R: Higher-order functions and their types

Sławek Staworko

Univ. Lille 3

2018

Outline

What is functional programming?

Functions in R

Use case: Map/Reduce

Type systems

How to type functions?

Functional programming paradigm

Functional programming

A style of writing programs that views computation as an evaluation of an expression with functions (mathematical)

- **side-effect free** – function returns the same result for the same arguments (no change in the state of the environment)
- **immutable data structure** – once created cannot be modified (but a modified "copy" can be created)
- **function are first-class citizens** – functions can be arguments of other functions and can be returned as results

Typically, FP has extensive support for list processing

```
quicksort [] = []
quicksort (x:xs) = quicksort small ++ [x] ++ quicksort large
  where small = [y | y <- xs, y <= x]
        large = [y | y <- xs, y > x]
```

R as a functional programming language

R is not purely functional

R combines elements of declarative and imperative programming

- functions are first-class citizens
- data is immutable but functions may have side-effects

Declarative programming

The output of a program is specified using expressions that specify what the output should be

+ Less programming errors
+ No concurrency issues (multi-processor environments)

Imperative programming

The output of program is specified using instructions that specify how the output should be calculated

+ Efficient code is easier to write

Variables in R

Variables

A variable is a name with an associated value (an object).

Example

We define a variable by assigning a value to it

- `x <- 2`
- `y <- x + 3`

And we can then use it in other expressions

- `x * y` $\mapsto$ `2 + 5` $\mapsto$ `10`

We have to use only variables that have already been defined

- `x + z` $\mapsto$ `error`

What is a function?

Function

is an object that takes an object and returns another object.

Example

- `sqrt(2.0)` $\mapsto$ `1.414214...`
- `substr("John Smith",6,10)` $\mapsto$ `"Smith"`
- `sort(`$\langle$`1,3,1,2`$\rangle$`)` $\mapsto$ $\langle$`1,1,2,3`$\rangle$
- `unique(`$\langle$`1,3,1,2`$\rangle$`)` $\mapsto$ $\langle$`1,3,2`$\rangle$
- `paste("John","Smith")` $\mapsto$ `"John Smith"`
- `nchar("Smith")` $\mapsto$ `5`
- `nchar(substr(paste("John","Smith"),6,10))` $\mapsto$ `5`

Functions in R

Defining a function on the spot

`function (`*vars*`)` *expr*

Example

- `square` $\leftarrow$ `function (x) x^2`
- `volume` $\leftarrow$ `function (a,b,c) a*b*c`

Function application (calling a function)

Substitute the arguments by supplied values

- `square(3)` $\mapsto$ `3^2` $\mapsto$ `9`
- `volume(2,3,5)` $\mapsto$ `2*3*5` $\mapsto$ `30`
- `(function (x) x+2)(4)` $\mapsto$ `4+2` $\mapsto$ `6`

Number of arguments must agree with the definition

- `volume(2,3)` $\mapsto$ `error`

Functions as first-class citizens

Higher-order function

A *higher-order function* (a.k.a *functor*) is a function that takes another function as an argument or returns a function.

Example

A function that takes another function as an argument

- `apply` $\leftarrow$ `function (f,`$\langle$`x,y,z`$\rangle$`)` $\langle$`f(x),f(y),f(z)`$\rangle$
- `apply(square,`$\langle$`1,3,2`$\rangle$`)` $\mapsto$ $\langle$`1,9,4`$\rangle$
- `apply(function (x) x+1,`$\langle$`1,3,2`$\rangle$`)` $\mapsto$ $\langle$`2,4,3`$\rangle$
- `apply(nchar,`$\langle$`"Hello","Ah","Boom"`$\rangle$`)` $\mapsto$ $\langle$`5,2,4`$\rangle$

Functions as first-class citizens

Example

A function that returns a function

- `add` $\leftarrow$ `function (x) { function (y) { x + y } }`

This function can be used to generate other functions

- `succ` $\leftarrow$ `add(1)` (= `function (y) 1 + y`)
- `pred` $\leftarrow$ `add(-1)` (= `function (y) -1 + y`)

Which can be used independently

- `succ(2)` $\mapsto$ `1 + 2` $\mapsto$ `3`
- `prec(3)` $\mapsto$ `-1 + 3` $\mapsto$ `2`

We can also call `add` as follows

- add(2)(3) $\mapsto$ (function (y) 2 + y)(3) $\mapsto$ 2+3 $\mapsto$ 5

But not like this

- add(2,3) $\mapsto$ error

Deferred computation

Curried functions

Sometimes it is more useful to work with functions that take their arguments one by one rather than functions that take all arguments at once.

Example

- `apply` $\leftarrow$ `function (f) function (`$\langle$`x,y,z`$\rangle$`)` $\langle$`f(x),f(y),f(z)`$\rangle$
- `inc_triple` $\leftarrow$ `apply(function (x) x + 1)`
- `inc_triple(`$\langle$`3,1,2`$\rangle$`)` $\mapsto$ $\langle$`4,2,3`$\rangle$
- `square_triple` $\leftarrow$ `apply(square)`
- `square_triple(`$\langle$`3,1,2`$\rangle$`)` $\mapsto$ $\langle$`9,1,4`$\rangle$

Currying

There is a function that transforms a function taking a pair to its curried version

```
curry ← function (f) {
    function (x) {
        function (y) {
            f(x,y)
        }
    }
}
```

Example

- `plus ← function (x,y) x + y`
- `add ← curry(plus)`
  `(add = function (x) function (y) x + y)`

Uncurrying

The conversion in the other direction is also possible

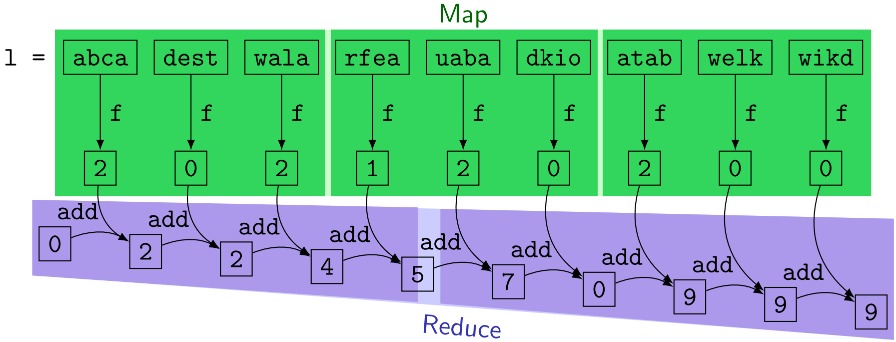
```
uncurry ← function (f) {
    function (x,y) {
        f(x)(y)
    }
}
```

Example

- `add ← function (x) function (y) x + y`
- `plus ← uncurry(plus)`
  `(plus = function (x,y) x + y)`

Use case: Map/Reduce

```
f ← function(s) str_count(s,"a")
add ← function(acc,c) acc+c
```




```
reduce(map(l,f),add,0)
```

Type systems

Type system associates with every object a property called type.

Example

`2.5` is a number, `"abc"` is a string (of characters), `exp` is a function that takes a number and returns a number.

Type errors

Errors caused by the discrepancy between the types of data as opposed to the types expected by a function (logic errors).

Example

`exp(2.5)` is error-free while `exp("abc")` has a type error because it uses a string where a number is expected.

Function type

Elementary knowledge of what the function does

R is dynamically but not statically typed

Static typing

- every object (including functions) has a type
- types might be inferred or may need to be declared
- type enforcement at compile time guarantees an error-free execution (strong type safety)
- type conversions often need to be explicit

Dynamic typing

- types of functions is not check at compile time so there is no need to declare them
- run time errors are raised if a function is called with the wrong type of an argument
- correctness of code is verified using test cases (unit testing)
- type conversions may implicit

Functions and types

Function type

1. what kind of objects a function takes
2. what kind of object it produces

Example

- `sqrt(2.0)` $\mapsto$ `1.414214...`
- `substr("abcdef",2,4)` $\mapsto$ `"bcd"`
- `unique(`$\langle$`1,3,1,2`$\rangle$`)` $\mapsto$ $\langle$`1,3,2`$\rangle$

- `substr` takes a string and two integers and returns a string
- `sqrt` takes a real number and returns a real number
- `unique` takes a list of numbers and return a list of numbers

ML-like type system for R

Atomic types

`log` `logical` – two Boolean values `FALSE` and `TRUE`

`num` `numeric` – floating-point numeric values, `0.1`, $\sqrt{2}$, π;
(the default computational data type, in double precision)

`int` `integer` – positive and negative integers `0`,`1`,`2`,…,`-1`,`-2`,…
In R we need to use `L` prefix to force it e.g., -30`L`.

`chr` `character` – characters and strings

`raw` `raw` – binary objects of arbitrary size

ML-like type system for R

Structural types

tuples a sequence of elements of various types

- $\texttt{chr} \times \texttt{int} \times \texttt{int}$ – triples of one string and two integers
- $\texttt{complex} = \texttt{num} \times \texttt{num}$ – complex numbers, where $\pi + \sqrt{2}i$ is represented as $\langle \pi, \sqrt{2} \rangle$.

vectors collections of the same type of a arbitrary length

- $\texttt{int}^*$ – vectors of integers
- $\texttt{chr}^*$ – vectors of strings

Tuples as fixed-size vectors

$\texttt{int}^3 = \texttt{int} \times \texttt{int} \times \texttt{int}$ is the type of

- triples of integers
- integer vectors of length 3

In general,

$$\texttt{int}^* = \texttt{int}^0 \cup \texttt{int}^1 \cup \texttt{int}^2 \cup \texttt{int}^3 \cup \dots$$

ML-like type system for R

Function f has type $T \to S$ if

is takes an object of type T and returns an object of type S

Example

- `sqrt(2.0)` $\mapsto$ `1.414214...`
- `substr("abcdef",2,4)` $\mapsto$ `"bcd"`
- $\texttt{unique}(\langle\texttt{1,3,1,2}\rangle) \mapsto \langle\texttt{1,3,2}\rangle$
- $\texttt{sqrt} : \texttt{num} \to \texttt{num}$
- $\texttt{substr} : \texttt{chr} \times \texttt{int} \times \texttt{int} \to \texttt{chr}$
- $\texttt{unique} : \texttt{num}^* \to \texttt{num}^*$

$\to$ is right-associative (grouped from the right)

$X \to Y \to Z$ is $X \to (Y \to Z)$ and not $(X \to Y) \to Z$

ML-like type system for R

Example

Some functions

- `sum(`$\langle$`3,2,5,7,2,5,8`$\rangle$`)` $\mapsto$ `32`
- `2.1 + 3.2` $\mapsto$ `5.3`
- `floor(2.8)` $\mapsto$ `2`
- `paste("John","Smith")` $\mapsto$ `"John Smith"`
- `nchar("John")` $\mapsto$ `4`

and their types

- $\texttt{sum} : \texttt{num}^* \to \texttt{num}$
- $\text{`}\texttt{+}\text{`} : \texttt{num} \times \texttt{num} \to \texttt{num}$
- $\texttt{floor} : \texttt{num} \to \texttt{int}$
- $\texttt{paste} \ : \ \texttt{chr} \times \texttt{chr} \to \texttt{chr}$
- $\texttt{nchar} \ : \ \texttt{chr} \to \texttt{num}$

ML-like type system for R

Identity function

- `id` $\leftarrow$ `function (x) x`

It takes an object and returns an object of precisely the same type

Polymorphic types $\alpha, \beta, \gamma, \ldots$

If nothing is known about a type, we can use universal types to constraint the types

$$\texttt{id} : \alpha \rightarrow \alpha$$

While we do not know anything about the type α, we know that `id` returns an object or precisely the same type it takes as an argument:

- `id(1.0)` $\mapsto$ `1.0`
- `id("abc")` $\mapsto$ `"abc"`

ML-like type system for R

A function that reverses a vector

- $\texttt{rev}(\langle 1,2,3\rangle) \mapsto \langle 3,2,1\rangle$
- $\texttt{rev}(\langle \texttt{"a"},\texttt{"b"},\texttt{"c"},\texttt{"d"}\rangle) \mapsto \langle \texttt{"d"},\texttt{"c"},\texttt{"b"},\texttt{"a"}\rangle$

A function that returns the first element of a vector

- $\texttt{head}(\langle 1,2,3\rangle) \mapsto 1$
- $\texttt{head}(\langle \texttt{"a"},\texttt{"b"},\texttt{"c"},\texttt{"d"}\rangle) \mapsto \texttt{"a"}$

A function that measures the length of a vector

- $\texttt{length}(\langle 1,2,3\rangle) \mapsto 3$
- $\texttt{length}(\langle \texttt{"a"},\texttt{"b"},\texttt{"c"},\texttt{"d"}\rangle) \mapsto 4$

Their types are:

- $\texttt{rev} : \alpha^* \to \alpha^*$
- $\texttt{head} : \alpha^* \to \alpha$
- $\texttt{length} : \alpha^* \to \texttt{int}$

Typing functions from definition

Given the following type assertions

- `sum` : $\texttt{num}^* \rightarrow \texttt{num}$
- `head` : $\alpha^* \rightarrow \alpha$
- `paste` : $\texttt{chr} \times \texttt{chr} \rightarrow \texttt{chr}$
- `‘+‘` : $\texttt{num} \times \texttt{num} \rightarrow \texttt{num}$

find the type of the functions defined as follows

- `shout` ← `function (x) paste(x,"!")`
- `f` ← `function (x,y) x + sum(y)`
- `g` ← `function (x,y) paste(head(x), y)`

The function types are

- `shout` : $\texttt{chr} \rightarrow \texttt{chr}$
- `f` : $\texttt{num} \times \texttt{num}^* \rightarrow \texttt{num}$
- `g` : $\texttt{chr}^* \times \texttt{chr} \rightarrow \texttt{chr}$

Typing higher-order functions

Given the following type assertions

- $\texttt{sum} : \texttt{num}^* \to \texttt{num}$
- $\texttt{length} : \alpha^* \to \texttt{int}$
- $\texttt{`/`} : \texttt{num} \times \texttt{num} \to \texttt{num}$
- $\texttt{nchar} : \texttt{chr} \to \texttt{int}$

infer the type of the functions

- F $\leftarrow$ `function (f,x) sum(x)/f(x)`
- G $\leftarrow$ `function (g,x) sum(g(len(x))`
- H $\leftarrow$ `function (h,x) h(nchar(x))/2`

The function types are

- $\texttt{F} : (\texttt{num}^* \to \texttt{num}) \times \texttt{num}^* \to \texttt{num}$
- $\texttt{G} : (\texttt{num} \to \texttt{num}^*) \times \texttt{num}^* \to \texttt{num}$
- $\texttt{H} : (\texttt{int} \to \texttt{num}) \times \texttt{chr} \to \texttt{num}$

Typing higher-order functions (contd.)

Example

```
power ← function (y) function (x) x^y
square ← power(2)
cube ← power(3)
square(2) ↦ 4
cube(2) ↦ 8
```

What is the type of `power`?

```
square :  num → num
cube :  num → num
power :  num → num → num
```

Typing higher-order functions (contd.)

Typing curried apply function

- `apply` $\leftarrow$ `function (f) function (`$\langle$`x,y,z`$\rangle$`)` $\langle$`f(x),f(y),f(z)`$\rangle$
- `square_triple` $\leftarrow$ `apply(square)`
- `square_triple(`$\langle$`3,1,2`$\rangle$`)` $\mapsto$ $\langle$`9,1,4`$\rangle$
- `nchar_triple` $\leftarrow$ `apply(nchar)`
- `nchar_triple(`$\langle$`"Hello","Ah","Boom"`$\rangle$`)` $\mapsto$ $\langle$`5,2,4`$\rangle$

The types are

- `square` : `num` $\rightarrow$ `num`
- `nchar` : `chr` $\rightarrow$ `int`
- `square_triple` : $\texttt{num}^3 \rightarrow \texttt{num}^3$
- `nchar_triple` : $\texttt{chr}^3 \rightarrow \texttt{int}^3$
- `apply` : $(\alpha \rightarrow \beta) \rightarrow \alpha^3 \rightarrow \beta^3$

Typing higher-order functions (contd.)

Recall the apply function

- apply $\leftarrow$ function (f,$\langle$x,y,z$\rangle$) $\langle$f(x),f(y),f(z)$\rangle$
- apply(id,$\langle$3,2,5$\rangle$) $\mapsto$ $\langle$3,2,5$\rangle$
- apply(square,$\langle$3,2,5$\rangle$) $\mapsto$ $\langle$4,9$\rangle$
- shout $\leftarrow$ function (s) paste(s,"!")
- apply(shout,$\langle$"a","b","c"$\rangle$) $\mapsto$ $\langle$"a !","b !","c !"$\rangle$
- apply(nchar,$\langle$"Hello","Ah","Boom"$\rangle$) $\mapsto$ $\langle$5,2,4$\rangle$

Its type is

- apply : $(\alpha \rightarrow \beta) \times \alpha^3 \rightarrow \beta^3$

Typing higher-order functions (contd.)

What is the type of the `curry` function

```
curry ← function (f) {
    function (x) {
        function (y) {
            f(x,y)
        }
    }
}
```

`curry` : $(\alpha \times \beta \to \gamma) \to \alpha \to \beta \to \gamma$

Typing higher-order functions (contd.)

And the uncurry function

```
uncurry ← function (f) {
    function (x,y) {
        f(x)(y)
    }
}
```

$\texttt{uncurry} : (\alpha \rightarrow \beta \rightarrow \gamma) \rightarrow \alpha \times \beta \rightarrow \gamma$

Types in Map/Reduce

General schema

$$\texttt{reduce(map(}\langle \texttt{x}_1\texttt{,...,x}_n \rangle\texttt{,f),add,0)}$$

$$\downarrow$$

$$\texttt{reduce(}\langle \texttt{f(x}_1\texttt{),...,f(x}_n\texttt{)} \rangle\texttt{,add,0)}$$

$$\downarrow$$

$$\texttt{add(...add(add(0,f(x}_1\texttt{)),f(x}_2\texttt{)),...,f(x}_n\texttt{))}$$

Types are

- `map` : $\alpha^* \times (\alpha \to \beta) \to \beta^*$
- `reduce` : $\beta^* \times (\gamma \times \beta \to \gamma) \times \gamma \to \gamma$