



Introduction to Functional Programming in R

Andrei Serjantov

12 August 2015

Client's logo
if needed

Client's logo
if needed

Contents

- Introduction to Functional Programming
 - Static Typing
 - Functions as first class objects
- Higher Order Programming
 - Map
 - Fold
- Partial Function Application
- R and Haskell

Client's logo
if needed

Functional Languages

- Static Typing
- Functions as first class “citizens”
 - Passing functions as arguments
 - Returning functions as results (partially applying functions)
- (often involves lots of recursion)
- Examples
 - ML (OCAML)
 - Haskell
 - Scheme

Static Typing (vs Dynamic Typing)

```
> 1+"c"
```

Error in 1 + "c" : non-numeric argument to binary operator

```
> if(FALSE) {1+"c"} else {print("c")}
```

```
[1] "c"
```

```
b = if False then (1+"c") else 1
```

```
perm.hs:82:19:
```

```
  No instance for (Num [Char])
```

```
    arising from the literal `1' at perm.hs:82:19
```

```
Possible fix: add an instance declaration for (Num [Char])
```

```
In the first argument of `(+)', namely `1'
```

```
In the expression: (1 + "c")
```

```
In the expression: if False then (1 + "c") else 1
```

Client's logo
if needed

Functions as “First Class Objects”

$\text{add} :: (\text{Int}, \text{Int}) \rightarrow \text{Int}$

$\text{add } (x, y) = x + y$

$\text{perform_operation} :: ((\text{Int}, \text{Int}) \rightarrow \text{Int}) \rightarrow \text{Int}$

$\text{perform_operation } f = f(3, 4)$

$\text{perform_operation_with_3} :: ((\text{Int}, \text{Int}) \rightarrow \text{Int}) \rightarrow \text{Int} \rightarrow \text{Int}$

$\text{perform_operation } f \ x = f(3, x)$

$\text{add_with_3} = \text{perform_operation_with_3 } \text{add}$

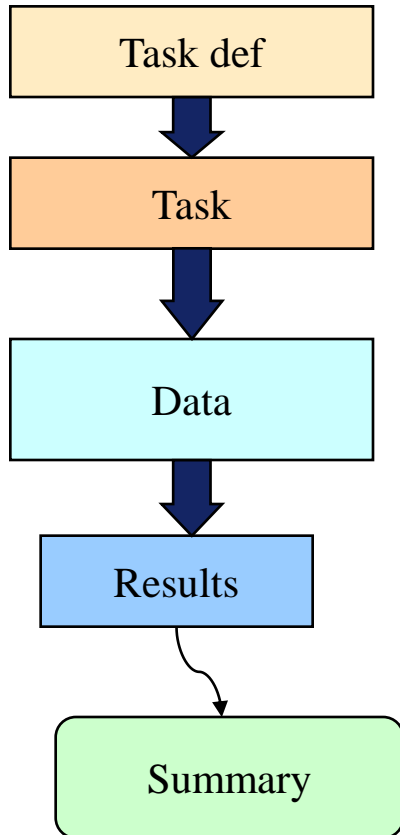
$\text{add_with_3 } 5$

> 8

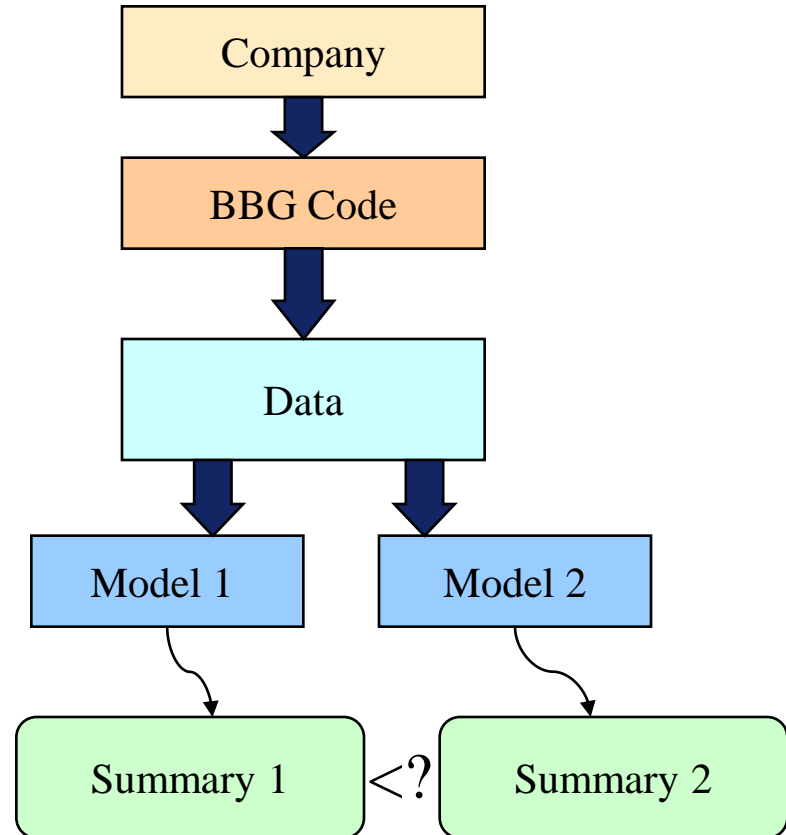
Client's logo
if needed

Example

Theory



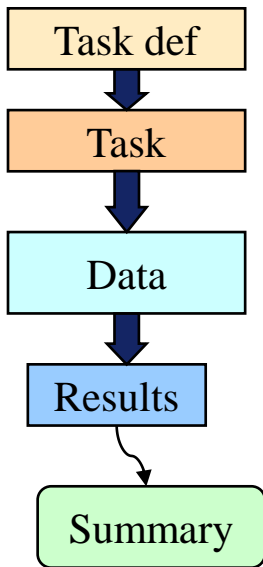
Practice



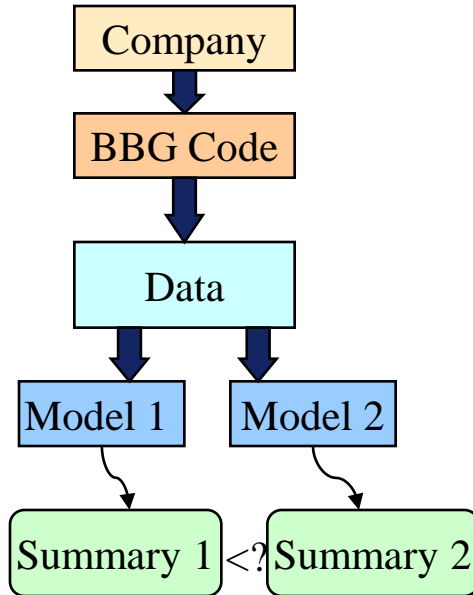
Client's logo
if needed

Example

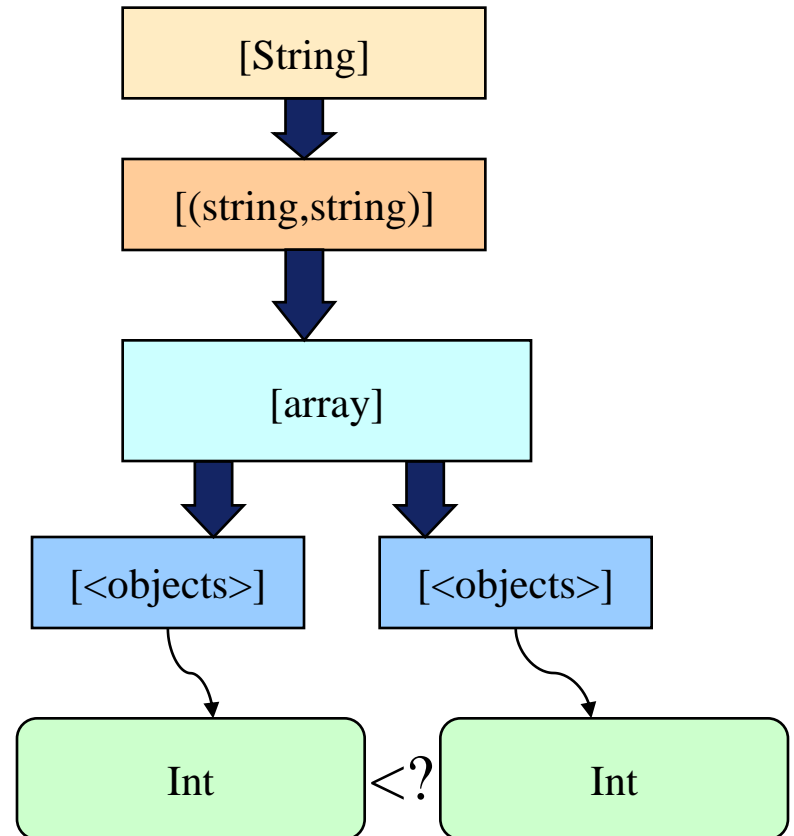
Theory



Practice

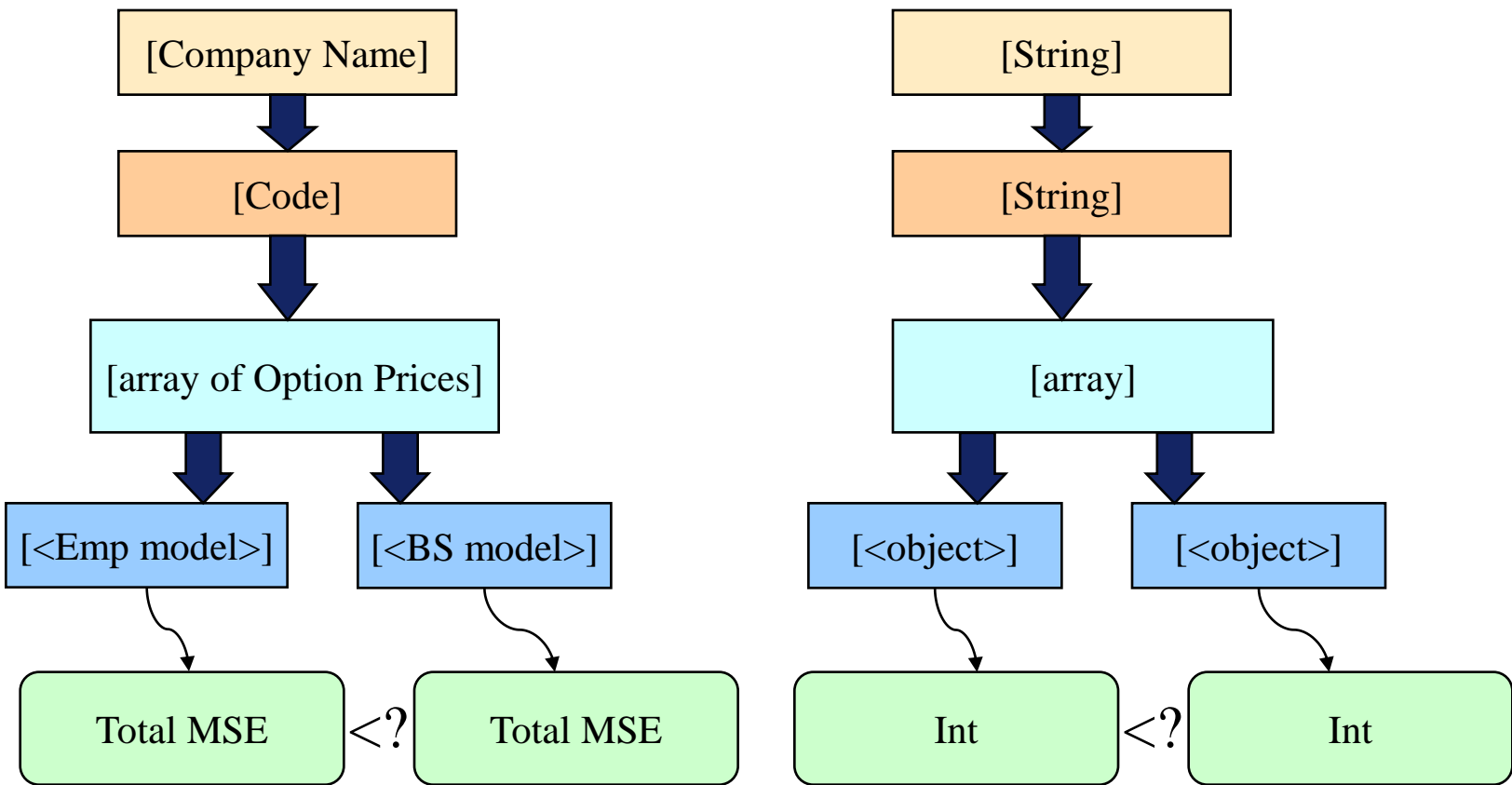


Practice



[1,2,3] is a list containing 1,2,3

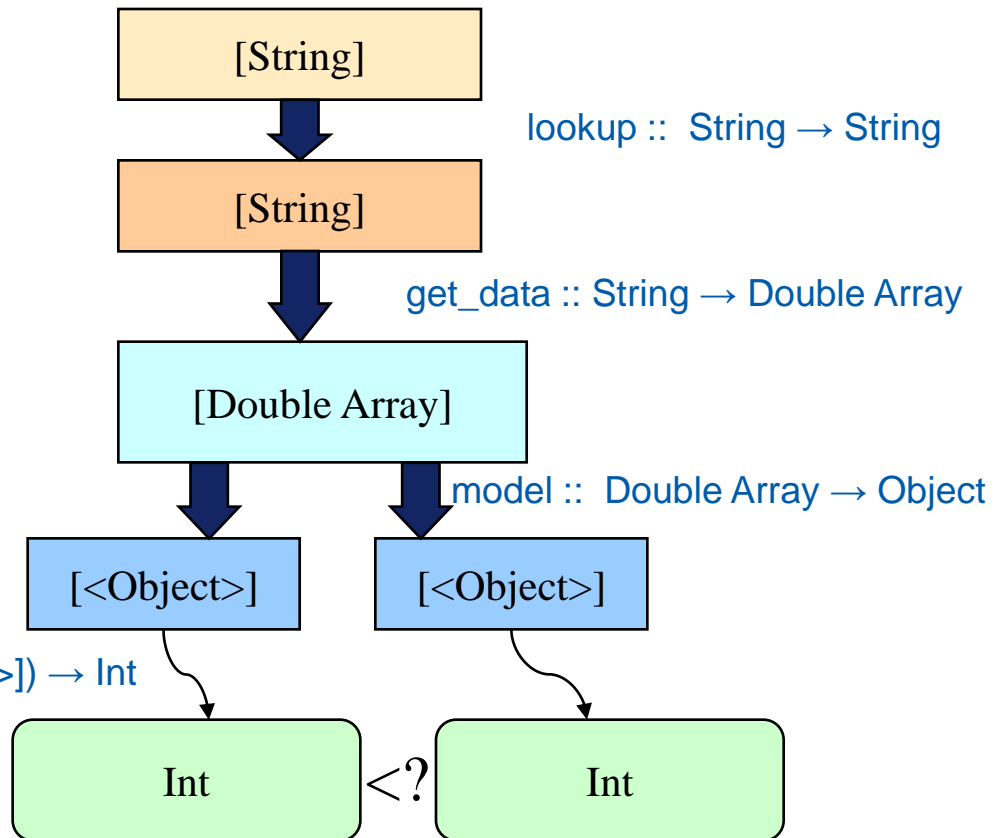
Client's logo
if needed



Client's logo
if needed

Map and fold

\Downarrow $\text{map} :: ((a \rightarrow b), [a]) \rightarrow [b]$
 \curvearrowright $\text{fold} :: (((a, b) \rightarrow a), a, [b]) \rightarrow a$



\curvearrowright $\text{fold} :: (((\text{Int}, \text{Im obj}) \rightarrow \text{Int}), \text{Int}, [\text{Im obj}]) \rightarrow \text{Int}$

Client's logo
if needed

Folds I

What does fold do?

Think of a conveyor:

conveyor :: [parts] → partly_built_car

worker :: (part,partly_built_car) → partly_built_car

fold :: (((α , β) → α), α , [β]) → α

fold :: ((part,partly_built_car) → partly_built_car, partly_built_car, [parts]) → partly_built_car

fold (funct, almost_car, []) = almost_car

fold (worker, almost_car, part:other_parts) = fold(worker, worker(part,almost_car), other_parts)

What's the difference between this and:

fold (worker, almost_car, part:other_parts) = worker(part,fold(worker, almost_car, other_parts))

Folds II

Summary in the previous slide can be implemented as follows:

```
obj_list = <list of lm objects>      # x:: [Object]
mse_list = map(extract_MSE, X) # extract_MSE :: Object → Double
summary_stat = sum(mse_list)
```

So, how do we write “sum” functional-style ???

```
sum :: [Double] → Double
```

```
sum [ ] = 0
```

```
sum (x:xs) = x + sum(xs)
```

OR notice that

```
+ :: (Double, Double) → Double
```

So

```
sum xs = fold(+,0.0,xs)
```

Client's logo
if needed

Functional R

map = lapply

fold = reduce

Also:

Mapply

Vectorize

Client's logo
if needed

In R

```
> A = list(3,4)
> add_3 = function(x,y=3) x+y
> add = function(x,y) x+y
> lapply(A,add_3)
[[1]]
[1] 6

[[2]]
[1] 7

> lapply(A,add)
Error in FUN(X[[1L]], ...) : element 2 is empty;
the part of the args list of '+' being evaluated was:
(x, y)
```

In Haskell

-- Compile time error which simply does not occur in dynamically typed languages

```
new_list = map (add_one) ["a", "b", "c"]
```

```
perm.hs:59:26:
```

```
Couldn't match expected type `Int' against inferred type `[Char]'
```

```
In the expression: "a"
```

```
In the second argument of `map', namely `["a", "b", "c"]'
```

```
In the expression: map (add_one) ["a", "b", "c"]
```

```
never_error = if (True) then [ ] else (map (add_one) ["a", "b", "c"])
```

```
perm.hs:55:54:
```

```
Couldn't match expected type `Int' against inferred type `[Char]'
```

```
In the expression: "a"
```

```
In the second argument of `map', namely `["a", "b", "c"]'
```

```
In the expression: (map (add_one) ["a", "b", "c"])
```

Client's logo
if needed

Partial Function Application in R

```
optim(par, fn, gr = NULL, ..., method = c("Nelder-Mead", "BFGS", "CG", "L-BFGS-B", "SANN"))
```

Arguments

... Further arguments to be passed to fn and gr.

```
f = function(x) log(x)+1
```

```
obj_fun = function(x,f) (x-f(x))^2
```

```
solution = optim(1000, obj_fun, NULL, f, method="BFGS")
```

Client's logo
if needed

Partial Function Application in Haskell

```
f :: Double → Double
```

```
f x = log(x) + 1
```

```
obj_fun :: (Double → Double) → Double → Double
```

```
obj_fun f x = (x - (f x))^2
```

```
a = optim(1000, (obj_fun f), method = "BFGS")
```


Some more partial function application

Consider the function `mapply` (in fact, it's more general than that):

`mapply :: ((α , β) \rightarrow γ , [α], [β]) \rightarrow [γ]`

```
list_of_stuff = list("a","b","c")           ##### ["a","b","c"]
some_labels = as.list(seq(10,14,by=2)) ##### [10,12,14]
mapply(pair_up, list_of_stuff,some_labels, SIMPLIFY=FALSE) ##### [ ["a",10],["b",12],["c",14] ]
```

There is also a function called `Vectorize`:

`Vectorize :: ((α , β) \rightarrow γ) \rightarrow (([α], [β]) \rightarrow [γ])`

```
new_function = Vectorize(pair_up, SIMPLIFY=FALSE)
```

```
new_function(list_of_stuff, some_labels)
```

in Haskell this happens automatically, `mapply :: ((α , β) \rightarrow γ) \rightarrow [α] \rightarrow [β] \rightarrow [γ]`

Client's logo
if needed

```
ins_every_position :: a -> [a] -> [[a]]
```

```
ins_every_position elt lst = insert elt [] lst
```

```
insert :: a -> [a] -> [a] -> [[a]]
```

```
insert elt front [ ] = [front ++ [elt]]
```

```
insert elt front (x:xs) = (front ++(elt:x:xs)):(insert elt xs (front++[x]))
```