Prolog lecture 5

- Data structures
- Difference lists
- Appendless append

Appending two Lists

Predicate definition is elegantly simple:

```
append([],L,L).
append([X|T],L,[X|R]) :- append(T,L,R).
```

Run-time performance is not good though

- Procedural languages would not scan a list to append

Want to modify the end of the list directly

- Prolog can achieve this

```
append([],L,L).
append([X|T],L,[X|R]) :- append(T,L,R).
```

append([1,2],[3,4],A).

$\downarrow$

append([X|T],L,[X|R]):-append(T,L,R).

append([1|[2]],[3,4],[1|V_1]):-append([2],[3,4],V_1). A=[1|V_1]

$\downarrow$

append([X|T],L,[X|R]):-append(T,L,R).

append([2|[]]],[3,4],[2|V_2]):-append([],[3,4],V_2). V_1=[2|V_2]

$\downarrow$

append([],L,L).

append([],[3,4],[3,4]). V_2=[3,4]

Difference Lists (p185)

Instead of storing one list, store two

- Represent our original list as the difference between these other two lists

We might represent the "normal" list [1,2,3] as

- [1,2,3,4,5]-[4,5] or
- [1,2,3,acr]-[acr] or
- [1,2,3|X]-X

It is the last form here that is key!

Difference List Append

Append one list to another...

```
1 :: ( 2 :: ( 3 :: [] ) )

4 :: ( 5 :: ( 6 :: [] ) )
```

Difference List Append

... in a single list-linking step

```
1 :: ( 2 :: ( 3 :: [] ) )

4 :: ( 5 :: ( 6 :: [] ) )
```

Difference List Append

`1 :: ( 2 :: ( 3 :: A ) )`

$$\underbrace{\texttt{4 :: (5 :: (6 :: B))}}_{A}$$

Although A "is" the second list, to "be" the second list just requires being a label for the beginning of that list.

Prolog syntax for the first list is [1,2,3|A]

Difference List Append

A potential representation of difference list append:

dapp(L1,V1,L2,V2,L3,V3).

First list
e.g. [1,2,3|V1]

The variable at the end of the first list

Ideally the two parts (L1 and V1) of the difference list would be kept together though...

Difference List Append

By convention we write our difference list pair as

A-B

But we could also write:

differenceList(A,B)

A+B

A*B, etc

dapp(L1-V1,L2-V2,L3-V3)

- Append difference list L2-V2 to L1-V1 and unify the result with L3-V3.

Difference List Append (implementation)

L1 $l1_0::l1_1:: \cdots ::l1_n::V1$

L2 $l2_0::l2_1:: \cdots ::l2_n::V2$

L3 $l1_0:: \cdots ::l1_n::l2_0:: \cdots ::l2_n::V2$

```
dapp(L1-V1,L2-V2,L3-V3) :- V1=L2,
                           L3=L1,
                           V3=V2.
```

Difference List Append (implementation)

L1 $\boxed{l1_0::l1_1::\ \cdots\ ::l1_n}::V1$

L2 $l2_0::l2_1::\ \cdots\ ::l2_n::V2$

L3 $l1_0::\ \cdot$

This is the value of the list we want to represent and so our difference list has to be
$[l1_0,l1_1,l1_2...|V1]\text{-}V1$

```
dapp(L1
                    V3=V2.
```

Difference List Append (implementation)

L1 $l1_0::l1_1:: \cdots ::l1_n::V1$

L2 $l2_0::l2_1:: \cdots ::l2_n::V2$

L3 $l1_0:: \cdots ::l1_n::l2_0:: \cdots ::l2_n::V2$

```
dapp(L1-V1,L2-V2,L3-V3) :- V1=L2,
                           L3=L1,
                           V3=V2.
```

Difference List Append (implementation)

```
dapp(L1-V1,L2-V2,L3-V3) :- V1=L2,
                           L3=L1,
                           V3=V2.
```

We know that V1 and L2 must be the same:

- replace all instances of V1 and L2 with new variable B
- (we can remove the B=B of course)

```
dapp(L1-B,B-V2,L3-V3) :- B=B,
                         L3=L1,
                         V3=V2.
```

Difference List Append (implementation)

So we have:

```
dapp(L1-B,B-V2,L3-V3) :- L3=L1,
                         V3=V2.
```

But we know that L3 and L1 must be the same

- Replace them with a new variable A:

```
dapp(A-B,B-V2,A-V3) :- A=A,
                       V3=V2.
```

Difference List Append (final implementation)

Now we have:

```
dapp(A-B,B-V2,A-V3) :- V3=V2.
```

But we know that V3 and V2 must be the same

– Substituting a new variable C:

```
dapp(A-B,B-C,A-C) :- C=C.
```

But that simplifies to the following final answer

– Gradual substitution like this is a useful technique

```
dapp(A-B,B-C,A-C).
```

Representing empty Difference Lists

An empty difference list is an empty list with a variable at its end ready for later binding.

- Let us call this variable `A`
- We've seen lists like `[1,2|A]`

If you understand the `[···|L]` syntax you will appreciate that removing `1,2` leaves (simply):

`A`

We write this in the conventional notation as:

`A-A`

Another Difference List Example

Define a procedure `rotate(X,Y)` where both X and Y are represented by difference lists, and Y is formed by rotating X to the left by one element.

[14 marks]

1996-6-7

(This is the second example in your handout)

Determine an answer first that does not use Difference Lists

Take the first element off the first list (H) and append it after the tail (i.e. at the end) in the solution (R)

```
rotate([H|T],R) :- append(T,[H],R).
```

Rewrite with Difference Lists

Allocate “tail variables” to our original lists

- Give list [H|T] tail variable T1
- Give list R tail variable S

```
rotate([H|T],R) :- append(T,[H],R).
```

becomes:

```
rotate([H|T]-T1,R-S) :-
                dapp(T-T1,[H|L]-L,R-S).
```

Why is this term not `[H|T1]-T1` ?

Rename variables to incorporate difference list append

Recall: difference list append just shuffles vars

```
rotate([H|T]-T1,R-S) :-
                dapp(T-T1,[H|L]-L,R-S).
```

Rename T1 to be [H|L]: unify with B in dapp/3

```
rotate([H|T]-[H|L],R-S) :-
            dapp(T-[H|L],[H|L]-L,R-S).
```

```
dapp(A-B,B-C,A-C).
```

Rename variables to incorporate difference list append

From the previous slide:

```
rotate([H|T]-[H|L],R-S) :-
          dapp(T-[H|L],[H|L]-L,R-S).
```

Rename R to be T: thus unifying with A in dapp/3

```
rotate([H|T]-[H|L],T-S) :-
          dapp(T-[H|L],[H|L]-L,T-S).
```

```
dapp(A-B,B-C,A-C).
```

Rename variables to incorporate difference list append

From the previous slide:

```
rotate([H|T]-[H|L],T-S) :-
            dapp(T-[H|L],[H|L]-L,T-S).
```

Rename S to be L: thus unifying with C in dapp/3

```
rotate([H|T]-[H|L],T-L) :-
            dapp(T-[H|L],[H|L]-L,T-L).
```

dapp call is now redundant and can be removed!

```
dapp(A-B,B-C,A-C).
```

Final Answer

```
rotate([H|T]-[H|A],T-A).
```

Beautifully concise... but also somewhat opaque!

It is recommended that you comment any line of Prolog like this really, really thoroughly!

Converting to difference lists

```
double([],[]).
double([H|T],[R|S]) :-
    R is H*2,
    double(T,S).
```

... add in the tail variables ...

```
double(A-A,B-B).
double([H|T]-T1,[R|S]-S1) :-
    R is H*2,
    double(T-T1,S-S1).
```

Question

What does `double([1,2,3|T]-T,R)` produce?

a) yes, R = [2,4,6|X]-X
b) no
c) yes, R = X-X
d) an exception

Towers of Hanoi Revisited

- Move n rings from Src to Dest
  - move n-1 rings from Src to Aux
  - move the nth ring from Src to Dest
  - move n-1 rings from Aux to Dest
- Base case: move 0 rings from Src to Dest

End

- Next lecture: solving Sudoku,
- constraint logic programming
- and where to go next...