Replacing Functions with Data

Bartosz Milewski, 2020

Functional Programming

- First class functions
  - function objects `a->d`
  - a.k.a., exponentials d^a
- As opposed to functions (a.k.a., morphisms)
- What's a function object?
- You can use apply on it

```
apply :: ( a->d , a) -> d
```

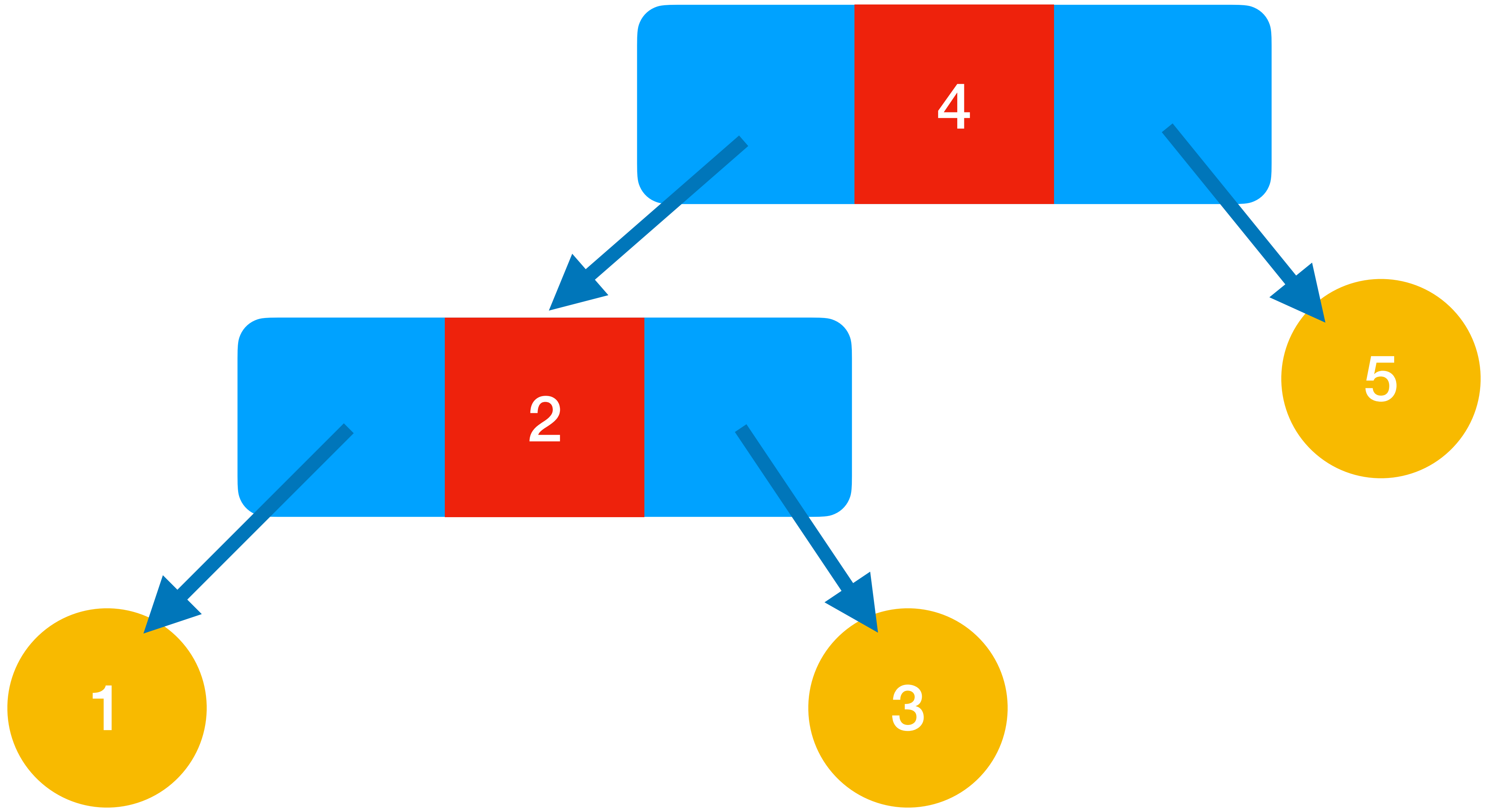
```
data Tree = Leaf String
          | Node Tree String Tree

tree :: Tree
tree = Node (Node (Leaf "1 ") "2 " (Leaf "3 "))
            "4 "
            (Leaf "5 ")

show1 :: Tree -> String
show1 (Leaf s) = s
show1 (Node l s r) =
  show1 l ++  s ++ show1 r

test = show1 tree
```

“1 2 3 4 5”



```haskell
show2 :: Tree -> (String -> a) -> a
show2 (Leaf s) k = k s
show2 (Node lft s rgt) k =
  show2 lft (\ls ->
    show2 rgt (\rs ->
      k (ls ++ s ++ rs)))
```

```haskell
\ls ->
    show2 rgt (\rs ->
      k (ls ++ s ++ rs))
```

```haskell
\rs ->
      k (ls ++ s ++ rs)
```

```haskell
show t = show2 t (\x -> x)
```

```haskell
\x -> x
```

$$\boxed{\texttt{\\x -> x}}$$

```
\ls ->
    show2 rgt (\rs ->
      k (ls ++ s ++ rs))
```

```
\rs ->
      k (ls ++ s ++ rs)
```

```
done s = s
next (s, rgt, k) ls = show3 rgt (conc (ls, s, k))
conc (ls, s, k) rs = k (ls ++ s ++ rs)
```

```
show3 :: Tree -> (String -> a) -> a
show3 (Leaf s) k = k s
show3 (Node lft s rgt) k =
  show3 lft (next (s, rgt, k))

show t = show3 t done
```

- Closures

```
\x -> x
```

```
\ls ->
    show2 rgt (\rs ->
      k (ls ++ s ++ rs))
```

```
\rs ->
      k (ls ++ s ++ rs)
```

- Named functions

```
done                   s = s
next (s, rgt, k) ls = show3 rgt (conc (ls, s, k))
conc (ls, s, k)  rs = k (ls ++ s ++ rs)
```

- Captured types

```
()
(String, Tree,   String -> String)
(String, String, String -> String)
```

Adjunction

- Adjunction between two functors:
  - (L c -> d) in one to one correspondence with (c -> R d)
  - L_a c = (c, a)
  - R_a d = a->d
- Currying

$L_a c$ $R_a d$

```
curry   :: ((c, a) -> d)    -> (c -> (a -> d))
uncurry :: (c -> (a -> d)) -> ((c, a) -> d)
```

Counit

- Counit of adjunction $L \circ R \to Id$
- $L_a(R_a d) \to Id\, d$
  - $(a \to d, a) \to d$
  - Counit is **`apply`**

- L_a c = (c, a)
- R_a d = a->d

Constructing the Adjoint

- If we were to construct a function object **a->d**
  - Take all closures from **a** to **d** for all possible environments **c**
  - Comma category with objects **(c, (c, a) -> d)**

```
( (String, Tree, String -> String)
,((String, Tree, String -> String), String) -> String
```

```
next (s, rgt, k) ls =
    show3 rgt (conc (ls, s, k))
```

Constructing the Adjoint

- Imagine enumerating all possible environments c_i
- Comma category $(c_i, (c_i, a) \to d)$
- The sum $\sum c_i$ is a good candidate for the function object
- apply: $(\sum c_i, a) \to d \cong \sum (c_i, a) \to d \cong \prod \big((c_i, a) \to d\big)$
  - Distributivity: $(\sum c_i, a) \cong \sum (c_i, a)$
- A lot of overcounting. *Colimit* instead of sum fixes that.

$$\boxed{\begin{array}{c}\textbf{apply}\\ (a \to d, a) \to d\end{array}}$$

Solution Set

- A set of environments (a solution sets, satisfying certain properties) is enough to define an adjoint functor.

- If we don’t insist on completeness and uniqueness, the sum (coproduct) of a limited set of environments can serve as a function object

c_i

```
()
(String, Tree,    String -> String)
(String, String, String -> String)
```

$\sum c_i$

```
data Kont = Done
          | Next String Tree   Kont
          | Conc String String Kont
```

$(\sum c_i, a) \to d$

```
apply :: Kont -> String -> String
apply Done s = s
apply (Next s rgt k) ls = show4 rgt (Conc ls s k)
apply (Conc ls s k) rs  = apply k (ls ++ s ++ rs)
```

```haskell
show3 :: Tree -> (String -> a) -> a
show3 (Leaf s) k = k s
show3 (Node lft s rgt) k =
  show3 lft (next (s, rgt, k))

show t = show3 t done
```

```haskell
show4 :: Tree -> Kont -> String
show4 (Leaf s) k = apply k s
show4 (Node lft s rgt) k =
  show4 lft (Next s rgt k)


show t = show4 t Done
```

```
data Kont = Done
          | Next String Tree   Kont
          | Conc String String Kont
```

- This recursive data structure can be represented as a list
- Empty list is constructed using **`Done`**
- Two constructors **`Next`** and **`Conc`** can be combined to form the *cons* of the list of **`(String, Either Tree String)`**
- Traversal with a user-defined stack

```
type Kont = [(String, Either Tree String)]
```

Applications

- In imperative programming, defunctionalization removes recursion (prevents stack overflow)
- Distributed programming, web services

BIBLIOGRAPHY

- John C. Reynolds, Definitional Interpreters for Higher-Order Programming Languages

- James Koppel, The Best Refactoring You’ve Never Heard Of