Higher-Order Programming:
Iterative computation (CTM Section 3.2)
Closures, procedural abstraction, genericity, instantiation, embedding (CTM Section 3.6.1)

Carlos Varela
RPI
September 13, 2016

Adapted with permission from:
Seif Haridi
KTH
Peter Van Roy
UCL
Functions

• Compute the factorial function:
  \[ n! = 1 \times 2 \times \cdots \times (n - 1) \times n \]

• Start with the mathematical definition

  \[
  \text{declare}
  \text{fun } \{\text{Fact } N\}
  \quad \text{if } N == 0 \quad \text{then } 1 \quad \text{else } N \times \{\text{Fact } N - 1\} \quad \text{end}
  \text{end}
  \]

• Fact is declared in the environment

• Try large factorial \{Browse \{\text{Fact } 100\}\}
Functions in Haskell

factorial :: Integer -> Integer
factorial 0 = 1
factorial n | n > 0 = n * factorial (n-1)
Structured data (lists)

- A list is a sequence of elements:
  
  \[1 4 6 4 1\]
  
- The empty list is written `nil`

- Lists are created by means of `”|”` (cons)

```declare
H=1
T = [2 3 4 5]
{Browse H|T}  % This will show [1 2 3 4 5]
```
Lists (2)

• Taking lists apart (selecting components)
• A cons has two components: a head, and a tail

```
declare L = [5 6 7 8]
L.1 gives 5
L.2 gives [6 7 8]
```
Pattern matching

• Another way to take a list apart is by use of pattern matching with a case instruction

```plaintext
case L of H|T then {Browse H} {Browse T}
   else {Browse ‘empty list’}
end
```
Lists in Haskell

• A list is a sequence of elements:
  \([1,4,6,4,1]\)
• The empty list is written \([]\)
• Lists are created by means of "\(" (cons)

```haskell
let h = 1
let t = [2,3,4,5]
h:t -- This will show [1,2,3,4,5]
```
Lists in Haskell (2)

- Taking lists apart (selecting components)
- A cons has two components: a head, and a tail

\[
\text{let } l = [5,6,7,8] \\
\text{head } l \text{ gives } 5 \\
\text{tail } l \text{ gives } [6,7,8]
\]
Another way to take a list apart is by use of pattern matching with a case instruction:

```haskell
case l of (h:t) -> h:t
        []    -> []
end
```

Or more typically as part of a function definition:

```haskell
id (h:t) -> h:t
id []    -> []
```
Functions over lists

declare
fun {Pascal N}
  if N==1 then [1]
  else
    {AddList
      {ShiftLeft {Pascal N-1}}
      {ShiftRight {Pascal N-1}}}]
  end
end
Functions over lists (2)

fun {ShiftLeft L}
  case L of H|T then
    H|{ShiftLeft T}
  else [0] end
end

fun {ShiftRight L} 0|L end

fun {AddList L1 L2}
  case L1 of H1|T1 then
    case L2 of H2|T2 then
      H1+H2|{AddList T1 T2}
    end
  else nil end
end
--- Pascal triangle row

```haskell
pascal :: Integer -> [Integer]
pascal 1 = [1]
pascal n = addList (shiftLeft (pascal (n-1)))
         (shiftRight (pascal (n-1)))

where
    shiftLeft []    = [0]
    shiftLeft (h:t) = h:shiftLeft t
    shiftRight l    = 0:l
    addList [] []   = []
    addList (h1:t1) (h2:t2) = (h1+h2):addList t1 t2
```
Complexity

• Pascal runs very slow, try \{Pascal 24\}
• \{Pascal 20\} calls: \{Pascal 19\} twice, \{Pascal 18\} four times, \{Pascal 17\} eight times, ..., \{Pascal 1\} $2^{19}$ times
• Execution time of a program up to a constant factor is called the program’s time complexity.
• Time complexity of \{Pascal N\} is proportional to $2^N$ (exponential)
• Programs with exponential time complexity are impractical

```pascal
declare
fun \{Pascal N\} =
if N==1 then [1]
else
{AddList \{ShiftLeft \{Pascal N-1\}\}
{ShiftRight \{Pascal N-1\}\}}
end
end
```
Faster Pascal

- Introduce a local variable L
- Compute \{FastPascal N-1\} only once
- Try with 30 rows.
- FastPascal is called N times, each time a list on the average of size N/2 is processed
- The time complexity is proportional to N² (polynomial)
- Low order polynomial programs are practical.

```plaintext
fun \{FastPascal N\}
  if N==1 then [1]
  else
    local L in
    L=\{FastPascal N-1\}
    \{AddList \{ShiftLeft L\} \{ShiftRight L\}\}
  end
end
end
```
Iterative computation

- An iterative computation is one whose execution stack is bounded by a constant, independent of the length of the computation.
- Iterative computation starts with an initial state $S_0$, and transforms the state in a number of steps until a final state $S_{\text{final}}$ is reached:

$$S_0 \rightarrow S_1 \rightarrow \ldots \rightarrow S_{\text{final}}$$
The general scheme

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

• \text{IsDone and Transform are problem dependent}
The computation model

- STACK : \[ R=\{\text{Iterate } S_0\} \]
- STACK : \[ S_1 = \{\text{Transform } S_0\}, \quad R=\{\text{Iterate } S_1\} \]

- STACK : \[ R=\{\text{Iterate } S_i\} \]
- STACK : \[ S_{i+1} = \{\text{Transform } S_i\}, \quad R=\{\text{Iterate } S_{i+1}\} \]

- STACK : \[ R=\{\text{Iterate } S_{i+1}\} \]
Newton’s method for the square root of a positive real number

- Given a real number \( x \), start with a guess \( g \), and improve this guess iteratively until it is accurate enough.
- The improved guess \( g' \) is the average of \( g \) and \( x/g \):
  \[
g' = \frac{(g + x / g)}{2}
\]
- \( \varepsilon = g - \sqrt{x} \)
- \( \varepsilon' = g' - \sqrt{x} \)

For \( g' \) to be a better guess than \( g \): \( \varepsilon' < \varepsilon \)

\[
\varepsilon' = g' - \sqrt{x} = \frac{(g + x / g)}{2} - \sqrt{x} = \frac{\varepsilon^2}{2g}
\]

i.e. \( \frac{\varepsilon^2}{2g} < \varepsilon \), \( \frac{\varepsilon}{2g} < 1 \)

i.e. \( \varepsilon < 2g \), \( g - \sqrt{x} < 2g \), \( 0 < g + \sqrt{x} \)
Newton’s method for the square root of a positive real number

- Given a real number $x$, start with a guess $g$, and improve this guess iteratively until it is accurate enough.
- The improved guess $g'$ is the average of $g$ and $x/g$:
- Accurate enough is defined as:

$$\frac{|x - g^2|}{x} < 0.00001$$
fun {SqrtIter Guess X}
  if {GoodEnough Guess X} then Guess
  else
    Guess1 = {Improve Guess X} in
    {SqrtIter Guess1 X}
  end
end

• Compare to the general scheme:
  – The state is the pair Guess and X
  – *IsDone* is implemented by the procedure GoodEnough
  – *Transform* is implemented by the procedure Improve
The program version 1

fun {Sqrt X}
  Guess = 1.0
in {SqrtIter Guess X}
end

fun {SqrtIter Guess X}
  if {GoodEnough Guess X} then
    Guess
  else
    {SqrtIter {Improve Guess X} X}
  end
end

fun {Improve Guess X}
  (Guess + X/Guess)/2.0
end

fun {GoodEnough Guess X}
  {Abs X - Guess*Guess}/X < 0.00001
end
Using local procedures

• The main procedure Sqrt uses the helper procedures SqrtIter, GoodEnough, Improve, and Abs
• SqrtIter is only needed inside Sqrt
• GoodEnough and Improve are only needed inside SqrtIter
• Abs (absolute value) is a general utility
• The general idea is that helper procedures should not be visible globally, but only locally
local
  fun {SqrtIter Guess X}
    if {GoodEnough Guess X} then Guess
    else {SqrtIter {Improve Guess X} X} end
  end
fun {Improve Guess X}
  (Guess + X/Guess)/2.0
end
fun {GoodEnough Guess X}
  {Abs X - Guess*Guess}/X < 0.000001
end
in
  fun {Sqrt X}
    Guess = 1.0
  in {SqrtIter Guess X} end
end
Sqrt version 3

• Define GoodEnough and Improve inside SqrtIter

```plaintext
local
  fun {SqrtIter Guess X}
    fun {Improve}
      (Guess + X/Guess)/2.0
    end
    fun {GoodEnough}
      {Abs X - Guess*Guess}/X < 0.000001
    end
  in
    if {GoodEnough} then Guess
    else {SqrtIter {Improve} X} end
  end
in
  fun {Sqrt X}
    Guess = 1.0 in
    {SqrtIter Guess X}
  end
end
```

C. Varela; Adapted w/permission from S. Haridi and P. Van Roy


**Sqrt version 3**

- Define `GoodEnough` and `Improve` inside `SqrtIter`

```plaintext
local
    fun \{SqrtIter \text{Guess} \text{X}\}
        fun \{Improve\}
            (\text{Guess} + \text{X}/\text{Guess})/2.0
        end
        fun \{GoodEnough\}
            \{Abs \text{X} - \text{Guess}^{\text{Guess}}\}/\text{X} < 0.000001
        end
    in
        if \{GoodEnough\} then \text{Guess}
        else \{SqrtIter \{Improve\} \text{X}\} end
    end
in
    fun \{Sqrt \text{X}\}
        \text{Guess} = 1.0 in
        \{SqrtIter \text{Guess} \text{X}\}
    end
end
```

The program has a single drawback: on each iteration two procedure values are created, one for `Improve` and one for `GoodEnough`
fun \{Sqrt X\} 
    fun \{Improve Guess\} 
        (Guess + X/Guess)/2.0 
    end 
    fun \{GoodEnough Guess\} 
        \{Abs X - Guess*Guess\}/X < 0.000001 
    end 
    fun \{SqrtIter Guess\} 
        if \{GoodEnough Guess\} then Guess 
        else \{SqrtIter \{Improve Guess\}\} end 
    end 
    Guess = 1.0 
in \{SqrtIter Guess\} 
end 

The final version is a compromise between abstraction and efficiency
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

• \textit{IsDone} and \textit{Transform} are problem dependent
From a general scheme to a control abstraction (2)

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 Iterate abstraction

fun {Sqrt X}
    fun {Improve Guess}
        (Guess + X/Guess)/2.0
    end
    fun {GoodEnough Guess}
        {Abs X - Guess*Guess}/X < 0.000001
    end
    Guess = 1.0
in
    {Iterate Guess GoodEnough Improve}
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

Iterate could become a linguistic abstraction
**Sqrt in Haskell**

```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.
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
Procedural abstraction

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

C. Varela; Adapted w/permission from S. Haridi and P. Van Roy
**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
```
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
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.
Exercises

18. CTM Exercise 3.10.5 (page 230)
19. Suppose you have two sorted lists. Merging is a simple method to obtain an again sorted list containing the elements from both lists. Write a Merge function that is generic with respect to the order relation.
20. Instantiate the FoldFactory to create a ProductList function to multiply all the elements of a list.
21. Create an AddFactory function that takes a list of numbers and returns a list of functions that can add by those numbers, e.g. \{AddFactory [1 2]\} == [Inc1 Inc2] where Inc1 and Inc2 are functions to increment a number by 1 and 2 respectively, e.g., \{Inc2 3\} => 5.
22. Implement exercises 18-21 in both Oz and Haskell.