

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

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

1

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

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

2

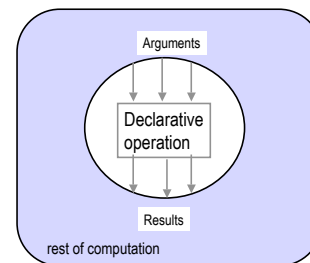
Declarative operations (1)

- 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

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

3

Declarative operations (2)



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

4

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

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

5

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
 $b = 7f(a)^2$ becomes $b = 7a^4$

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

6

Classification of declarative programming

```

graph LR
    DP[Declarative programming] --> D[Descriptive]
    DP --> P[Programmable]
    P --> O[Observational]
    P --> DEF[Definitional]
    DEF --> DM[Declarative model]
    DM --> FP[Functional programming]
    DM --> DLP[Deterministic logic programming]
    DM --> NLP[Nondeterministic logic 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*

C. Varela; Adapted w/permission from S. Haridi and P. Van Roy 7

Descriptive language

```

(s) ::= skip
      | (x) = (y)
      | (x) = (record)
      | (s1) (s2)
      | local (x) in (s1) end
  
```

empty statement
variable-variable binding
variable-value binding
sequential composition
declaration

Other descriptive languages include HTML and XML

C. Varela; Adapted w/permission from S. Haridi and P. Van Roy 8

Descriptive language

```

<person id = "530101-xxx">
  <name> Seif </name>
  <age> 48 </age>
</person>
  
```

Other descriptive languages include HTML and XML

C. Varela; Adapted w/permission from S. Haridi and P. Van Roy 9

Kernel language

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

```

(s) ::= skip
      | (x) = (y)
      | (x) = (v)
      | (s1) (s2)
      | local (x) in (s1) end
      | proc '{(x) (y1 ... (yn) }' (s1) end
      | if (x) then (s1) else (s2) end
      | '{ (x) (y1 ... (yn) }'
      | case (x) of (pattern) then (s1) else (s2) end
  
```

empty statement
variable-variable binding
variable-value binding
sequential composition
declaration
procedure introduction
conditional
procedure application
pattern matching

C. Varela; Adapted w/permission from S. Haridi and P. Van Roy 10

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

are all declarative (independent, stateless, deterministic).

C. Varela; Adapted w/permission from S. Haridi and P. Van Roy 11

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_{final} is reached:

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

C. Varela; Adapted w/permission from S. Haridi and P. Van Roy 12

The general scheme

```

fun {Iterate Si}
  if {IsDone Si} then Si
  else Si+1 in
    Si+1 = {Transform Si}
    {Iterate Si+1}
  end
end

```

- *IsDone* and *Transform* are problem dependent

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

13

The computation model

- STACK : [R={Iterate S₀}]
- STACK : [S₁ = {Transform S₀}, R={Iterate S₁}]
- STACK : [R={Iterate S_i}]
- STACK : [S_{i+1} = {Transform S_i}, R={Iterate S_{i+1}}]
- STACK : [R={Iterate S_{i+1}}]

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

14

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 :

$$g' = (g + x/g) / 2$$

$$\epsilon = g - \sqrt{x}$$

$$\epsilon' = g' - \sqrt{x}$$

For g' to be a better guess than g : $\epsilon' < \epsilon$

$$\epsilon' = g' - \sqrt{x} = (g + x/g) / 2 - \sqrt{x} = \epsilon^2 / 2g$$

$$\text{i.e. } \epsilon^2 / 2g < \epsilon, \quad \epsilon / 2g < 1$$

$$\text{i.e. } \epsilon < 2g, \quad g - \sqrt{x} < 2g, \quad 0 < g + \sqrt{x}$$

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

15

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:

$$|x - g^2| / x < 0.00001$$

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

16

SqrtIter

```

fun {SqrtIter Guess X}
  if {GoodEnough Guess X} then Guess
  else
    Guess1 = {Improve Guess X} in
      {SqrtIter Guess1 X}
    end
  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*

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

17

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

```

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

18

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

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

19

Sqrt version 2

```

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

```

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

20

Sqrt version 3

- Define GoodEnough and Improve inside SqrtIter

```

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

21

Sqrt version 3

- Define GoodEnough and Improve inside SqrtIter

```

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

22

The program has a single drawback: on each iteration two procedure values are created, one for Improve and one for GoodEnough

Sqrt final version

```

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

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

23

From a general scheme to a control abstraction (1)

```

fun {Iterate Si}
  if {IsDone Si} then Si
  else Si+1 in
    Si+1 = {Transform Si}
    {Iterate Si+1}
  end
end

```

- *IsDone* and *Transform* are problem dependent

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

24

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
end

fun {Iterate Si}
  if {IsDone Si} then Si
  else Si+1 in
    Si+1 = {Transform Si}
    {Iterate Si+1}
  end
end
end
  
```

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

25

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
end
  
```

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

26

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

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

27

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

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

28

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>

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

29

Procedural abstraction

```

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

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

30

Procedural abstraction

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

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

31

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 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

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

32

Genericity

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

```
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
```

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

33

Instantiation

```
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
```

- Instantiation is when a procedure returns a procedure value as its result
- Calling {FoldFactory fun {\$ A B} A+B end U} returns a function that behaves identically to SumList, which is an «instance» of a folding function

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

34

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.

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

35

Control Abstractions

```
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
```

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

36

Control Abstractions

```

proc {ForAll Xs P}
  case Xs
  of nil then skip
  [] X|Xr then
    {P X} {ForAll Xr P}
  end
end

{ForAll [a b c d]
  proc {$ I} {System.showInfo "the item is: " # I} end}

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

```

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

37

Control abstractions

```

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 → S1 → S2
U → {F x1 U} → {F x2 {F x1 U}} → ... →

```

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

38

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 nil}}

```

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

39

List-based techniques

```

fun {Map Xs F}
  case Xs
  of nil then nil
  [] X|Xr then
    {F X}|{Map Xr F}
  end
end

```

```

fun {Filter Xs P}
  case Xs
  of nil then nil
  [] X|Xr andthen {P X} then
    X|{Filter Xr P}
  end
end

```

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

40

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
end

Call {P T} at each node T

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
end

```

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

41

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.

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

42

Currying

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

\Rightarrow

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

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

43

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

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

44

Example: A Stack

- Assume we want to define a new datatype \langle stack $T \rangle$ whose elements are of any type T
`fun {NewStack}: \langle Stack $T \rangle$`
`fun {Push \langle Stack $T \rangle$ $T \rangle$: \langle Stack $T \rangle$`
`fun {Pop \langle Stack $T \rangle$ $T \rangle$: \langle Stack $T \rangle$`
`fun {IsEmpty \langle Stack $T \rangle$ }: \langle Bool`
- These operations normally satisfy certain conditions:
`{IsEmpty {NewStack}} = true`
for any E and $S0, S1 = \{Push\ S0\ E\}$ and $S0 = \{Pop\ S1\ E\}$ hold
`{Pop {NewStack} E}` raises error

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

45

Stack (implementation)

```
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
```

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

46

Stack (another implementation)

```
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
```

```
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
```

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

47

Dictionaries

- The datatype dictionary is a finite mapping from a set T to \langle value \rangle , where T is either \langle atom \rangle or \langle integer \rangle
- `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

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

48

Implementation

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

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

49

Implementation

```
fun {CondGet Ds Key Default}
  case Ds
  of nil then Default
  [] (K#V)|Dr andthen Key==K then
    V
  [] (K#V)|Dr andthen K>Key then
    Default
  [] (K#V)|Dr andthen K<Key then
    {CondGet Dr Key Default}
  end
end
fun {Domain Ds}
  {Map Ds fun {$ K#_} K end}
end
```

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

50

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.

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

51

Secure abstract data types: Stack is not secure

```
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
```

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

52

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?

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

53

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

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

54

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} if K==Key then X end end
  end
  fun {Unwrap C}
    {C Key}
  end
end
```

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

55

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
```

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

56

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 (« ticket ») 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 data types
- Capabilities originated in operating systems research
 - A capability can give a process the right to create a file in some directory

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

57

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 ?

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

58

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
```

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

59

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
```

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

60

Exercises

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.

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

61

Exercises

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).

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

62