Higher-Order Programming:

Closures, functional abstraction, genericity, instantiation, embedding. Control abstractions: iterate, map, reduce, fold, filter (CTM Sections 1.9, 3.6, 4.7)

Carlos Varela

RPI

September 20, 2024

Adapted with permission from:

Seif Haridi

KTH

Peter Van Roy

UCL

Higher-order programming

- Assume we want to write another Pascal function which instead of adding numbers, performs exclusive-or on them
- It calculates for each number whether it is odd or even (parity)
- Either write a new function each time we need a new operation, or write one generic function that takes an operation (another function) as argument
- The ability to pass functions as arguments, or return a function as a result is called *higher-order programming*
- Higher-order programming is an aid to build generic abstractions

Variations of Pascal

- Compute the parity Pascal triangle

```
xor x y = if x == y then 0 else 1
```

```
          1
        1   1
      1   2   1
    1   3   3   1
  1   4   6   4   1
```

```
          1
        1   1
      1   0   1
    1   1   1   1
  1   0   0   0   1
```

Higher-order programming

```
genericPascal op 1 = [1]
genericPascal op n = opList op  (shiftLeft l)
                                (shiftRight l)
  where
    l = genericPascal op (n-1)
    shiftLeft [] = [0]
    shiftLeft (h:t) = h:shiftLeft t
    shiftRight l = 0:l
    opList op [] [] = []
    opList op (h1:t1) (h2:t2) = (op h1 h2):opList t1 t2
```

```
add x y = x + y
xor x y = if x == y then 0 else 1


pascal n = genericPascal add n
parityPascal n = genericPascal xor n
```

add and xor functions are passed as arguments.

The iterate' control abstraction

```
iterate' s isDone transform =
  if isDone s then s
  else let s1 = transform s in
          iterate' s1 isDone transform
```

iterate' s_i =
  if *isDone* s_i then s_i
  else let s_{i+1} = *transform* s_i in
          iterate' s_{i+1}

Sqrt using iterate'

```
iterate' s isDone transform =
  if isDone s then s
  else let s1 = transform s in
          iterate' s1 isDone transform

sqrt' x = iterate' 1.0 goodEnough improve
  where goodEnough = \g -> (abs (x - g*g))/x < 0.00001
           improve = \g -> (g + x/g)/2.0
```

Sqrt using iterate abstraction

```
sqrt8 x = head (dropWhile (not . goodEnough) sqrtGuesses)
        where
            goodEnough guess = (abs (x – guess*guess))/x < 0.00001
            improve guess = (guess + x/guess)/2.0
            sqrtGuesses = iterate improve 1
```

This sqrt example uses infinite lists enabled by lazy evaluation, and the iterate control abstraction.

Sqrt using infinite lists and map

```
sqrt7 x = head (dropWhile (not . goodEnough) sqrtGuesses)
        where
            goodEnough guess = (abs (x – guess*guess))/x < 0.00001
            improve guess = (guess + x/guess)/2.0
            sqrtGuesses = 1:(map improve sqrtGuesses)
```

This sqrt example uses infinite lists enabled by lazy evaluation, and the map control abstraction.

Map control abstraction

```
map' :: (a -> b) -> [a] -> [b]
map' _ []    = []
map' f (h:t) = f h:map' f t
```

_ means that the argument is not used (read "don't care").
map' is to distinguish it from the Prelude's map function.

Higher-order programming

- Higher-order programming = the set of programming techniques that are possible with function values (lexically-scoped closures)
- Basic operations
  - Functional abstraction: creating function values with lexical scoping
  - Genericity: function values as arguments
  - Instantiation: function values as return values
  - Embedding: function values in data structures
- Higher-order programming is the foundation of component-based programming and object-oriented programming

Functional abstraction

- Functional abstraction is the ability to convert any statement into a function value
  - A function value is usually called a closure, or more precisely, a lexically-scoped closure
  - A function value is a pair: it combines the function code with the environment where the function was created (the contextual environment)
- Basic scheme:
  - Consider any expression <e>
  - Convert it into a function value: f = \z -> <e> (z fresh)
  - Executing (f undefined) has exactly the same effect as executing <e>

Functional abstraction in Oz/Haskell

```
fun {AndThen B1 B2}
   if B1 then B2 else false
   end
end
```

```
andthen b1 b2 =
   if b1 then b2 else False
```

Sample usage:

```
(x /= 0) `andthen` (y/x > 1)
```

Functional abstraction

```
fun {AndThen' B1 B2}
   if {B1} then {B2} else false
   end
end
```

```
andthen' b1 b2 =
   if (b1 undefined)
   then (b2 undefined)
   else False
```

Sample usage:

```
(\z -> (x /= 0)) `andthen'`
(\z -> (y/x > 1))
```

Functional abstraction

- Any expression can be abstracted to a function by selecting a number of the 'free' variable identifiers and enclosing the expression into a function with the identifiers as parameters, e.g.:

```
if x >= y then x else y
```

- Abstracting over all variables:

```
max x y = if x >= y then x else y
```

- Abstracting over x:

```
lowerBound x = if x >= y then x else y
```

Lexical scope

```
b = 2
exp' n =  if (n == 0) then 1 else b * exp' (n-1)

let b = 3 in exp' 4
```

We want exp' to behave as the function $f(n) = 2^n$, independently of any value of b in the function calling context.

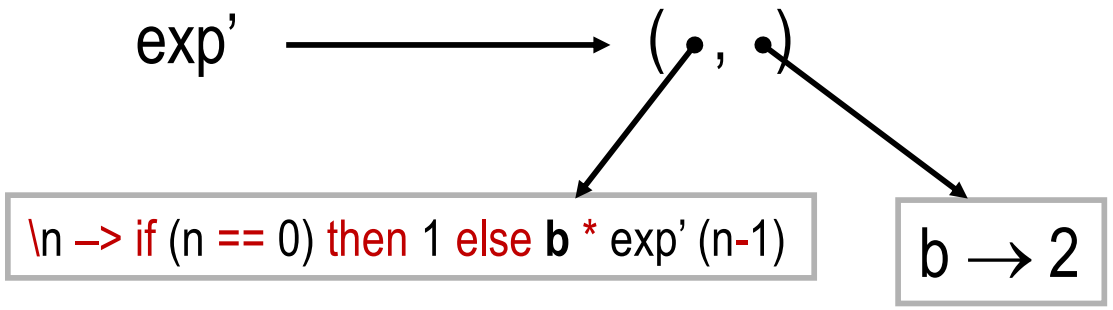
Function values

- Constructing a function value in the memory system is not simple because a function may have external references

```
b = 2
exp' n =  if (n == 0) then 1 else b * exp' (n-1)

let b = 3 in exp' 4
```

Function values (2)




```
b = 2
exp' n =  if (n == 0) then 1 else b * exp' (n-1)

let b = 3 in exp' 4
```

Function values (3)

- The semantic expression is

  $(\langle x \rangle\ \langle y_1 \rangle \ldots \langle y_n \rangle = \langle e \rangle, E)$

- $\langle y_1 \rangle \ldots \langle y_n \rangle$ are the (formal) parameters of the function
- Other free identifiers of $\langle e \rangle$ are called external references $\langle z_1 \rangle \ldots \langle z_k \rangle$
- These are defined by the environment E where the function is declared (lexical scoping)
- The contextual environment of the function CE is $E|_{\{\langle z1 \rangle \ldots \langle zk \rangle\}}$
- When the function is called CE is used to construct the environment of $\langle e \rangle$

$(\backslash\ \langle y_1 \rangle \ldots \langle y_n \rangle \rightarrow$
$\langle e \rangle,$
$CE)$

Function values (4)

- Function values are pairs:

  (\ ⟨y_1⟩ ... ⟨y_n⟩ -> ⟨e⟩, CE)

- They are stored in the memory system just as any other value

(\ ⟨y_1⟩ ... ⟨y_n⟩ ->
    ⟨e⟩,
CE)

Function application

- The semantic expression is
  $(\langle x \rangle \langle y_1 \rangle \ldots \langle y_n \rangle, E)$
- Execution proceeds as follows:
  - If $E(\langle x \rangle)$ is not a function value, or a function with arity that is less than n, raise an error
  - $E(\langle x \rangle)$ is (\ $\langle z_1 \rangle \ldots \langle z_n \rangle$ -> $\langle e \rangle$, CE) ,
    push
    ($\langle e \rangle$, CE + $\{\langle z_1 \rangle \rightarrow E(\langle y_1 \rangle) \ldots \langle z_n \rangle \rightarrow E(\langle y_n \rangle)\}$)
    on the stack

A common limitation

- Most popular imperative languages (C, Pascal) do not have function values
- They have only half of the pair: variables can reference function code, but there is no contextual environment
- This means that control abstractions cannot be programmed in these languages
  - They provide a predefined set of control abstractions (for, while loops, if statement)
- Generic operations are still possible
  - They can often get by with just the function code. The contextual environment is often empty.
- The limitation is due to the way memory is managed in these languages
  - Part of the store is put on the stack and deallocated when the stack is deallocated
  - This is supposed to make memory management simpler for the programmer on systems that have no garbage collection
  - It means that contextual environments cannot be created, since they would be full of dangling pointers
- Object-oriented programming languages can use objects to encode function values by making external references (contextual environment) instance variables.

Genericity

- Replace specific entities (zero 0 and addition +) by function arguments
- The same routine can do the sum, the product, the logical or, etc.

```
sumlist :: (Num a) => [a] -> a
sumlist []    = 0
sumlist (h:t) = h+sumlist t
```

```
foldr' :: (a->b->b) -> b -> [a] -> b
foldr' _ u []    = u
foldr' f u (h:t) = f h (foldr' f u t)
```

Instantiation

- Instantiation is when a function returns a function value as its result
- Calling foldFactory (\x y -> x+y) 0 returns a function that behaves identically to sumlist, which is an « instance » of a folding function
- Notice that in Haskell, foldFactory and foldr' are the same (higher-order) function (after three eta-reductions).

```
foldFactory f u = \l -> foldr' f u l
```

Currying

- Currying is a technique that can simplify programs that heavily use higher-order programming.
- The idea: function of n arguments ⇒ n nested functions of one argument.
- Advantage: The intermediate functions can be useful in themselves.
- In Haskell, all functions are curried:

```
max x y = if x >= y then x else y
```

```
max x = \y -> if x >= y then x else y
```

Embedding

- Embedding is when function values are put in data structures
- Embedding has many uses:
  - Modules: a module is a record/tuple that groups together a set of related operations
  - Software components: a software component is a generic function that takes a set of modules as its arguments and returns a new module. It can be seen as specifying a module in terms of the modules it needs.
  - Delayed evaluation (also called explicit lazy evaluation): build just a small part of a data structure, with functions at the extremities that can be called to build more. The consumer can control explicitly how much of the data structure is built.

Control Abstractions

```
foldl' :: (a->b->b) -> b -> [a] -> b
foldl' _ u []    = u
foldl' f u (h:t) = foldl' f (f h u) t
```

Assume a list [*x1,x2,x3,....*]

S0 $\rightarrow$ *S1* $\rightarrow$ *S2*

u $\rightarrow$ f *x1* u $\rightarrow$ f *x2* (f *x1* u) $\rightarrow$ ....$\rightarrow$

Control Abstractions

```
foldl' :: (a->b->b) -> b -> [a] -> b
foldl' _ u []    = u
foldl' f u (h:t) = foldl' f (f h u) t
```

What does this program do ?

```
foldl' (\x y -> x:y) [] [1,2,3]
```

Fold left

```
foldl' :: (a->b->b) -> b -> [a] -> b
foldl' _ u []    = u
foldl' f u (h:t) = foldl' f (f h u) t
```

Notice the unit u is of type b, list elements are of type a, and the function f is of type a->b->b.

Two more folding functions

Given a list $[e_1\ e_2\ ...\ e_n]$ and a binary function $\odot$, with unit U, the previous folding functions do the following:

$(e_1 \odot ... (e_{n-1} \odot (e_n \odot U))...)$	fold right
$(e_n \odot ... (e_2 \odot (e_1 \odot U))...)$	fold left

But there are two other possibilities:

$(...((U \odot e_n) \odot e_{n-1})... \odot e_1)$	fold right unit left
$(...((U \odot e_1) \odot e_2)... \odot e_n)$	fold left unit left

FoldL unit left

```
foldlul :: (b->a->b) -> b -> [a] -> b
foldlul _ u []    = u
foldlul f u (h:t) = foldlul f (f u h) t
```

Notice the unit u is of type b, list elements are of type a, and the function f is of type b->a->b.

List-based techniques

```
map' :: (a -> b) -> [a] -> [b]
map' _ []    = []
map' f (h:t) = f h:map' f t
```

```
filter' :: (a-> Bool) -> [a] -> [a]
filter' _ []    = []
filter' p (h:t) = if p h then h:filter' p t
                         else filter' p t
```

Filter as FoldR application

```
filter'' :: (a-> Bool) -> [a] -> [a]
filter'' p l = foldr
        (\h t ->  if p h
                  then h:t
                  else t) [] l
```

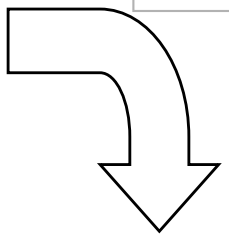
```
foldr' :: (a->b->b) -> b -> [a] -> b
foldr' _ u []    = u
foldr' f u (h:t) = f h (foldr' f u t)
```

Tree-based techniques

```
data Tree a = Leaf a | Branch (Tree a) (Tree a)

multiply (Leaf x)       = x
multiply (Branch l r) = multiply l * multiply r
```

Call unary lf at each leaf node, and binary bf at each branch node



```
visit bf lf (Leaf x)       = lf x
visit bf lf (Branch l r) = bf (visit bf lf l)
                              (visit bf lf r)
```

Explicit lazy evaluation

- Supply-driven evaluation. (e.g.The list is completely calculated independent of whether the elements are needed or not.)
- Demand-driven execution.(e.g. The consumer of the list structure asks for new list elements when they are needed.)
- Technique: a programmed trigger.
- How to do it with higher-order programming? The consumer has a function that it calls when it needs a new list element. The function call returns a pair: the list element and a new function. The new function is the new trigger: calling it returns the next data item and another new function. And so forth.

Explicit lazy functions in Oz

```
from n = n : from (n+1)
```

```
fun lazy {From N}
   N | {From N+1}
end
```



```
fun {From N}
   fun {$} N | {From N+1} end
end
```

Exercises

23. Define an incList function to take a list of numbers and increment all its values, using the map control abstraction. For example:

```
incList [3,1,7]    => [4,2,8]
```

24. Create a higher-order mapReduce function that takes as input two functions corresponding to map and reduce respectively, and returns a function to perform the composition. Illustrate your mapReduce function with an example.
25. Using the tree visit control abstraction, create a tree2List function to serialize all the binary tree elements.