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 19, 2023

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

\[
\text{fun \{Xor X Y\} if } X==Y \text{ then 0 else 1 end end}
\]

\[
\begin{array}{cccccc}
1 & & & & & 1 \\
1 & 1 & & & & 1 \\
1 & 2 & 1 & & & 1 \\
1 & 3 & 3 & 1 & & 1 \\
1 & 4 & 6 & 4 & 1 & 1 \\
\end{array}
\]
## 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
- 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 using Iterate in Haskell

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 in Haskell

\[
\text{sqrt } x = \text{head (dropWhile (not . goodEnough) sqrtGuesses)}
\]

where

\[
\text{goodEnough guess} = \frac{|x - \text{guess} \times \text{guess}|}{x} < 0.00001
\]

\[
\text{improve guess} = \frac{\text{guess} + \frac{x}{\text{guess}}}{2.0}
\]

\[
\text{sqrtGuesses} = 1:(\text{map improve sqrtGuesses})
\]

This \text{sqrt} example uses infinite lists enabled by lazy evaluation, and the \text{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

\[
\text{map'} :: (a \to b) \to [a] \to [b]
\]

\[
\text{map'} _ [] = []
\]

\[
\text{map'} f (h:t) = f h : \text{map'} f t
\]

_ means that the argument is not used (read “don’t care”).
map’ is to distinguish it from the Prelude’s \text{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 <s>
  - Convert it into a procedure value: \( P = \text{proc} \{\$\} <s> \text{ end} \)
  - Executing \{P\} has exactly the same effect as executing <s>
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 $X \geq Y$ then $Z = X$ else $Z = Y$ end
- Abstracting over all variables
  ```
  proc {Max X Y Z}
      if $X \geq Y$ then $Z = X$ else $Z = Y$ end
  end
  ```
- Abstracting over $X$ and $Z$
  ```
  proc {LowerBound X Z}
      if $X \geq Y$ 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

```python
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)

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

```
proc {$ …} {Q …} end

Browse → x_0
```

```
x_2 ( , , )
  proc {$ …} {Browse hello} end
  Q → x_2

P → x_1 ( , , )
```

```
x_1 ( , , )
  proc {$ …} {Q …} end
```

C. Varela; Adapted w/permission from S. Haridi and P. Van Roy
Procedure values (3)

- The semantic statement is \(\text{proc } \{ x \langle y_1 \rangle \ldots \langle y_n \rangle \} \langle s \rangle \text{ end, } E\)
- \(\langle y_1 \rangle \ldots \langle y_n \rangle\) are the (formal) parameters of the procedure
- Other free identifiers of \(\langle s \rangle\) are called external references \(\langle z_1 \rangle \ldots \langle z_k \rangle\)
- These are defined by the environment \(E\) where the procedure is declared (lexical scoping)
- The contextual environment of the procedure \(CE\) is \(E \mid \{\langle z_1 \rangle \ldots \langle z_k \rangle\}\)
- When the procedure is called \(CE\) is used to construct the environment of \(\langle s \rangle\)
Procedure values (4)

- Procedure values are pairs:
  \[
  (\text{proc} \{$ y_1 \ldots y_n \} \langle s \rangle \text{ end }, CE) \]

- They are stored in the store just as any other value

\[
\begin{align*}
\text{proc} \{$ y_1 \ldots y_n \} \langle s \rangle \\
\text{end}, \\
CE)
\end{align*}
\]
Procedure application

- The semantic statement is
  \[(\{\langle x \rangle \langle y_1 \rangle \ldots \langle y_n \rangle \}, E)\]
- The activation condition \(E(\langle x \rangle)\) is true:
  - If \(E(\langle x \rangle)\) is not procedure value, or a procedure with arity that is not equal \(n\), raise an error
  - \(E(\langle x \rangle)\) is \(\text{proc} \{\$ \langle z_1 \rangle \ldots \langle z_n \rangle\} \langle s \rangle \text{ end}, CE\), push
    \[(\langle s \rangle, CE + \{\langle z_1 \rangle \rightarrow E(\langle y_1 \rangle) \ldots \langle z_n \rangle \rightarrow E(\langle y_n \rangle)\})\]
    on the stack
- The activation condition \(E(\langle x \rangle)\) is false: suspend
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.

```plaintext
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

- 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 >= 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
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 \{\text{FoldL} Xs F U\}
  case Xs
  of nil then U
  [] X|Xr then \{\text{FoldL} Xr F \{F X U\}\}
  end
end

Assume a list \([x1 \ x2 \ x3 \ ....]\)

\[S0 \to S1 \to S2\]

\[U \to \{F \ x1 \ U\} \to \{F \ x2 \ \{F \ x1 \ U\}\} \to ....\to\]
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' :: (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 \([e_1 \ e_2 \ldots e_n]\) and a binary function \(\triangleright\), with unit \(U\), the previous folding functions do the following:

\[
\begin{align*}
& (e_1 \triangleright \ldots (e_{n-1} \triangleright (e_n \triangleright U))\ldots) \quad \text{fold right} \\
& (e_n \triangleright \ldots (e_2 \triangleright (e_1 \triangleright U))\ldots) \quad \text{fold left}
\end{align*}
\]

But there are two other possibilities:

\[
\begin{align*}
& (...((U \triangleright e_n) \triangleright e_{n-1})\ldots \triangleright e_1) \quad \text{fold right unit left} \\
& (...((U \triangleright e_1) \triangleright e_2)\ldots \triangleright e_n) \quad \text{fold left unit left}
\end{align*}
\]
FoldL unit left in Haskell

\[
\text{foldlul} :: (b\rightarrow a\rightarrow b) \rightarrow b \rightarrow [a] \rightarrow b
\]

\[
\text{foldlul} \_ u [] = u
\]

\[
\text{foldlul} f u (h:t) = \text{foldlul} f (f u h) t
\]

Notice the unit \( u \) is of type \( b \), list elements are of type \( a \), and the function \( f \) is of type \( b\rightarrow a\rightarrow b \).
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

\[
\begin{align*}
\text{filter'} & : (\text{a} \to \text{Bool}) \to \text{[a]} \to \text{[a]} \\
\text{filter'} \_ \ [\_] & = [] \\
\text{filter'} \ p \ (\text{h:t}) & = \text{if } p \ h \ \text{then } \text{h:filter'} \ p \ t \\
& \quad \quad \text{else } \text{filter'} \ p \ t
\end{align*}
\]
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

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

Call {P T} at each node T
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

\[
\text{fun lazy } \{\text{From } N\} \\
N \mid \{\text{From } N+1\} \\
\text{end}
\]

\[
\text{fun } \{\text{From } N\} \\
\text{fun } \{\$\} N \mid \{\text{From } N+1\} \text{ end} \\
\text{end}
\]
Exercises

23. Define an \texttt{IncList} function to take a list of numbers and increment all its values, using the \texttt{Map} control abstraction. For example:

\begin{verbatim}
{IncList [3 1 7]} => [4 2 8]
\end{verbatim}

24. Create a higher-order \texttt{MapReduce} function that takes as input two functions corresponding to \texttt{Map} and \texttt{Reduce} respectively, and returns a function to perform the composition. Illustrate your \texttt{MapReduce} function with an example.

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