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

Carlos Varela
RPI
September 15, 2017

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

• *IsDone* and *Transform* are problem dependent
The computation model

- STACK : [ R={Iterate S₀} ]
- STACK : [ S₁ = {Transform S₀},
  R={Iterate S₁} ]

- STACK : [ R={Iterate Sᵢ} ]
- STACK : [ Sᵢ₊₁ = {Transform Sᵢ},
  R={Iterate Sᵢ₊₁} ]

- STACK : [ R={Iterate Sᵢ₊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. \( \frac{\varepsilon^2}{2g} < \varepsilon \), \( \varepsilon / 2g < 1 \)

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\ \text{Guess} \ X\}
    \begin{align*}
    \text{if} & \ \{\text{GoodEnough Guess X}\} \ \text{then} \ \text{Guess} \\
    \text{else} & \ \text{Guess1} = \{\text{Improve Guess X}\} \ \text{in} \\
    & \ \{\text{SqrtIter Guess1 X}\}
    \end{align*}
end
end

• Compare to the general scheme:
  – The state is the pair \text{Guess} and \ X
  – \text{IsDone} is implemented by the procedure \text{GoodEnough}
  – \text{Transform} is implemented by the procedure \text{Improve}
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 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
```

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
From a general scheme to a control abstraction (1)

fun \{\text{Iterate } S_i\} \\
\hspace{1em} \text{if } \{\text{IsDone } S_i\} \text{ then } S_i \\
\hspace{1em} \text{else } S_{i+1} \text{ in} \\
\hspace{2em} S_{i+1} = \{\text{Transform } S_i\} \\
\hspace{3em} \{\text{Iterate } S_{i+1}\} \\
\hspace{2em} \text{end} \\
\hspace{1em} \text{end} \\

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

\[
\text{fun } \{ \text{Iterate } S \ \text{IsDone Transform} \} \\
\quad \text{if } \{ \text{IsDone } S \} \text{ then } S \\
\quad \text{else } S_1 \text{ in} \\
\quad \quad S_1 = \{ \text{Transform } S \} \\
\quad \quad \{ \text{Iterate } S_1 \ \text{IsDone Transform} \} \\
\quad \text{end} \\
\text{end}
\]

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

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

```ml
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.