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

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:

\[ S = (X, Y, Z, \ldots) \]

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:

\[ (X_{\text{in}}, X_{\text{out}}, Y_{\text{in}}, Y_{\text{out}}, Z_{\text{in}}, Z_{\text{out}}, \ldots) \]

A Trivial Example

\[
\begin{align*}
\text{proc} \{ \text{Increment} \ N0 \ N \} & \quad N = N0 + 1 \\
\text{end} \\
\text{proc} \{ \text{Square} \ N0 \ N \} & \quad N = N0 \times N0 \\
\text{end} \\
\text{proc} \{ \text{IncSquare} \ N0 \ N \} & \quad N1 \text{in} \{ \text{Increment} \ N0 \ N1 \} \\
& \quad \{ \text{Square} \ N1 \ N \} \\
\text{end}
\end{align*}
\]

Increment takes N0 as the input and produces N as the output by adding 1 to N0.
Square takes N0 as the input and produces N as the output by multiplying N0 to itself.
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.

Example

Consider a variant of MergeSort with accumulator

\[
\text{fun} \{ \text{MergeSort} \ Xs \} = \\
\{ \text{MergeSort1} \ \{ \text{Length} \ Xs \} \ Xs \_ \ Ys \} \\
\text{Ys}
\text{end}
\]

Example (2)

\[
\text{fun} \{ \text{MergeSort} \ Xs \} = \{ \text{MergeSort1} \ Xs \_ \ Ys \} \\
\text{Ys}
\text{end}
\]

Fun: MergeSort1 N S0 S Xs

- N is an integer,
- S0 is an input list to be sorted
- S is the reminder 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

Accumulators

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

\[ S = (X, Y, Z, \ldots) \]

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:

\[ (X_{\text{in}}, X_{\text{out}}, Y_{\text{in}}, Y_{\text{out}}, Z_{\text{in}}, Z_{\text{out}}, \ldots) \]
Multiple accumulators

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

Multiple accumulators (2)

- Example: \((1+4)-3\)
- The arithmetic expressions are represented as trees:
  \[
  \text{minus(plus(1 4) 3)}
  \]
- Write a procedure that takes arithmetic expressions represented as trees and output a list of stack machine instructions and counts the number of instructions
  \[
  \text{proc (ExprCode Expr Cin Cout Nin Nout)}
  \]
  - \(\text{Cin}\): initial list of instructions
  - \(\text{Cout}\): final list of instructions
  - \(\text{Nin}\): initial count
  - \(\text{Nout}\): final count

Multiple accumulators (3)

\[
\text{proc (ExprCode Expr Cin Cout Nin Nout)}
\]
\[
\text{case Expr of}
\]
\[
\begin{align*}
\text{plus(Expr1 Expr2)} & \rightarrow \text{C1 N1 in} \\
& \text{C1 = plus(C0)} \\
& \text{N1 = N0 + 1} \\
& \text{[SeqCode (Expr2 Expr1) C1 N1 N0]}
\end{align*}
\]
\[
\begin{align*}
\text{minus(Expr1 Expr2)} & \rightarrow \text{C1 N1 in} \\
& \text{C1 = minus(C0)} \\
& \text{N1 = N0 + 1} \\
& \text{[SeqCode (Expr2 Expr1) C1 N1 N0]}
\end{align*}
\]
\[
\text{I andthen (lastInt I) then}
\]
\[
\begin{align*}
\text{C = push(I)|C0} \\
& N = N0 + 1
\end{align*}
\]

Multiple accumulators (4)

\[
\text{proc (ExprCode Expr Cin Cout Nin Nout)}
\]
\[
\text{case Expr of}
\]
\[
\begin{align*}
\text{plus(Expr1 Expr2)} & \rightarrow \text{C1 N1 in} \\
& \text{C1 = plus(C0)} \\
& \text{N1 = N0 + 1} \\
& \text{[SeqCode (Expr2 Expr1) C1 C N1 N0]} \\
& \text{[SeqCode Er C1 C N1 N0]}
\end{align*}
\]
\[
\text{minus(Expr1 Expr2)} \rightarrow \text{C1 N1 in} \\
\text{C1 = minus(C0)} \\
\text{N1 = N0 + 1} \\
\text{[SeqCode (Expr2 Expr1) C1 C N1 N0]} \\
\text{[SeqCode Er C1 C N1 N0]}
\]
\[
\text{I andthen (lastInt I) then}
\]
\[
\begin{align*}
\text{C = push(I)|C0} \\
& N = N0 + 1
\end{align*}
\]

Shorter version (4)

\[
\text{proc (ExprCode Expr Cin Cout Nin Nout)}
\]
\[
\text{case Expr of}
\]
\[
\begin{align*}
\text{plus(Expr1 Expr2)} & \rightarrow \text{C1 N1 in} \\
& \text{C1 = plus(C0)} \\
& \text{N1 = N0 + 1} \\
& \text{[SeqCode (Expr2 Expr1) plus(C0 C N0 + 1)]}
\end{align*}
\]
\[
\begin{align*}
\text{minus(Expr1 Expr2)} & \rightarrow \text{C1 N1 in} \\
& \text{C1 = minus(C0)} \\
& \text{N1 = N0 + 1} \\
& \text{[SeqCode (Expr2 Expr1) minus(C0 C N0 + 1)]}
\end{align*}
\]
\[
\text{I andthen (lastInt I) then}
\]
\[
\begin{align*}
\text{C = push(I)|C0} \\
& N = N0 + 1
\end{align*}
\]

Functional style (4)

\[
\text{fun (ExprCode Expr Cin Cout Nin Nout)}
\]
\[
\text{case Expr of}
\]
\[
\begin{align*}
\text{plus(Expr1 Expr2)} & \rightarrow \text{C1 N1 in} \\
& \text{C1 = plus(C0)} \\
& \text{N1 = N0 + 1} \\
& \text{[SeqCode (Expr2 Expr1) plus(C0 N0 + 1)]}
\end{align*}
\]
\[
\begin{align*}
\text{minus(Expr1 Expr2)} & \rightarrow \text{C1 N1 in} \\
& \text{C1 = minus(C0)} \\
& \text{N1 = N0 + 1} \\
& \text{[SeqCode (Expr2 Expr1) minus(C0 N0 + 1)]}
\end{align*}
\]
\[
\text{I andthen (lastInt I) then}
\]
\[
\begin{align*}
\text{C = push(I)|C0} \\
& N = N0 + 1
\end{align*}
\]

C. Varela; Adapted w/permission from S. Haridi and P. Van Roy
**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
- $[\text{a}] \# [\text{a}]$ % idem
- $(\text{a b c X}) \# X$ % Represents $[\text{a b c}]$
- $[\text{a b c d}] \# [\text{d}]$ % idem

**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 $[1|2|3|4|5|Y]#Y$

**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.

**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
  
  $X|E1 q(N+1 S E1)$
  
  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

**Flatten (revisited)**

- fun {Flatten Xs}
  
  case $Xs$ of
  
  nil then nil
  
  $[\text{X}Xr]$ andthen {IsLeaf X} then
  
  $X|\{\text{Flatten Xr}\}$
  
  $[\text{X}Xr]$ andthen {Not {IsLeaf X}} then
  
  $\{\text{Append} \{\text{Flatten X}\} \{\text{Flatten Xr}\}\}$
  
  end
  
  end

- Flatten takes a list of elements and sub-lists and returns a list with only the elements, e.g.,
  
  $\{\text{Flatten \{1 \{2\}\{3\}\}}\} = \{1 \ 2 \ 3\}$

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

**Flatten with difference lists (1)**

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

---

Note: The text contains LaTeX code for mathematical expressions and some code snippets for functional programming, which are not fully rendered in the text representation.
Flatten with difference lists (2)

```prolog
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 X Y1#Y2}
    [] X andthen [IsLeaf X] then Y in (X|Y)#Y end
  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.

Reverse (revisited)

- Here is our recursive reverse:

```prolog
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

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

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

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

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.

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.