Declarative Programming Techniques

Higher-Order Programming (VRH 3.6)
Abstract Data Types (VRH 3.7)

Carlos Varela
RPI
October 12, 2006
Adapted with permission from:
Seif Haridi
KTH
Peter Van Roy
UCL

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
- Control abstractions
  - Integer and list loops, accumulator loops, folding a list (left and right)
- Data-driven techniques
  - List filtering, tree folding
- Explicit lazy evaluation, currying
- 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 = proc {$} <s> end
  - Executing {P} has exactly the same effect as executing <s>

A common limitation

- Most popular imperative languages (C, C++, Java) 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
- 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 programmers on systems that have no garbage collection
- It means that contextual environments cannot be created, since they would be full of dangling pointers

Procedural abstraction

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

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
Genericity

- Replace specific entities (zero 0 and addition +) by function arguments
- The same routine can do the sum, the product, the logical or, etc.

```javascript
fun {SumList L}
  case L
  of nil then 0
      L2 then X{SumList L2}
  end
end
```

```javascript
fun {FoldR L F U}
  case L
  of nil then U
      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

```javascript
fun {FoldFactory F U}
  fun {FoldR L F U}
    case L
    of nil then U
        L2 then {F X {FoldR L2 F U}}
    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.

```javascript
proc {For I J P}
  if I >= J then skip
  else {P I} {For I+1 J P}
  end
end
```

Control Abstractions

```javascript
declare
proc {For I J P}
  if I >= J then skip
  else {P I} {For I+1 J P}
  end
end
```

Control Abstractions

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

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

Control abstractions

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

```javascript
Assume a list [x1 x2 x3 ....]
S0 -> S1 -> S2
U -> {F x1 U} -> {F x2 {F x1 U}} -> ....->
```
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
end

List-based techniques

```
fun {Filter Xs P}
case Xs
of nil then nil
| X|Xr then {P X} then X|{Filter Xr P}
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.

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
```

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

Currying

- Currying is a technique that can simplify programs that heavily use higher-order programming.
- The idea: function of n arguments => 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
```

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

Abstract data types

- A datatype is a set of values and an associated set of operations
- A datatype is abstract only if it is completely described by its set of operations regardless of its implementation
- This means that it is possible to change the implementation of the datatype without changing its use
- The datatype is thus described by a set of procedures
- These operations are the only thing that a user of the abstraction can assume
**Example: A Stack**

- Assume we want to define a new datatype \( \langle \text{stack } T \rangle \) whose elements are of any type \( T \)
  
  ```
  fun \{NewStack\}(\langle \text{stack } T \rangle)
  fun \{Push\}(\langle \text{stack } T \rangle)(T);(\langle \text{stack } T \rangle)
  fun \{Pop\}(\langle \text{stack } T \rangle)(T);(\langle \text{stack } T \rangle)
  fun \{IsEmpty\}(\langle \text{stack } T \rangle);(\langle \text{Bool} \rangle)
  ```

- These operations normally satisfy certain conditions:
  
  \[
  \{\text{IsEmpty } \{\text{NewStack}\}\} = \text{true}
  \]
  
  for any \( E \) and \( S0 \), \( S1 = \{\text{Push } S0\} E \) and \( S0 = \{\text{Pop } S1\} E \) hold

*Example: A Stack*

*Stack (implementation)*

```plaintext
fun \{NewStack\} nil end
fun \{Push S\} E|S end
fun \{Pop S\} case S of X|S1 then E = X S1 end end
fun \{IsEmpty S\} S==nil end
```

*Stack (another implementation)*

```plaintext
fun \{NewStack\} nil end
fun \{Push S\} E|S end
fun \{Pop S\} case S of X|S1 then E = X S1 end end
fun \{IsEmpty S\} S==nil end
```

*Implementation*

```plaintext
fun \{Put Ds Key Value\}
  case Ds of nil then [\{Key=Value\}]
  | (K#V)|Dr and then Key==K then (K#V)|Dr
  | (K#V)|Dr and then K>Key then (K#V)|Dr
  | (K#V)|Dr and then K<Key then (K#V)|{Put Dr Key Value} end end
```

*Implementation*

```plaintext
fun \{CondGet Ds Key Default\}
  case Ds of nil then Default
  | (K#V)|Dr and then Key==K then V
  | (K#V)|Dr and then K>Key then Default
  | (K#V)|Dr and then K<Key then \{CondGet Dr Key Default\} end end
  fun \{Domain Ds\}
  (Map Ds fun (\{K#_\} K end end
```
Further implementations

- Because of abstraction, we can replace the dictionary ADT implementation using a list, whose complexity is linear (i.e., O(n)), for a binary tree implementation with logarithmic operations (i.e., O(log(n)).
- Data abstraction makes clients of the ADT unaware (other than through perceived efficiency) of the internal implementation of the data type.
- It is important that clients do not use anything about the internal representation of the data type (e.g., using Length Dictionary to get the size of the dictionary). Using only the interface (defined ADT operations) ensures that different implementations can be used in the future.

Secure abstract data types: Stack is not secure

- The representation of the stack is visible: [a b c d]
- Anyone can use an incorrect representation, i.e., by passing other language entities to the stack operation, causing it to malfunction (like a|b|X or Y=a|b|Y)
- Anyone can write new operations on stacks, thus breaking the abstraction-representation barrier
- How can we guarantee that the representation is invisible?

Secure abstract data types II

- The representation of the stack is visible:

```
[a b c d]
```
- Anyone can use an incorrect representation, i.e., by passing other language entities to the stack operation, causing it to malfunction (like a|b|X or Y=a|b|Y)
- Anyone can write new operations on stacks, thus breaking the abstraction-representation barrier
- How can we guarantee that the representation is invisible?

Secure abstract data types III

- The model can be extended. Here are two ways:
  - By adding a new basic type, an unforgeable constant called a name
  - By adding encapsulated state.
- A name is like an atom except that it cannot be typed in on a keyboard or printed!
- The only way to have a name is if one is given it explicitly
- There are just two operations on names:
  - N=(NewName) : returns a fresh name
  - N1==N2 : returns true or false

Secure abstract datatypes IV

- We want to « wrap » and « unwrap » values
- Let us use names to define a wrapper & unwrapper

```
proc [NewWrapper ?Wrap ?Unwrap]
  Key=(NewName)
  in
  fun (Wrap X)
    fun (K|K=Key then X end end)
  end
  fun (Unwrap C)
    C (Key)
  end
end
```

Secure abstract data types: A secure stack

With the wrapper & unwrapper we can build a secure stack

```
local [Wrap Unwrap in]
  (NewWrapper Wrap Unwrap)
  fun (NewStack (Wrap nil) end)
  fun (Push S E) (Wrap E)(Unwrap S) end
  fun (Pop S E)
    case (Unwrap S) of X|S1 then E*X S1 end
  end
  fun (isEmpty S) (Unwrap S)==nil end
end
```
Capabilities and security

- We say a computation is secure if it has well-defined and controllable properties, independent of the existence of other (possibly malicious) entities (either computations or humans) in the system.
- What properties must a language have to be secure?
- One way to make a language secure is to base it on capabilities:
  - A capability is an unforgeable language entity (a ticket s) that gives its owner the right to perform a particular action and only that action.
  - In our model, all values are capabilities (records, numbers, procedures, names) since they give the right to perform operations on the values.
  - Having a procedure gives the right to call that procedure. Procedures are very general capabilities, since what they do depend on their argument.
  - Using names as procedure arguments allows very precise control of rights; for example, it allows us to build secure abstract datatypes.
-Capabilities originated in operating systems research.
- A capability can give a process the right to create a file in some directory.

Secure abstract datatypes V

- We add two new concepts to the computation model:
  - \{NewChunk Record\}
  -  \{NewName\}
-\{NewChunk foo\(a:1 b:2\)\} makes impossible to access the third component, if you do not know the arity.
- Returns what?

Secure abstract datatypes VI

```
proc (NewWrapper ?Wrap ?Unwrap)
  Key=(NewName)
  in
  fun (Wrap X)
    (NewChunk foo(Key:X))
  end
  fun (Unwrap C)
    C.Key
  end
end
```

Secure abstract data types: Another secure stack

With the new wrapper & unwrapper we can build another secure stack (since we only use the interface to wrap and unwrap, the code is identical to the one using higher-order programming):

```
local Wrap Unwrap in
  (NewWrapper Wrap Unwrap)
  fun (NewStack [Wrap nil]) end
  fun (Push S E) (Wrap E[Unwrap S]) end
  fun (Pop S E)
    case (Unwrap S) of
      X|S1 then E\x (Wrap S1) end
      S1 end
  end
  fun (IsEmpty S) (Unwrap S)=nil end
end
```

Exercises

47. Implement the function \{FilterAnd Xs P Q\} that returns all elements of Xs in order for which P and Q return true. Hint: Use \{Filter Xs P\}.
48. Compute the maximum element from a nonempty list of numbers by folding.
49. *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.
50. *VRH Exercise 3.10.17 (pg. 232). You do not need to implement it using gump, simply specify how you would add currying to Oz (syntax and semantics).