Declarative Programming Techniques

Accumulators (CTM 3.4.3)
Difference Lists (CTM 3.4.4)

Carlos Varela

Rensselaer Polytechnic Institute

Adapted with permission from:

Seif Haridi

KTH

Peter Van Roy

UCL

November 26, 2024

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

  $S = (X,Y,Z,...)$

- 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 been evaluated.
- *S* is represented as

  $(X_{in}, X_{out}, Y_{in}, Y_{out}, Z_{in}, Z_{out},...)$

A Trivial Example

```
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 by 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-N, 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 p_1 to p_n are predicates.

accumulator

$p(X_0,X, Y_0,Y, Z_0,Z)$:-
  $p1(X_0,X_1, Y_0,Y_1, Z_0,Z_1)$,
  $p2(X_1,X_2, Y_1,Y_2, Z_1,Z_2)$,
  :
  $pn(X_{n-1},X, Y_{n-1},Y, Z_{n-1},Z)$.

MergeSort Example

- Consider a variant of MergeSort with accumulator

  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 has two outputs: S and Xs

MergeSort Example

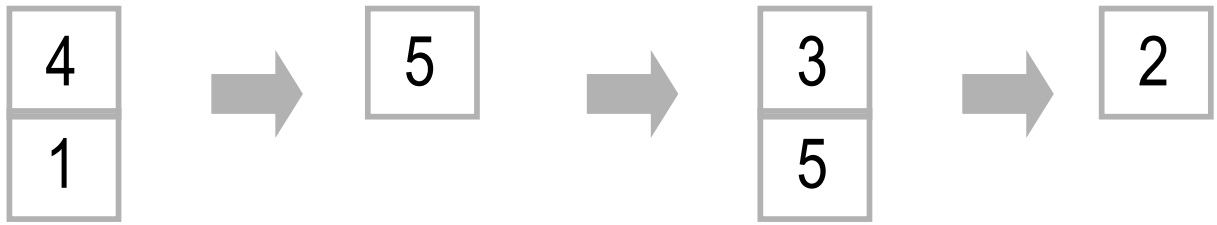
```
mergesort(Xs,Ys) :-
   length(Xs,N),
   mergesort1(N,Xs,_,Ys).
```

```
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 predicate that takes arithmetic expressions represented as trees and outputs a list of stack machine instructions and counts the number of instructions

  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)

```
exprCode(plus(Expr1,Expr2),C0,C,N0,N) :-
    !,
    N1 is N0 + 1,
    seqCode([Expr2,Expr1],[plus|C0],C,N1,N).

exprCode(minus(Expr1,Expr2),C0,C,N0,N) :-
    !,
    N1 is N0 + 1,
    seqCode([Expr2,Expr1],[minus|C0],C,N1,N).

exprCode(I,C0,[push(I)|C0],N0,N) :-
    integer(I),
    N is N0 + 1.
```

Multiple accumulators (4)

```
exprCode(plus(Expr1,Expr2),C0,C,N0,N) :-
    !,
    N1 is N0 + 1,
    seqCode([Expr2,Expr1],[plus|C0],C,N1,N).

exprCode(minus(Expr1,Expr2),C0,C,N0,N) :-
    !,
    N1 is N0 + 1,
    seqCode([Expr2,Expr1],[minus|C0],C,N1,N).

exprCode(I,C0,[push(I)|C0],N0,N) :-
    integer(I),
    N is N0 + 1.
```

```
seqCode([],C,C,N,N) :- !.

seqCode([E|Er], C0,C, N0,N) :-
    exprCode(E,C0,C1,N0,N1),
    seqCode(Er,C1,C,N1,N).
```

Difference lists

- 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
- [] , [] % idem
- [a] , [a] % idem
- [a,b,c|X] , X % Represents [a,b,c]
- [a,b,c,d] , [d] % idem
- [a,b,c,d|Y] , [d|Y] % idem
- [a,b,c,d|Y] , Y % Represents [a,b,c,d]

Difference lists (2)

- When the second list is unbound, an append operation with another difference list takes constant time

```
append_dl(S1,E1, S2,E2, S1,E2)  :-  E1 = S2.
```

- ?- append_dl([1,2,3|X],X, [4,5|Y],Y, S,E).

Displays

X = [4, 5|E],

Y = E,

S = [1, 2, 3, 4, 5|E].

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 the end and delete removes it from the beginning
- Queues can be implemented with lists. If L represents the queue content, then deleting X can remove the head of the list matching [X|T] but inserting X requires traversing the list with append(L,[X]) (insert element at the end).
  - Insert 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)

```
newQueue(q(0,X,X)).

insert(q(N0,S,E0),X,q(N,S,E)) :- N is N0 + 1, E0 = [X|E].

delete(q(N0,S0,E),X,q(N,S,E)) :- N is N0 - 1, [X|S] = S0.

emptyQueue(q(0,_S,_E)).
```

- 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 used to keep track of the number of queue elements

A FIFO queue with difference lists (3)

```
newQueue2(q(X,X)).

insert2(q(S,E0),X,q(S,E)) :- E0 = [X|E].

delete2(q(S0,E),X,q(S,E)) :- [X|S] = S0.

emptyQueue2(q(S,E)) :- S == E.
```

- Inserting ‘b’:
  - In: q(a|T],T)
  - Out: q([a,b|U],U)
- Deleting X:
  - In: q([a,b|U],U)
  - Out: q([b|U],U) and X=a
- Difference list allows operations at both ends
- Co-binding test is used to test for an empty queue.

Flatten

```
isLeaf(N) :- not(is_list(N)).

flatten1([],[]).
flatten1([X|Xr], [X|Yr]) :- isLeaf(X), !,
    flatten1(Xr,Yr).
flatten1([X|Xr], Ys) :- flatten1(X,Y),
    flatten1(Xr, Yr),
    append(Y,Yr,Ys).
```

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

?- flatten1([1,[2],[[3]]],L)
L = [1 2 3].

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

Flatten with difference lists (1)

Here is what it looks like as text:

- Flatten of [] is X,X
- Flatten of a leaf [X|Xr] is [X|Y1],Y
  - flatten of Xr is Y1,Y
- Flatten of [X|Xr] is Y0,Y where
  - flatten of X is Y0#Y1
  - flatten of Xr is Y2#Y
  - equate Y1 and Y2

Flatten with difference lists (2)

```
flattenD([],Y,Y).
flattenD([X|Xr],[X|Y1],Y) :-
    isLeaf(X), !, flattenD(Xr,Y1,Y).
flattenD([X|Xr],Y0,Y) :-
    flattenD(X,Y0,Y1), flattenD(Xr,Y1,Y).

flatten2(Xs,Ys) :- flattenD(Xs,Ys,[]).
```

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

Reverse

- Consider the recursive reverse:

```
reverse1([],[]).

reverse1([X|Xr],Ys) :-
    reverse1(Xr,Yr),
    append(Yr,[X],Ys).
```

- Rewriting it with difference lists:
  - Reverse of [] is X,X
  - Reverse of [X|Xr] is S,E, where
    - reverse of Xr is S,E0, and
    - equate E0 and [X|E]

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
- In reverseD, we use three arguments: Xs,S,E instead of two for reverse: Xs,Ys.
- With a minor change we can make it tail-recursive as well

```
reverseD([],Y,Y).

reverseD([X|Xr],S,E) :-
    reverseD(Xr,S,E0),
    E0 = [X|E].

reverse2(Xs,Ys) :-
    reverseD(Xs,Ys,[]).
```

Reverse with difference lists (3)

```
reverseD2([],Y,Y).

reverseD2([X|Xr],S,E) :-
      reverseD2(Xr,S,[X|E]).

reverse3(Xs,Ys) :-
      reverseD2(Xs,Ys,[]).
```

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 in Prolog and are used to implement definite clause grammar (DCG) rules for natural language parsing.

Exercises

85. Draw the search trees for Prolog queries:
    - `append([1,2],[3],L).`
    - `append(X,Y,[1,2,3]).`
    - `append_dl([1,2|X],X,[3|Y],Y,S,E).`
86. CTM Exercise 3.10.11 (page 232). Do it in Prolog.
87. CTM Exercise 3.10.14 (page 232). Do it in Prolog.
88. CTM Exercise 3.10.15 (page 232). Do it in Prolog.