Declarative Programming Techniques

Accumulators (CTM 3.4.3)

Carlos Varela

RPI

Adapted with permission from:

Seif Haridi

KTH

Peter Van Roy

UCL

September 9, 2014

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 predicate 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

  $(X_{in}, X_{out}, Y_{in}, Y_{out}, Z_{in}, Z_{out}, \ldots)$

A Trivial Example in Prolog

```
increment(N0,N) :-
    N is N0 + 1.

square(N0,N) :-
    N is N0 * N0.

inc_square(N0,N) :-
    increment(N0,N1),
    square(N1,N).
```

`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.

`inc_square` 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*.

A Trivial Example in Oz

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

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

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

`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*.

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 in Oz

$$
\begin{array}{l}
\text{proc } \{P\ \underbrace{X_0\ X}_{\text{accumulator}}\ Y_0\ Y\ Z_0\ Z\} \\
\qquad \vdots \\
\qquad \{P1\ X_0\ X_1\ Y_0\ Y_1\ Z_0\ Z_1\} \\
\qquad \{P2\ X_1\ X_2\ Y_1\ Y_2\ Z_1\ Z_2\} \\
\qquad \vdots \\
\qquad \{Pn\ X_{n-1}\ X\ Y_{n-1}\ Y\ Z_{n-1}\ Z\} \\
\text{end}
\end{array}
$$

The same concept applies to predicates in Prolog

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

MergeSort 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 in Oz because it has two outputs S and Xs

Example (2)

```
fun {MergeSort Xs}
   Ys in
   {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
```

MergeSort Example in Prolog

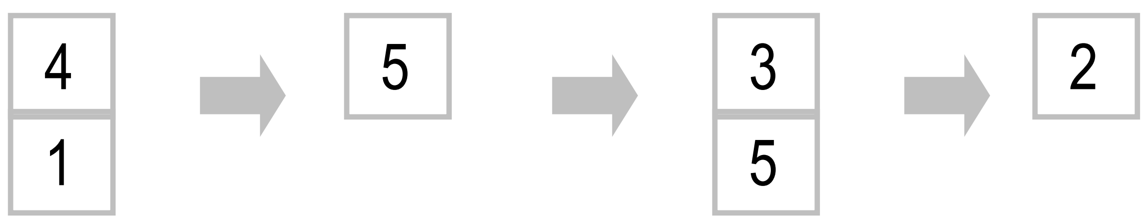
```
mergesort(Xs,Ys) :-
  length(Xs,N),
  mergesort1(N,Xs,_,Ys).
```

```
mergesort1(0,S,S,[]) :- !.
mergesort1(1,[X|S],S,[X]) :- !.
mergesort1(N,S0,S,Xs) :-
    NL is N // 2,
    NR is N - NL,
    mergesort1(NL,S0,S1,Xs1),
    mergesort1(NR,S1,S,Xs2),
    merge(Xs1,Xs2,Xs).
```

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:

    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

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

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

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 {IsInt I} then
    C = push(I)|C0
    N = N0 + 1
  end
end
```

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 {IsInt 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
```

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 {IsInt 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
```

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 {IsInt 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
```

Exercises

15. Understand how Oz supports logic programming by comparing it to Prolog (read CTM Sect. 9.7; pp.660-671)
    a) Download Mozart (Oz run-time system) and install it in your laptop.
    b) Rewrite your Prolog family program in Oz.
    c) Rewrite the Prolog list append predicate in Oz.
    d) Rewrite the Oz multiple accumulators example in Prolog.