

## Declarative Programming Techniques

Accumulators, Difference Lists (VRH 3.4.3-3.4.4)

Carlos Varela

RPI

Adapted with permission from:

Seif Haridi

KTH

Peter Van Roy

UCL

November 20, 2006

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

1

## Accumulators

- *Accumulator programming* is a way to handle state in declarative programs. It is a programming technique that uses arguments to carry state, transform the state, and pass it to the next procedure.
- Assume that the state  $S$  consists of a number of components to be transformed individually:  
 $S = (X, Y, Z, \dots)$
- For each procedure  $P$ , each state component is made into a pair, the first component is the *input* state and the second component is the output state after  $P$  has terminated
- $S$  is represented as  
 $(X_{in}, X_{out}, Y_{in}, Y_{out}, Z_{in}, Z_{out}, \dots)$

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

2

## A Trivial Example

```
proc {Increment N0 N}
  N = N0 + 1
end
```

**Increment** takes  $N0$  as the input and produces  $N$  as the output by adding 1 to  $N0$ .

```
proc {Square N0 N}
  N = N0 * N0
end
```

**Square** takes  $N0$  as the input and produces  $N$  as the output by multiplying  $N0$  to itself.

```
proc {IncSquare N0 N}
  N1 in
  {Increment N0 N1}
  {Square N1 N}
end
```

**IncSquare** takes  $N0$  as the input and produces  $N$  as the output by using an intermediate variable  $N1$  to carry  $N0+1$  (the output of **Increment**) and passing it as input to **Square**. The pairs  $N0-N1$  and  $N1-N$  are called *accumulators*.

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

3

## Accumulators

- Assume that the state  $S$  consists of a number of components to be transformed individually:

$S = (X, Y, Z)$

- Assume  $P1$  to  $Pn$  are procedures:

```
accumulator
proc {P X0 X Y0 Y Z0 Z}
  {P1 X0 X1 Y0 Y1 Z0 Z1}
  {P2 X1 X2 Y1 Y2 Z1 Z2}
  ...
  {Pn Xn-1 X Yn-1 Y Zn-1 Z}
end
```

- The procedural syntax is easier to use if there is more than one accumulator

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

4

## Example

- Consider a variant of MergeSort with accumulator
- `proc {MergeSort1 N S0 S Xs}`
  - $N$  is an integer,
  - $S0$  is an input list to be sorted
  - $S$  is the remainder of  $S0$  after the first  $N$  elements are sorted
  - $Xs$  is the sorted first  $N$  elements of  $S0$
- The pair  $(S0, S)$  is an accumulator
- The definition is in a procedural syntax because it has two outputs  $S$  and  $Xs$

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

5

## Example (2)

```
fun {MergeSort Xs}
  {MergeSort1 (Length Xs) Xs _ Ys}
  Ys
end
```

```
proc {MergeSort1 N S0 S Xs}
  if N==0 then S = S0 Xs = nil
  elseif N == 1 then X in X|S = S0 Xs=[X]
  else %N > 1
    local S1 Xs1 Xs2 NL NR in
    NL = N div 2
    NR = N - NL
    {MergeSort1 NL S0 S1 Xs1}
    {MergeSort1 NR S1 S Xs2}
    Xs = {Merge Xs1 Xs2}
  end
end
end
```

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

6

## Multiple accumulators

- Consider a stack machine for evaluating arithmetic expressions
- Example:  $(1+4)-3$
- The machine executes the following instructions



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

7

## Multiple accumulators (2)

- Example:  $(1+4)-3$
- The arithmetic expressions are represented as trees:  
 $\text{minus}(\text{plus}(1\ 4)\ 3)$
- Write a procedure that takes arithmetic expressions represented as trees and output a list of stack machine instructions and counts the number of instructions

```
proc {ExprCode Expr Cin Cout Nin Nout}
```

- Cin: initial list of instructions
- Cout: final list of instructions
- Nin: initial count
- Nout: final count

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

8

## Multiple accumulators (3)

```

proc {ExprCode Expr C0 C N0 N}
  case Expr
  of plus(Expr1 Expr2) then C1 N1 in
    C1 = plus(C0)
    N1 = N0 + 1
    {SeqCode [Expr2 Expr1] C1 C N1 N}
  [] minus(Expr1 Expr2) then C1 N1 in
    C1 = minus(C0)
    N1 = N0 + 1
    {SeqCode [Expr2 Expr1] C1 C N1 N}
  [] I andthen {!sint I} then
    C = push(I)(C0)
    N = N0 + 1
  end
end
    
```

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

9

## Multiple accumulators (4)

```

proc {ExprCode Expr C0 C N0 N}
  case Expr
  of plus(Expr1 Expr2) then C1 N1 in
    C1 = plus(C0)
    N1 = N0 + 1
    {SeqCode [Expr2 Expr1] C1 C N1 N}
  [] minus(Expr1 Expr2) then C1 N1 in
    C1 = minus(C0)
    N1 = N0 + 1
    {SeqCode [Expr2 Expr1] C1 C N1 N}
  [] I andthen {!sint I} then
    C = push(I)(C0)
    N = N0 + 1
  end
end
    
```

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

10

```

proc {SeqCode Es C0 C N0 N}
  case Es
  of nil then C = C0 N = N0
  [] E|Er then N1 C1 in
    {ExprCode E C0 C1 N0 N1}
    {SeqCode Er C1 C N1 N}
  end
end
    
```

## Shorter version (4)

```

proc {ExprCode Expr C0 C N0 N}
  case Expr
  of plus(Expr1 Expr2) then
    {SeqCode [Expr2 Expr1] plus(C0 C N0 + 1 N)}
  [] minus(Expr1 Expr2) then
    {SeqCode [Expr2 Expr1] minus(C0 C N0 + 1 N)}
  [] I andthen {!sint I} then
    C = push(I)(C0)
    N = N0 + 1
  end
end
    
```

```

proc {SeqCode Es C0 C N0 N}
  case Es
  of nil then C = C0 N = N0
  [] E|Er then N1 C1 in
    {ExprCode E C0 C1 N0 N1}
    {SeqCode Er C1 C N1 N}
  end
end
    
```

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

11

## Functional style (4)

```

fun {ExprCode Expr t(C0 N0)}
  case Expr
  of plus(Expr1 Expr2) then
    {SeqCode [Expr2 Expr1] t(plus(C0 N0 + 1))}
  [] minus(Expr1 Expr2) then
    {SeqCode [Expr2 Expr1] t(minus(C0 N0 + 1))}
  [] I andthen {!sint I} then
    t(push(I)(C0 N0 + 1))
  end
end
    
```

```

fun {SeqCode Es T}
  case Es
  of nil then T
  [] E|Er then
    T1 = {ExprCode E T} in
    {SeqCode Er T1}
  end
end
    
```

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

12

## Difference lists (1)

- A *difference list* is a pair of lists, each might have an unbound tail, with the invariant that one can get the second list by removing zero or more elements from the first list
- $X \# X$  % Represent the empty list
- $\text{nil} \# \text{nil}$  % idem
- $[a] \# [a]$  % idem
- $(a|b|c|X) \# X$  % Represents  $[a\ b\ c]$
- $[a\ b\ c\ d] \# [d]$  % idem

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

13

## Difference lists (2)

- When the second list is unbound, an append operation with another difference list takes constant time
- ```
fun {AppendD D1 D2}
  S1 # E1 = D1
  S2 # E2 = D2
in
  E1 = S2
  S1 # E2
end
```
- ```
local X Y in {Browse {AppendD (1|2|3|X)#X (4|5|Y)#Y}}
```

 end
- Displays  $(1|2|3|4|5|Y)\#Y$

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

14

## A FIFO queue with difference lists (1)

- A *FIFO queue* is a sequence of elements with an insert and a delete operation.
  - Insert adds an element to one end and delete removes it from the other end
- Queues can be implemented with lists. If  $L$  represents the queue content, then inserting  $X$  gives  $X|L$  and deleting  $X$  gives  $\{\text{ButLast } L\ X\}$  (all elements but the last).
  - Delete is inefficient: it takes time proportional to the number of queue elements
- With difference lists we can implement a queue with **constant-time insert and delete operations**
  - The queue content is represented as  $q(N\ S\ E)$ , where  $N$  is the number of elements and  $S\#E$  is a difference list representing the elements

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

15

## A FIFO queue with difference lists (2)

```
fun {NewQueue} X in q(0 X X) end
```

```
fun {Insert Q X}
  case Q of q(N S E) then E1 in E=X|E1 q(N+1 S E1) end
end
```

```
fun {Delete Q X}
  case Q of q(N S E) then S1 in X|S1=S q(N-1 S1 E) end
end
```

```
fun {EmptyQueue} case Q of q(N S E) then N==0 end end
```

- Inserting 'b':
  - In:  $q(1\ a|T\ T)$
  - Out:  $q(2\ a|b|U\ U)$
- Deleting X:
  - In:  $q(2\ a|b|U\ U)$
  - Out:  $q(1\ b|U\ U)$  and  $X=a$
- Difference list allows operations at **both ends**
- $N$  is needed to keep track of the number of queue elements

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

16

## Flatten (revisited)

```
fun {Flatten Xs}
  case Xs
  of nil then nil
  [] X|Xr andthen {IsLeaf X} then
    X|{Flatten Xr}
  [] X|Xr andthen {Not {IsLeaf X}} then
    {Append {Flatten X} {Flatten Xr}}
  end
end
```

Flatten takes a list of elements and sub-lists and returns a list with only the elements, e.g.:

```
{Flatten [1 [2] [[3]]]} = [1 2 3]
```

Let us replace lists by difference lists and see what happens.

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

17

## Flatten with difference lists (1)

- Flatten of nil is  $X\#X$
- Flatten of  $X|Xr$  is  $Y1\#Y2$ 
  - flatten of  $X$  is  $Y1\#Y2$
  - flatten of  $Xr$  is  $Y3\#Y4$
  - equate  $Y2$  and  $Y3$
- Flatten of a leaf  $X$  is  $(X|Y)\#Y$

Here is what it looks like as text

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

18

## Flatten with difference lists (2)

```
proc {FlattenD Xs Ds}
case Xs
of nil then Y in Ds = Y#Y
[] X|Xr then Y0 Y1 Y2 in
Ds = Y0#Y2
{FlattenD X Y0#Y1}
{FlattenD Xr Y1#Y2}
[] X andthen {isLeaf X} then Y in (X|Y)#Y
end
end
fun {Flatten Xs} Y in {FlattenD Xs Y#nil} Y end
```

Here is the new program. It is much more efficient than the first version.

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

19

## Reverse (revisited)

- Here is our recursive reverse:

```
fun {Reverse Xs}
case Xs
of nil then nil
[] X|Xr then {Append (Reverse Xr) [X]}
end
end
```

- Rewrite this with difference lists:

- Reverse of nil is X#X
- Reverse of X|Xs is Y1#Y, where
  - reverse of Xs is Y1#Y2, and
  - equate Y2 and X|Y

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

20

## Reverse with difference lists (1)

- The naive version takes time proportional to the **square** of the input length
- Using difference lists in the naive version makes it **linear time**
- We use two arguments Y1 and Y instead of Y1#Y
- With a minor change we can make it **iterative** as well

```
fun {ReverseD Xs}
proc {ReverseD Xs Y1 Y}
case Xs
of nil then Y1=Y
[] X|Xr then Y2 in
{ReverseD Xr Y1 Y2}
Y2 = X|Y
end
end
R in
{ReverseD Xs R nil}
R
end
```

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

21

## Reverse with difference lists (2)

```
fun {ReverseD Xs}
proc {ReverseD Xs Y1 Y}
case Xs
of nil then Y1=Y
[] X|Xr then
{ReverseD Xr Y1 X|Y}
end
end
R in
{ReverseD Xs R nil}
R
end
```

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

22

## Difference lists: Summary

- Difference lists are a way to represent lists in the declarative model such that **one append operation can be done in constant time**
  - A function that builds a big list by concatenating together lots of little lists can usually be written efficiently with difference lists
  - The function can be written naively, using difference lists and append, and will be efficient when the append is expanded out
- Difference lists are declarative, yet have **some of the power of destructive assignment**
  - Because of the single-assignment property of dataflow variables
- Difference lists originated from **Prolog** and are used to implement, e.g., definite clause grammar rules for natural language parsing.

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

23

## Exercises

81. VRH Exercise 3.10.11 (page 232)
82. VRH Exercise 3.10.14 (page 232)
83. \*VRH Exercise 3.10.15 (page 232)
84. \*Modify the PLP11.3 “tic-tac-toe” example so that the computer strategy does not lose. Recall that the PLP code attempts to block a potential split from the opponent, but if there are two potential splits, it should instead try to win using a line that will not cause the opponent to create a double split.

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

24