Declarative Programming Techniques

Declarativeness, iterative computation (VRH 3.1-3.2)
Higher-order programming (VRH 3.6)
Abstract data types (VRH 3.7)

Carlos Varela
Rensselaer Polytechnic Institute
November 15, 2007

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

Overview

- What is declarativeness?
  - Classification,
  - Advantages for large and small programs
- Control Abstractions
  - Iterative programs
- Higher-Order Programming
  - Procedural abstraction
  - Genericity
  - Instantiation
  - Embedding
- Abstract data types
  - Encapsulation
  - Security

Declarative operations (1)

- An operation is declarative if whenever it is called with the same arguments, it returns the same results independent of any other computation state
- A declarative operation is:
  - Independent (depends only on its arguments, nothing else)
  - Stateless (no internal state is remembered between calls)
  - Deterministic (call with same operations always give same results)
- Declarative operations can be composed together to yield other declarative components
  - All basic operations of the declarative model are declarative and combining them always gives declarative components

Declarative operations (2)

Why declarative components (1)

- There are two reasons why they are important:
  - (Programming in the large) A declarative component can be written, tested, and proved correct independent of other components and of its own past history.
    - The complexity (reasoning complexity) of a program composed of declarative components is the sum of the complexity of the components
    - In general the reasoning complexity of programs that are composed of nondeclarative components explodes because of the intimate interaction between components
  - (Programming in the small) Programs written in the declarative model are much easier to reason about than programs written in more expressive models (e.g., an object-oriented model).
    - Simple algebraic and logical reasoning techniques can be used

Why declarative components (2)

- Since declarative components are mathematical functions, algebraic reasoning is possible i.e. substituting equals for equals
- The declarative model of chapter 2 guarantees that all programs written are declarative
- Declarative components can be written in models that allow stateful data types, but there is no guarantee

Given \( f(a) = a^2 \)
We can replace \( f(a) \) in any other equation
\( b = 7f(a)^3 \) becomes \( b = 7a^6 \)
Classification of declarative programming

- The word *declarative* means many things to many people. Let’s try to eliminate the confusion.
- The basic intuition is to program by defining the *what* without explaining the *how*.

Descriptive language

\[
\begin{align*}
(s) &\triangleright \text{skip} & \text{empty statement} \\
   &\triangleright (x) & \text{variable-variable binding} \\
   &\triangleright (x) = (y) & \text{variable-value binding} \\
   &\triangleright (\text{local } (x) \in (s)) & \text{sequential composition} \\
   &\triangleright (\text{local } (x) \in (s) \text{ end}) & \text{declaration}
\end{align*}
\]

Other descriptive languages include HTML and XML.

Kernel language

The following defines the syntax of a statement, \((s)\) denotes a statement

\[
\begin{align*}
(s) &\triangleright \text{skip} & \text{empty statement} \\
   &\triangleright (x) = (y) & \text{variable-variable binding} \\
   &\triangleright (x) = (v) & \text{variable-value binding} \\
   &\triangleright (s_1) (s_2) & \text{sequential composition} \\
   &\triangleright (\text{local } (x) \in (s)) & \text{declaration} \\
   &\triangleright (\text{proc } \{(x) \ y_1 \ldots y_n\} \text{ end}) & \text{procedure introduction} \\
   &\triangleright (\text{if } (x) \text{ then } (s_1) \text{ else } (s_2)) & \text{conditional} \\
   &\triangleright (\text{case } (x) \text{ of } (\text{pattern}) \text{ then } (s_k) \text{ end}) & \text{procedure application} \\
   &\triangleright (\text{case } (x) \text{ of } (\text{pattern}) \text{ else } (s_k) \text{ end}) & \text{pattern matching}
\end{align*}
\]

Why the KL is declarative

- All basic operations are declarative
- Given the components (sub-statements) are declarative,
  - sequential composition
  - local statement
  - procedure definition
  - procedure call
  - if statement
  - case statement
  - all declarative (independent, stateless, deterministic).

Iterative computation

- An iterative computation is a 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:

\[
S_0 \xrightarrow{\text{...}} S_1 \xrightarrow{\text{...}} S_{\text{final}}
\]
The general scheme

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

• \(IsDone\) and \(Transform\) are problem dependent

The computation model

• STACK : \[ R = \{Iterate } S_0 \text{\} \]
• STACK : \[ S_1 = \{Transform } S_0 \text{\}, \ R = \{Iterate } S_1 \text{\} \]
• STACK : \[ R = \{Iterate } S_{i+1} \text{\} \]
• STACK : \[ S_{i+1} = \{Transform } S_i \text{\}, \ R = \{Iterate } S_{i+1} \text{\} \]
• STACK : \[ R = \{Iterate } S_{i+1} \text{\} \]

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 \(x/g\):
  \[
g' = \frac{g + x/g}{2} \\
\]
  \[
\epsilon = g' - \sqrt{x} \\
\]
• \(\epsilon\) to be a better guess than \(g\): \(\epsilon' < \epsilon\)
  \[
\epsilon' = g' - \sqrt{x} = \frac{g + x/g}{2} - \sqrt{x} = \frac{\epsilon^2}{2g} < \epsilon, \quad \epsilon / 2g < 1 \\
\]
  \[
i.e. \quad \epsilon < 2g, \quad g = \sqrt{x} < 2g, \quad 0 < g + \sqrt{x} \\
\]

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 \(x/g\):
  \[
g' = \frac{g + x/g}{2} \\
\]
• Accurate enough is defined as:
  \[
| x - g'^2 | / x < 0.00001 \\
\]

SqrtIter

\[
\text{fun \{SqrtIter } \text{Guess } X \text{\} \ if \ {GoodEnough } \text{Guess } X \text{\} then } \text{Guess} \\
\text{else } \text{Guess1} = \{Improve } \text{Guess } X \text{\} \text{ in} \\
\{SqrtIter } \text{Guess1 } X \text{\} \text{ end} \\
\text{end} \\
\text{end} \\
\]

• 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 \(Improve\)
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

Sqrt version 2

```plaintext
local
fun {Sqrt X}
  if {GoodEnough X} then
    X
  else
    {SqrtIter {Improve X} X}
  end
end

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

fun {GoodEnough X} {Abs X - Guess*Guess}/X < 0.000001 end

in
  Guess = 1.0
  {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.

Sqrt version 3

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

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

fun {GoodEnough X} {Abs X - Guess*Guess}/X < 0.000001 end

in
  Guess = 1.0
  {SqrtIter Guess X}
end
end
```

The final version is a compromise between abstraction and efficiency.

Sqrt final version

```plaintext
fun {Iterate S_i}
  if {IsDone S_i} then
    S_i
  else
    S_{i+1} in {Iterate S_{i+1}}
  end
end

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

The final version is a compromise between abstraction and efficiency.

From a general scheme to a control abstraction (1)

```plaintext
fun {Iterate S_i}
  if {IsDone S_i} then
    S_i
  else
    S_{i+1} in {Iterate S_{i+1}}
  end
end

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

The final version is a compromise between abstraction and efficiency.

- IsDone and 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 (GoodEnough Guess)
    (Abs X - Guess*Guess)/X < 0.000001
    Guess = 1.0
    in
    (Iterate S i+1)
  end
end

Sqrt using the Iterate abstraction

fun (Sqrt X)
  (Iterate 1.0)
  fun (G)
    (G + X/G)/2.0
  end
end

fun (AndThen B1 B2)
  if B1 then B2 else false
end

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, carrying
- 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} <s> end
  - Executing [P] has exactly the same effect as executing <s>
Procedural abstraction

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

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
  - They provide a predefined set of control abstractions (for, while loops, if-else)
- 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

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
```

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

```plaintext
fun {FoldFactory F U}
  fun {FoldR L F U}
    case L of
      nil then U
      X|L2 then {F X  {FoldR L2 F U}}
    end
  end
  in fun {$L} {FoldR L F U} 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

```plaintext
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

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

```plaintext
{ForAll [a b c d] 
proc {S 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

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

Assume a list \([x1 \ x2 \ x3 \ ...]\)  
\(S0 \rightarrow S1 \rightarrow S2\)  
\(U \rightarrow \{F \ x1 \ U\} \rightarrow \{F \ x2 \ \{F \ x1 \ U\}\} \rightarrow \ldots\)

### Control abstractions

```plaintext
fun {Map Xs F}  
case Xs  
of nil then nil  
  X|Xr then  
  {F X}|{Map Xr F}  
end  
end
```

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

What does this program do?  
{Browse {FoldL [1 2 3] 
  fun {S \ X Y|X \ Y \ nil \}}}
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.

```plaintext
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
```

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 (stack T) whose elements are of any type T
  ```plaintext
  fun NewStack: Stack T end
  fun Push S E: S|E end
  fun Pop S E: case S of X|S1 then E = X S1 end end
  fun IsEmpty S: S==nil end
  ```
- These operations normally satisfy certain conditions:
  ```plaintext
  isEmpty {NewStack} = true
  for any E and S0, S1={Push S0 E} and S0={Pop S1 E} hold
  {Pop {NewStack} E} raises error
  ```

Stack (implementation)

```plaintext
fun NewStack nil end
fun Push S E|S end
fun Pop S E 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 E case S of X|S1 then E = X S1 end end
fun IsEmpty S S==nil end
fun NewStack emptyStack end
fun Push S E stack(E S) end
fun Pop S E case S of stack(X S1) then E = X S1 end end
fun IsEmpty S S==emptyStack end
```

Dictionaries

- The datatype dictionary is a finite mapping from a set T to (value), where T is either (atom) or (integer)
- fun (NewDictionary) returns an empty mapping
- fun (Put D Key Value) returns a dictionary identical to D except Key is mapped to Value
- fun (CondGet D Key Default) returns the value corresponding to Key in D, otherwise returns Default
- fun (Domain D) returns a list of the keys in D
Implementation

```haskell
fun {Put Ds Key Value}
case Ds
of nil then [Key#Value]
[|Dr

andthen Key==K then
(Key#Value) | Dr
[

andthen K>Key then
(Key#Value)|(K#V)|Dr
[

andthen K<Key then
(K#V)|{Put Dr Key Value}
end
end
```

Implementation

```haskell
fun {CondGet Ds Key Default}
case Ds
of nil then Default
[|Dr

andthen Key==K then
V
[

andthen K>Key then
Default
[

andthen K<Key then
{CondGet Dr Key Default}
end
end
fun {Domain Ds}
{Map Ds fun (K#V) 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

```haskell
fun (NewStack) nil end
fun (Push S E) E|S end
fun (Pop S E)
case S
of X|S1 then E=X S1 end
end
fun (IsEmpty S)
S==nil end
```

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 ab|N or Y=ab|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}
    if 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 (Wrap 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 depends 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]
    returns a value similar to record but its arity cannot be inspected
    recall {Arity foo(a:1 b:2)} is [a b]
  - [NewName]
    a function that returns a new symbolic (unforgeable, i.e. cannot be guessed) name
    foo(a:1 b:2 {NewName}:3) makes impossible to access the third component, if you do not know the arity
    - [NewChunk foo(a:1 b:2 {NewName}:3) ]
      - Returns what ?

Secure abstract datatypes VI

```
proc {NewWrapper ?Wrap ?Unwrap}
  Key={NewName}
in
  fun {Wrap X}
    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
  end
  fun {IsEmpty S} (Unwrap S)==nil end
end
```
76. Modify the Pascal function to use local functions for AddList, ShiftLeft, ShiftRight. Think about the abstraction and efficiency tradeoffs.

77. VRH Exercise 3.10.2 (page 230)
78. *VRH Exercise 3.10.3 (page 230)
79. *Develop a control abstraction for iterating over a list of elements.

80. Implement the function \{\text{FilterAnd Xs P Q}\} that returns all elements of Xs in order for which P and Q return true. Hint: Use \{\text{Filter Xs P}\}.

81. Compute the maximum element from a nonempty list of numbers by folding.

82. *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.

83. *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).