Declarative Programming Techniques
Difference Lists (CTM 3.4.4)

Carlos Varela
RPI
Adapted with permission from:
Seif Haridi
KTH
Peter Van Roy
UCL

September 12, 2014
Difference lists in Oz

• 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
Difference lists in Prolog

- 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.

- \[ X, X \] \quad \% Represent the empty list
- \[ [], [] \] \quad \% idem
- \[ [a], [a] \] \quad \% idem
- \[ [a,b,c|X], X \] \quad \% Represents \([a,b,c]\)
- \[ [a,b,c,d], [d] \] \quad \% idem
Difference lists in Oz (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
Difference lists in Prolog (2)

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

\[
\text{append} \_ \text{dl}(S1,E1, S2,E2, S1,E2) \ :- \ E1 = S2.
\]

- \texttt{?- append} \_ \texttt{dl}([1,2,3|X],X, [4,5|Y], Y, S,E).

Displays

\[
\begin{align*}
X & = [4, 5|_G193] \\
Y & = _G193 \\
S & = [1, 2, 3, 4, 5|_G193] \\
E & = _G193;
\end{align*}
\]
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 \{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)

- 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

```hlkotlin
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 Q}
    case Q of q(N S E) then N==0 end end
```
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.
Flatten with difference lists (1)

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

Here is what it looks like as text
Flatten with difference lists (2)

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

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

• Here is our recursive reverse:

```plaintext
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
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.

```plaintext
fun {ReverseD Xs}
proc {ReverseD Xs Y1 Y1}
  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
```
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
  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

16. 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).`

17. CTM Exercise 3.10.11 (page 232)
18. CTM Exercise 3.10.14 (page 232)
19. CTM Exercise 3.10.15 (page 232)