

# Higher-Order Programming:

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

Carlos Varela

RPI

September 20, 2019

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

```
fun {Xor X Y} if X==Y then 0 else 1 end end
```

	1						1				
	1	1					1	1			
	1	2	1				1	0	1		
	1	3	3	1			1	1	1	1	
	1	4	6	4	1		1	0	0	0	1

# Higher-order programming

```
fun {GenericPascal Op N}
  if N==1 then [1]
  else L in L = {GenericPascal Op N-1}
    {OpList Op {ShiftLeft L} {ShiftRight L}}
  end
end
fun {OpList Op L1 L2}
  case L1 of H1|T1 then
    case L2 of H2|T2 then
      {Op H1 H2}{OpList Op T1 T2}
    end
  end
  end
  else nil end
end
```

```
fun {Add N1 N2} N1+N2 end
fun {Xor N1 N2}
  if N1==N2 then 0 else 1 end
end
fun {Pascal N} {GenericPascal Add N} end
fun {ParityPascal N}
  {GenericPascal Xor N}
end
```

Add and Xor functions  
are passed as  
arguments.

# The Iterate control abstraction

```
fun {Iterate S IsDone Transform}
  if {IsDone S} then S
  else S1 in
    S1 = {Transform S}
    {Iterate S1 IsDone Transform}
  end
end
```

```
fun {Iterate  $S_i$ }
  if {IsDone  $S_i$ } then  $S_i$ 
  else  $S_{i+1}$  in
     $S_{i+1} = \{Transform S_i\}$ 
    {Iterate  $S_{i+1}$ }
  end
end
```

# Sqrt using the control abstraction

```
fun {Sqrt X}  
  {Iterate  
    1.0  
    fun {$ G} {Abs X - G*G}/X < 0.000001 end  
    fun {$ G} (G + X/G)/2.0 end  
  }  
end
```

IsDone and Transform anonymous functions are passed as arguments.

# Sqrt in Haskell

```
let sqrt 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.

# Functions are procedures in Oz

```
fun {Map Xs F}  
  case Xs  
  of nil then nil  
  [] X|Xr then {F X}|{Map Xr F}  
  end  
end
```



```
proc {Map Xs F Ys}  
  case Xs  
  of nil then Ys = nil  
  [] X|Xr then Y Yr in  
    Ys = Y|Yr  
    {F X Y}  
    {Map Xr F Yr}  
  end  
end
```



# Map in Haskell

`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 procedure values (lexically-scoped closures)
- Basic operations
  - **Procedural abstraction**: creating procedure values with lexical scoping
  - **Genericity**: procedure values as arguments
  - **Instantiation**: procedure values as return values
  - **Embedding**: procedure values in data structures
- Higher-order programming is the foundation of component-based programming and object-oriented programming

# Procedural abstraction

- Procedural abstraction is the ability to convert any statement into a procedure value
  - A procedure value is usually called a **closure**, or more precisely, a **lexically-scoped closure**
  - A procedure value is a pair: it combines the procedure code with the environment where the procedure was created (the contextual environment)
- Basic scheme:
  - Consider any statement  $\langle s \rangle$
  - Convert it into a procedure value:  $P = \text{proc } \{\$ \} \langle s \rangle \text{ end}$
  - Executing  $\{P\}$  has **exactly the same effect** as executing  $\langle s \rangle$

# Procedural abstraction

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

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

# Procedure abstraction

- Any statement can be abstracted to a procedure by selecting a number of the 'free' variable identifiers and enclosing the statement into a procedure with the identifiers as parameters

- `if X >= Y then Z = X else Z = Y end`

- Abstracting over all variables

```
proc {Max X Y Z}
    if X >= Y then Z = X else Z = Y end
end
```

- Abstracting over X and Z

```
proc {LowerBound X Z}
    if X >= Y then Z = X else Z = Y end
end
```

# Lexical scope

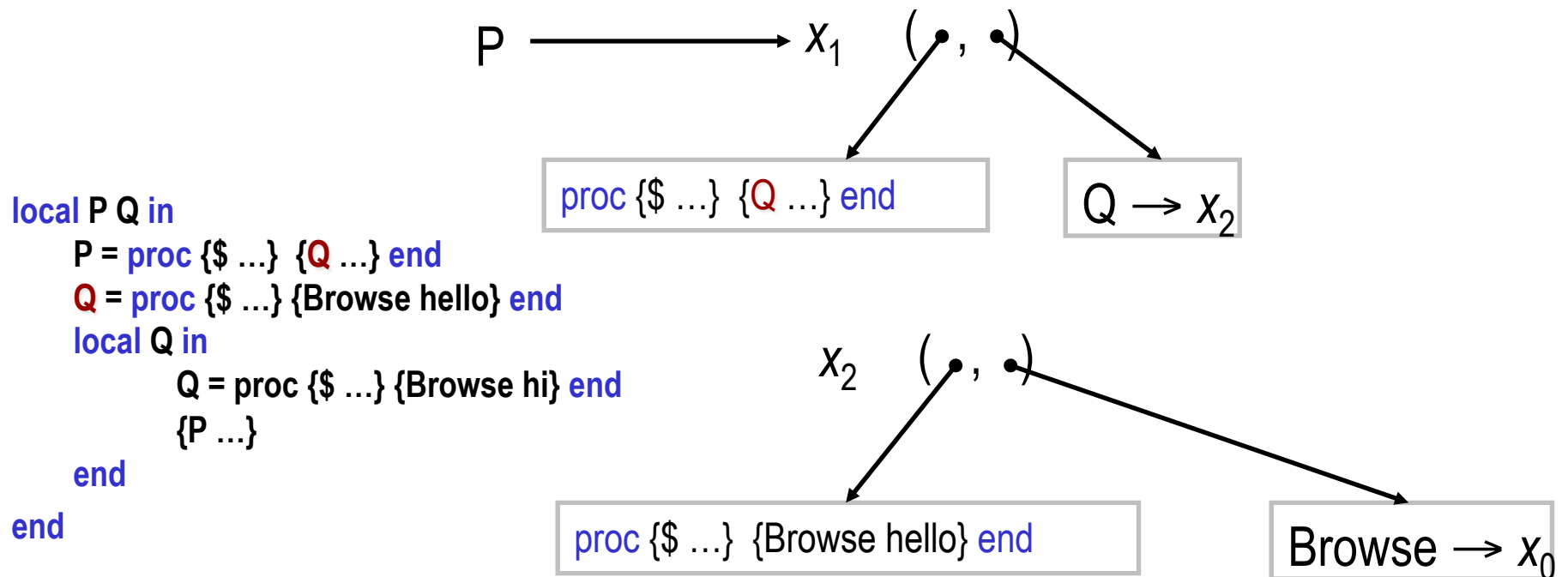
```
local P Q in
  proc {P ...} {Q ...} end
  proc {Q ...} {Browse hello} end
  local Q in
    proc {Q ...} {Browse hi} end
    {P ...}
  end
end
```

# Procedure values

- Constructing a procedure value in the store is not simple because a procedure may have external references

```
local P Q in
  P = proc {$ ...} {Q ...} end
  Q = proc {$ ...} {Browse hello} end
  local Q in
    Q = proc {$ ...} {Browse hi} end
    {P ...}
  end
end
```

# Procedure values (2)





# Procedure values (3)

- The semantic statement is  $(\text{proc } \{\langle x \rangle \langle y_1 \rangle \dots \langle y_n \rangle\} \langle s \rangle \text{ end}, E)$
- $\langle y_1 \rangle \dots \langle y_n \rangle$  are the (formal) parameters of the procedure
- Other free identifiers of  $\langle s \rangle$  are called external references  $\langle z_1 \rangle \dots \langle z_k \rangle$
- These are defined by the environment  $E$  where the procedure is declared (lexical scoping)
- The contextual environment of the procedure  $CE$  is  $E \mid_{\{\langle z_1 \rangle \dots \langle z_k \rangle\}}$
- When the procedure is called  $CE$  is used to construct the environment of  $\langle s \rangle$

```
(proc { $\$ \langle y_1 \rangle \dots \langle y_n \rangle$ }  
   $\langle s \rangle$   
end ,  
CE)
```

# Procedure values (4)

- Procedure values are pairs:  
(`proc` {\$  $\langle y_1 \rangle$  ...  $\langle y_n \rangle$   $\langle s \rangle$  `end` , *CE*)
- They are stored in the store just as any other value

```
(proc {$  $\langle y_1 \rangle$  ...  $\langle y_n \rangle$  }  
   $\langle s \rangle$   
end ,  
CE)
```

# A common limitation

- Most popular imperative languages (C, Pascal) do **not** have procedure values
- They have only **half** of the pair: variables can reference procedure 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 procedure 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 procedure 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.

```
fun {SumList L}
  case L
  of nil then 0
  [] X|L2 then X+{SumList L2}
  end
end
```



```
fun {FoldR L F U}
  case L
  of nil then U
  [] X|L2 then {F X {FoldR L2 F U}}
  end
end
```

# Genericity in Haskell

- 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

```
fun {FoldFactory F U}
  fun {FoldR L}
    case L
    of nil then U
    [] X|L2 then {F X {FoldR L2}}
    end
  end
in
  FoldR
end
```

- Instantiation is when a procedure returns a procedure value as its result
- Calling {FoldFactory fun {\$ A B} A+B end 0} returns a function that behaves identically to SumList, which is an « **instance** » of a folding function

# Currying

- Currying is a technique that can simplify programs that heavily use higher-order programming.
- The idea: function of n arguments  $\Rightarrow$  n nested functions of one argument.
- Advantage: The intermediate functions can be useful in themselves.

```
fun {Max X Y}
  if X>=Y then X else Y end
end
```



```
fun {Max X}
  fun {$ Y}
    if X>=Y then X else Y end
  end
end
```

# Embedding

- Embedding is when procedure values are put in data structures
- Embedding has many uses:
  - **Modules**: a module is a record 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

declare

proc {For I J P}

  if I  $\geq$  J then skip

  else {P I} {For I+1 J P}

  end

end

{For 1 10 Browse}

for I in 1..10 do {Browse I} end

# Control Abstractions

```
proc {ForAll Xs P}  
  case Xs  
  of nil then skip  
  [] X|Xr then  
    {P X} {ForAll Xr P}  
  end  
end
```

```
{ForAll [a b c d]  
  proc {$ I} {System.showInfo "the item is: " # I} end}
```

```
for I in [a b c d] do  
  {System.showInfo "the item is: " # I}  
end
```

# Control Abstractions

```
fun {FoldL Xs F U}  
  case Xs  
  of nil then U  
  [] X|Xr then {FoldL Xr F {F X U}}  
  end  
end
```

Assume a list  $[x1\ x2\ x3\ \dots]$

$$S0 \rightarrow S1 \rightarrow S2$$
$$U \rightarrow \{F\ x1\ U\} \rightarrow \{F\ x2\ \{F\ x1\ U\}\} \rightarrow \dots \rightarrow$$

# Control Abstractions

```
fun {FoldL Xs F U}  
  case Xs  
  of nil then U  
  [] X|Xr then {FoldL Xr F {F X U}}  
  end  
end
```

What does this program do ?

```
{Browse {FoldL [1 2 3]  
  fun {$ X Y} X|Y end nil}}
```

# FoldL in Haskell

`foldl' :: (b->a->b) -> b -> [a] -> b`

`foldl' _ u [] = u`

`foldl' f u (h:t) = foldl' f (f u h) t`

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

# List-based techniques

```
fun {Map Xs F}
  case Xs
  of nil then nil
  [] X|Xr then
    {F X}|{Map Xr F}
  end
end
```

```
fun {Filter Xs P}
  case Xs
  of nil then nil
  [] X|Xr andthen {P X} then
    X|{Filter Xr P}
  [] X|Xr then {Filter Xr P}
  end
end
```

# Filter in Haskell

`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

```
fun {Filter P L}
  {FoldR fun {$ H T}
    if {P H} then
      H|T
    else T end
  end nil L}
end
```

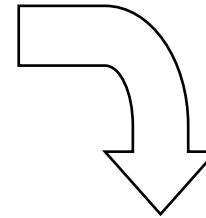
```
filter" :: (a-> Bool) -> [a] -> [a]
filter" p l = foldr
  (\h t -> if p h
    then h:t
    else t) [] l
```



# Tree-based techniques

```
proc {DFS Tree}
  case Tree of tree(node:N sons:Sons ...) then
    {Browse N}
    for T in Sons do {DFS T} end
  end
end
```

Call {P T} at each node T



```
proc {VisitNodes Tree P}
  case Tree of tree(node:N sons:Sons ...) then
    {P N}
    for T in Sons do {VisitNodes T P} end
  end
end
```

# 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

```
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:

$$\{\text{IncList [3 1 7]}\} \Rightarrow [4 2 8]$$

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. Write solutions for exercises 23 and 24 in both `Oz` and `Haskell`. Compare your solutions.