function, a constructor constant, or something else.

If test\_expr is a function call, we test this using is\_lang from rlang, then it might be a function constructor. To figure out if it is, we look the function name up in the evaluation environment, i.e., the environment where the test pattern was written. We then test if the function inherits "constructor". The functions we create in our DSL will also be constructor objects, so if it does, we know we have such one. We then check if expr has an attribute "constructor". If it was generated by a call to a constructor, it will. If it doesn’t, then we cannot have a match and we escape with NULL. We also escape if the names of the constructors do not match. If they do, we iterate through all the elements in the pattern and expression calls and attempt to match these. If they do not match, we will never return from a recursive call—they will have used the escape function to jump directly to the callCC point in test\_pattern. If they return, the pattern did match the expression, and we return the match\_env that now contains any variables that were bound in the matching.

If test\_expr is not a function call, it might still be a constructor. If it is, then it will be a symbol and the symbol will be a variable in the eval\_env scope—we test this with the exists function. These two tests only tell us that there is a variable with the name from test\_expr in eval\_env. Not that it is a constant constructor. To test this, we get the value the variable return to, using the get function, and checks if it has an attribute called "constructor\_constant". If it does, then that is the name of the constructor, and we can test that against the value in expr that will either be the object that represents the constant—in which case we have a match—or it will be something else—in which case we escape.

If we get past the first two tests, we do not have a constructor. We now either have a value that we must test against expr, or we have a variable that we should bind to the value of expr. Here, I have decided that any symbol will be interpreted as a variable that should be bound and anything else should be a value that we evaluate and compared against expr. We could also have checked if the variable was bound and used its value in that case, but that could very clearly lead to hard-to-fix bugs. In any case, you can always use quasi-quoting to achieve the same effect.

x <- 1  
y <- 2  
match(L(1),  
 L(!!x) -> "x",  
 L(!!y) -> "y")

## [1] "x"

So, we bind any variable by assigning the value of expr to the symbol in the match\_env environment. Anything that isn’t a symbol should be a value that is the same as expr. To get this value, we evaluate the test\_expr in the eval\_env.

That was the entire implementation of pattern matching. It might not be trivial code, but it is not horribly complicated either, and we have created a very efficient language in less than 200 lines of code.

We can try it out, now, by implementing a depth-first traversal of the binary tree type we defined earlier. The function below is a simple traversal that adds together all the values found in leaves. The match call consider the basis case—a leaf—and the recursive case—a tree with a left and right sub-tree—and doesn’t need an otherwise case.

dft <- function(tree) {  
 match(tree,  
 L(v) -> v,  
 T(left, right) -> dft(left) + dft(right))  
}  
  
dft(L(1))

## [1] 1

dft(T(L(1),L(2)))

## [1] 3

dft(T(T(L(1),L(2)),L(3)))

## [1] 6

## Lists

We have used linked lists many places in this book, but the functions we have used had to access the list elements in a list. We can define linked lists using our new language like this:

linked\_list := NIL | CONS(car, cdr : linked\_list)

A list is either empty, we use the constant NIL to represent that, or it has a head element, car, and a tail, cdr. Since we implement values we construct from our language as list objects, we automatically get the linked-list implementation form this specification.

With pattern matching, we can write very simple functions for manipulating lists. For example, the following function reverses a list using an accumulator list. In the base case, when the first list is empty, we return the accumulator. Otherwise, we take the head of the list and prepend it to the accumulated and then recurse. We force evaluation of the accumulator to avoid lazy evaluation. With lazy evaluation, we would be building larger and later CONS expressions that would not be evaluated until the end of the recursion. At this point, we might have built the expression too large for the stack space needed to evaluate the functions (see, e.g. Mailund (2017b) and Mailund (2017a) for explanations of how recursion and lazy evaluation can collide to exceed the stack space).

reverse\_list <- function(lst, acc = NIL) {  
 force(acc)  
 match(lst,  
 NIL -> acc,  
 CONS(car, cdr) -> reverse\_list(cdr, CONS(car, acc)))  
}

We can write a very similar function to compute the length of a list. This will follow the same pattern of using an accumulator, which we return in the base case and update in the recursive case.

list\_length <- function(lst, acc = 0) {  
 force(acc)  
 match(lst,  
 NIL -> acc,  
 CONS(car, cdr) -> list\_length(cdr, acc + 1))  
}

A function for translating a linked list into a list object is a little more involved, but can still be written succinctly using pattern matching. We need to figure out the length of the list object first, then allocate it, and finally iterate through the linked list to update the list. We use a closure, f, to recursively traverse the linked list. In the base case, I return NULL. It doesn’t matter what we return here since we only do the recursion for its side-effects, which are handled in the recursive case. Here, I use a code block—the curly-braces operator—to evaluate two statements. The first update the list v and the second continue the recursion. It is precisely because we use tidy evaluation that this function works. It is essential that the assignment we do in the recursive case is evaluated in the environment where we write the expression. Otherwise, we would not be updating the correct list.

list\_to\_vector <- function(lst) {  
 n <- list\_length(lst)  
 v <- vector("list", length = n)  
 f <- function(lst, i) {  
 force(i)  
 match(lst,  
 NIL -> NULL,  
 CONS(car, cdr) -> {  
 v[[i]] <<- car  
 f(cdr, i + 1)  
 }  
 )  
 }  
 f(lst, 1)  
 v %>% unlist  
}

Translating a list to a linked list is simple enough and doesn’t use any pattern matching—we cannot pattern match on list objects, after all.

vector\_to\_list <- function(vec) {  
 lst <- NIL  
 for (i in seq\_along(vec)) {  
 lst <- CONS(vec[[i]], lst)  
 }  
 reverse\_list(lst)  
}

With these few functions, we can translate to and from vectors and work with lists.

lst <- vector\_to\_list(1:5)  
list\_length(lst)

## [1] 5

list\_to\_vector(lst)

## [1] 1 2 3 4 5

lst %>% reverse\_list %>% list\_to\_vector

## [1] 5 4 3 2 1

Extending the functionality of linked lists with additional functions, I will leave as an exercise to the interested reader. You can experiment to your heart’s desire.

## Search trees

As another example, we can implement search trees. These are trees, containing ordered values, that satisfy the recursive property that all elements in the left sub-tree of a search tree will have values less than the value stored at the root of the tree, and all elements in the right sub-tree of a search tree will have values larger than the value in the root.

To define search trees, we try a different approach than the binary trees from earlier. We define an empty tree, E, and a tree with two sub-trees, left and right, and a value.

search\_tree :=   
 E | T(left : search\_tree, value, right : search\_tree)

We will just implement two functions for the example, insertion and test for membership. For more functions on search tree, I will refer to Mailund (2017a, chap. 6).

Both functions search recursively down the tree until they either find the value they want to insert or want to check is in the three, respectively. For insertion, the base case, when it hits a leaf, is to insert the element there. It will create a tree, with two empty sub-trees, containing the value. The recursive function builds a tree in the recursion by using the T constructor to create new trees in each recursive call. Thus, the tree created at a leaf will be put into the updated tree that the insert function creates. For the member function, we do not need to update the tree. If we hit a leaf, we know that the element is not in the tree and we can just return FALSE. In both functions, the search checks the value in the tree they see in the recursive case. If the value there is greater than x, then the only place x could be found would be in the left sub-tree, so we continue the search there. If, on the other hand, x is greater than the value, then we must search in the right sub-tree. If it is neither smaller than or greater than the value, it must be equal to the value. For insertion, this means we do not have to do anything, and we can just return the tree that already contains x. For the membership test, we can return TRUE.

insert <- function(tree, x) {  
 match(tree,  
 E -> T(E, x, E),  
 T(left, val, right) -> {  
 if (x < val)  
 T(insert(left, x), val, right)  
 else if (x > val)  
 T(left, val, insert(right, x))  
 else  
 T(left, x, right)  
 })  
}  
  
member <- function(tree, x) {  
 match(tree,  
 E -> FALSE,  
 T(left, val, right) -> {  
 if (x < val) member(left, x)  
 else if (x > val) member(right, x)  
 else TRUE  
 })  
}

We can build a tree like this:

tree <- E  
for (i in sample(2:4))  
 tree <- insert(tree, i)

Once built, we can test membership like this:

for (i in 1:6) {  
 cat(i, " : ", member(tree, i), "\n")  
}

## 1 : FALSE   
## 2 : TRUE   
## 3 : TRUE   
## 4 : TRUE   
## 5 : FALSE   
## 6 : FALSE

The worst-case time usage for both of these functions is proportional to the depth of the tree, and that can be linear in the number of elements stored in the tree. If we keep the tree balanced, though, the time is reduced to logarithmic in the size of the tree. A classical data structure for keeping search trees balanced is so-called *red-black* search trees. Implementing these using pointer or reference manipulation in languages such as C/C++ or Java can be quite challenging, but in a functional language, balancing such trees is a simple matter of transforming trees based on local structure, see, e.g. Okasaki (1999), Germane and Might (2014), or Mailund (2017a).

Red-black search trees are binary search trees where each tree has a colour associated, either red or black. We can define colours using constructors like this:

colour :=  
 R | B

We add a colour to all non-empty trees like this:

rb\_tree :=  
 E | T(col : colour, left : rb\_tree, value, right : rb\_tree)

Except for including the colour in the pattern matching, the member function for this data structure is the same as for the plain search tree.

member <- function(tree, x) {  
 match(tree,  
 E -> FALSE,  
 T(col, left, val, right) -> {  
 if (x < val) member(left, x)  
 else if (x > val) member(right, x)  
 else TRUE  
 })  
}  
  
tree <- T(R, E, 2, T(B, E, 5, E))  
for (i in 1:6) {  
 cat(i, " : ", member(tree, i), "\n")  
}

## 1 : FALSE   
## 2 : TRUE   
## 3 : FALSE   
## 4 : FALSE   
## 5 : TRUE   
## 6 : FALSE

What keeps red-black search trees balanced is that we always enforce these two invariants:

1. No red node has a red parent.
2. Every path from the root to a leaf has the same number of black nodes.

If every path from root to a leaf has the same number of black nodes, then the tree is perfectly balanced if we ignored the red nodes. Since no red node has a red parent, the longest path, when red nodes are considered, can be no longer than twice the length of the shortest path.

These invariants can be guaranteed by always inserting new values in red leaves, potentially invalidating the first invariant, and then rebalancing all sub-trees that invalidate this invariant, and at the end setting the root to be black. The rebalancing is done when returning from the recursive insertion calls that otherwise work as insertion in the plain search tree.

insert\_rec <- function(tree, x) {  
 match(tree,  
 E -> T(R, E, x, E),  
 T(col, left, val, right) -> {  
 if (x < val)  
 balance(T(col, insert\_rec(left, x), val, right))  
 else if (x > val)  
 balance(T(col, left, val, insert\_rec(right, x)))  
 else  
 T(col, left, x, right) # already here  
 })  
}  
insert <- function(tree, x) {  
 tree <- insert\_rec(tree, x)  
 tree$col <- B  
 tree  
}

The transformation rules for the balance function are shown in fig. 1. Whenever we see any of the four trees on the edges, we have to transform it into the one in the middle. The implementation I presented in Mailund (2017a) contained mostly code for testing the structure of the tree to match and very little to construct the modified tree. With pattern matching, we can implement these rules by matching for each of the four cases like this:

![Figure 1: Re-balancing transformations when inserting into a red-black search tree.](data:application/pdf;base64,)

Figure 1: Re-balancing transformations when inserting into a red-black search tree.

balance <- function(tree) {  
 match(tree,  
 T(B,T(R,a,x,T(R,b,y,c)),z,d) -> T(R,T(B,a,x,b),y,T(B,c,z,d)),  
 T(B,T(R,T(R,a,x,b),y,c),z,d) -> T(R,T(B,a,x,b),y,T(B,c,z,d)),  
 T(B,a,x,T(R,b,y,T(R,c,z,d))) -> T(R,T(B,a,x,b),y,T(B,c,z,d)),  
 T(B,a,x,T(R,T(R,b,y,c),z,d)) -> T(R,T(B,a,x,b),y,T(B,c,z,d)),  
 otherwise -> tree)  
}

This is the function we used to motivate the domain-specific language, and so we come full circle, having implemented the language we wanted.

Germane, K, and M Might. 2014. “Deletion: The curse of the red-black tree.” *Journal of Functional Programming* 24 (04): 423–33.

Mailund, Thomas. 2017a. *Functional Data Structures in R*. Advanced Statistical Programming in R. Apress.

———. 2017b. *Functional Programming in R*. Advanced Statistical Programming for Data Science, Analysis and Finance. Apress.

Okasaki, C. 1999. “Red-Black Trees in a Functional Setting.” *Journal of Functional Programming* 9 (4). Cambridge University Press: 471–77.