Higher-Order Programming:

Iterative computation (CTM Section 3.2)
Closures, functional abstraction, genericity, instantiation, embedding (CTM Section 3.6.1)

Carlos Varela

RPI

September 17, 2024

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_{final} is reached:

$$S_0 \rightarrow S_1 \rightarrow \ldots \rightarrow S_{final}$$

From a general scheme to a control abstraction (1)

iterate' s =

if *isDone* s_i then s_i

else let s_{i+1} = *transform* s_i in

iterate' s_{i+1}

- *isDone* and *transform* are problem dependent

From a general scheme to a control abstraction (2)

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

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

Sqrt using the Iterate abstraction

```
sqrt5 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 the control abstraction

```
sqrt6 x = iterate' 1
                  (\g -> (abs (x - g*g))/x < 0.00001)
                  (\g -> (g + x/g)/2.0)
```

iterate' can be a linguistic abstraction

Sqrt using infinite lists

```
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.

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.

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))
```

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 []    = 0
sumlist (h:t) = h+sumlist t
```

```
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
```

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.

Exercises

15. CTM Exercise 3.10.2 (page 230)
16. CTM Exercise 3.10.3 (page 230)
17. Develop a control abstraction for iterating over a list of elements.
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.

Exercises

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. Look at the types of sumlist and foldr. How are they different? Come up with a usage of foldr that requires the more general type.