15-819K: Logic Programming

Lecture 11

Difference Lists

Frank Pfenning

October 3, 2006

In this lecture we look at programming techniques that are specific to logic programming, or at least significantly more easily expressed and reasoned about in logic programming than other paradigms. The first example is *difference lists*, which we use for a queue data structure, list reversal, an improvement of our earlier quicksort implementation, and a breadth-first logic programming engine that can be seen as the core of a theorem prover. We also introduce a program for peg solitaire as a prototype for state exploration. This will lead us towards considering imperative logic programming.

11.1 Functional Queues

We would like to implement a queue with operations to enqueue, dequeue, and test a queue for being empty. For illustration purposes we use a list of instructions `enq(`x`)` and `deq(`x`)`. Starting from an empty queue, we execute the instructions in the order given in the list. When the instruction list is empty we verify that the queue is also empty. Later we will use queues to implement a breadth-first logic programming interpreter.

First, a naive, and very inefficient implementation, where a queue is simply a list.

```
queue0(Is) :- q0(Is, []).

q0([enq(X)|Is], Q) :- append(Q, [X], Q2), q0(Is, Q2).
q0([deq(X)|Is], [X|Q]) :- q0(Is, Q).
q0([], []).
```

This is inefficient because of the repeaated calls to `append` which copy the queue.

In a more efficient functional implementation we instead maintain two lists, one the front of the list and one the back. We enqueue items on the back and dequeue them from the front. When the front is empty, we *reverse* the back and make it the new front.

```
queue1(Is) :- q1(Is, [], []).

q1([enq(X)|Is], F, B) :- q1(Is, F, [X|B]).
q1([deq(X)|Is], [X|F], B) :- q1(Is, F, B).
q1([deq(X)|Is], [], B) :- reverse(B,[X|F]), q1(Is,F,[]).
q1([], [], []).
```

Depending on the access patterns for queues, this can much more efficient since the first since the cost of the list reversal can be amortized over the enqueuing and dequeuing operations.

11.2 Queues as Difference Lists

The idea behind this implementation is that the a queue with elements $x_1, \ldots, x_n$ is represented as a pair `[`x_1`, ..., `x_n` | B] \ B`, where `B` is a logic variable. Here `\` is simply a constructor written in infix form to suggest list difference because the actual queue of elements for `F \ B` is the list `F` minus the tail `B`.

One may think of the variable `B` as a pointer to the end of the list, providing a means to add an element at the end in constant time (instead of calling `append` as in the very first implementation). Here is a first implementation using this idea:

```
queue(Is) :- q(Is, B\B).

q([enq(X)|Is], F\[X|B]) :- q(Is, F\B).
q([deq(X)|Is], [X|F]\B) :- q(Is, F\B).
q([], []\[]).
```

We consider is line by line, in each case considering the invariant:

> A queue $x_1, \ldots, x_n$ is represented by `[`x_1`, ..., `x_n` | B] \ B` for a logic variable `B`.

In the first clause

```
queue(Is) :- q(Is, B\B).
```

we see that the empty queue is represented as `B\B` for a logic variable `B`, which is an instance of the invariant for $n = 0$.

The second clause

```
q([enq(X)|Is], F\[X|B]) :- q(Is, F\B).
```

is trickier. A goal matching the head of this clause will have the form

?- q([enq(x_{n+1})|l], [x_1, ..., x_n | B0] \ B0).

for a term x_{n+1}, list l, terms $x_1, \ldots, x_n$ and variable `B0`. Unification will instantiate

X = x_{n+1}
Is = l
F = [x_1, ..., x_n, x_{n+1} | B]
B0 = [x_{n+1} | B]

where `B` is a fresh logic variable. Now the recursive call is

?- q(l, [x_1, ..., x_n, x_{n+1} | B1] \ B1).

satisfying our representation invariant.

The third clause

```
q([deq(X)|Is], [X|F]\B) :- q(Is, F\B).
```

looks straightforward, since we are just working on the front of the queue, removing its first element. However, there is a tricky issue when the queue is empty. In that case it has the form `B0 \ B0` for some *logic variable* `B0`, so it can actually unify with `[X|F]`. In that case, `B0 = [X|F]`, so the recursive call will be on `q(`l`, F \ [X|F])` which not only violates our invariant, but also unexpectedly allows us to remove an element from the empty queue!

The invariant will right itself once we enqueue another element that matches `X`. In other words, we have constructed a "negative" queue, borrowing against future elements that have not yet arrived. If this behavior is undesirable, it can be fixed in two ways: we can either add a counter as a third argument that tracks the number of elements in the queue, and then verify that the counter is positive before dequeuing and element. Or we can check if the queue is empty before dequeuing and fail explicitly in that case.

Let us consider the last clause.

```
q([], []\[]).
```

This checks that the queue is empty by unifying it with `[]\[]`. From the invariant we can see that it succeeds exactly if the queue is empty (neither positive nor negative, if borrowing is allowed).

When we started with the empty queue, we used the phrase `B\B` for a logic variable `B` to represent the empty queue. Logically, this is equivalent to checking unifiability with `[]\[]`, but operationally this does not work because of the lack of occurs-check in Prolog. A non-empty queue such as `[`x_1`|B0] \ B0` will incorrectly "unify" with `B1\B1` with `B1` being instantiated to a circular term `B1 = [`x_1`|B1]`.

To complete this example, we show the version that prevents negative queues by testing if the front is unifiable with `[]` before proceeding with a dequeue operation.

```
queue(Is) :- q(Is, B\B).

q([enq(X)|Is], F\[X|B]) :- q(Is, F\B).
q([deq(X)|Is], F\B) :-
        F = [] -> fail ; F = [X|F1], q(Is, F1\B).
q([], []\[]).
```

11.3 Other Uses of Difference Lists

In the queue example, it was important that the tail of the list is always a logic variable. There are other uses of difference list where this is not required. As a simple example consider `reverse`. In its naive formulation it overuses `append`, as in the naive formulation of queues.

```
naive_reverse([X|Xs], Zs) :-
        naive_reverse(Xs, Ys),
        append(Ys, [X], Zs).
naive_reverse([], []).
```

To make this more efficient, we use a difference list as the second argument.

```
reverse(Xs, Ys) :- rev(Xs, Ys\[]).

rev([X|Xs], Ys\Zs) :- rev(Xs, Ys\[X|Zs]).
rev([], Ys\Ys).
```

This time, the front of the difference list is a logic variable, to be filled in when the input list is empty.

Even though this program is certainly correctly interpreted using list difference, the use here corresponds straightforwardly to the idea of *accumulators* in functional programming: In `rev(Xs, Ys\Zs)`, `Zs` accumulates the reverse list and eventually returns it in `Ys`.

Seasoned Prolog hackers will often break up an argument which is a difference list into two top-level arguments for efficiency reasons. So the `reverse` code above might actually look like

```
reverse(Xs, Ys) :- rev(Xs, Ys, []).

rev([X|Xs], Ys, Zs) :- rev(Xs, Ys, [X|Zs]).
rev([], Ys, Ys).
```

where the connection to difference lists is harder to recognize.

Another useful example of difference lists is in quicksort from Lecture 2, omitting here the code for `partition/4`. We construct two lists, `Ys1` and `Ys2` and append them, copying `Ys1` again.

```
quicksort([], []).
quicksort([X0|Xs], Ys) :-
        partition(Xs, X0, Ls, Gs),
        quicksort(Ls, Ys1),
        quicksort(Gs, Ys2),
        append(Ys1, [X0|Ys2], Ys).
```

Instead, we can use a difference list.

```
quicksort(Xs, Ys) :-
        qsort(Xs, Ys\[]).

qsort([], Ys\Ys).
qsort([X0|Xs], Ys\Zs) :-
        partition(Xs, X0, Ls, Gs),
        qsort(Gs, Ys2\Zs),
        qsort(Ls, Ys\[X0|Ys2]).
```

In this instance of difference lists, it may be helpful to think of

`qsort(`Xs`,` $Ys\backslash Zs$`)`

as adding the sorted version of Xs to the front of Zs to obtain Ys. Then, indeed, the result of subtracting Zs from Ys is the sorted version of Xs. In order to see this most directly, we have swapped the two recursive calls to `qsort` so that the tail of the difference list is always ground on invocation.

11.4 A Breadth-First Logic Programming Interpreter

With the ideas of queues outlined above, we can easily construct a breadth-first interpreter for logic programs. Breadth-first search consumes a lot of space, and it is very difficult for the programmer to obtain a good model of program efficiency, so this is best thought of as a naive, first attempt at a theorem prover that can find proofs even where the interpreter would loop.

The code can be found on the course website.[1] It is obtained in a pretty simple way from the depth-first interpreter, by replacing the failure continuation F by a pair $F_1 \backslash F_2$, where F_2 is always a logic variable that occurs at the tail of F_1. While F_1 is not literally a list, it should be easy to see what this means.

We show here only four interesting clauses. First three that are directly concerned with the failure continuation.

```
% prove(G, Gamma, S, F1\F2, N, J)
% Gamma |- G / S / FQ, N is next free variable
% J = success or failure
...
prove(bot, _, _, bot\bot, _, J) :- !, J = failure.
prove(bot, Gamma, _, or(and(G2,S),F1)\F2, N, J) :-
        prove(G2, Gamma, S, F1\F2, N, J).
prove(or(G1,G2), Gamma, S, F1\or(and(G2,S),F2), N, J) :-
        prove(G1, Gamma, S, F1\F2, N, J).
```

The first clause here needs to commit so that the second clause cannot borrow against the future. It should match only if there is an element in the queue.

In order to make this prover complete, we need to cede control immediately after an atomic predicate is invoked. Otherwise predicates such as

```
diverge :- diverge ; true.
```

would still not terminate since alternatives, even though queued rather than stacked, would never be considered. The code for this case looks like

[1] `http://www.cs.cmu.edu/~fp/courses/lp/code/11-diff/meta.pl`

```
prove(app(Pred, Ts), Gamma, S, FQ, N, J) :-
        ...
        prove(or(bot,GTheta), Gamma, S, FQ, N, J).
```

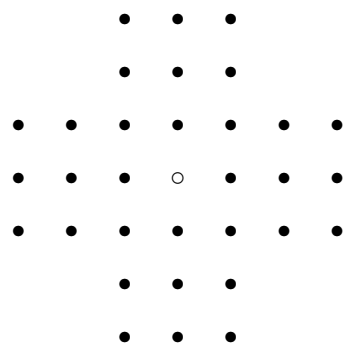
where `GTheta` will be the body of the definition of `Pred`, with arguments `Ts` substituted for the argument variables. We assume here that the program is already in residuated form, as is necessary for this semantics to be accurate.

In the next step we will queue up `GTheta` on the failure continuation and then fail while trying to prove `bot`. Another alternative, suspended earlier, will then be removed from the front of the queue and its proof attempted.[2]

11.5 State Exploration

We now switch to a different category of programs, broadly categorized as exploring state. Game playing programs are in this class, as are puzzle solving programs. One feature of logic programming we hope to exploit is the backtracking nature of the operational semantics.

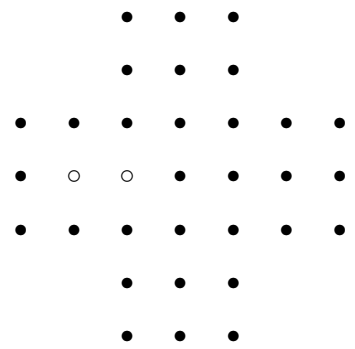
The example we use is *peg solitaire*. We are given a board of the form



where a solid circle • is a hole filled with a peg, while a ○ hollow circle represents an empty hole. In each move, a peg can jump over an adjacent one (right, down, left, or up), if the hole behind is empty. The peg that is jumped over is removed from the board. For example, in the initial position shown above there are four possible moves, all ending up in the center. If

[2] I have no proof that this really is complete—I would be interested in thoughts on the issue.

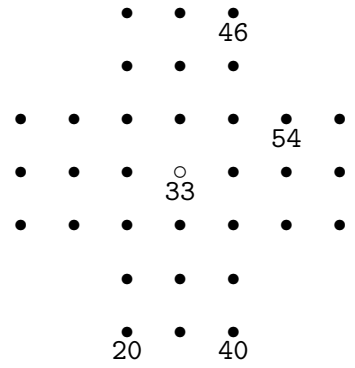
we take the possible jump to the right, we would be in the position



The objective is continue jumps until only one peg is left over.

This puzzle has been extensively analyzed (see, for example, the Wikipedia article on the subject). Our logic programming implementation will to inefficient to solve the problem by brute force, but it is nonetheless an illustrative example.[3]

We introduce a unique name for every place on the board by using a integer coordinate address and concatenating the two digits starting with `00` at the lower-left hand corner which is unoccupied, proceeding to `66` in the upper right-hand corner. Some place names are drawn in the following diagram.



Now the current state of the board during the search for a solution is represented by a list containing `peg(`ij`)` if there is a peg at location ij and

[3]This program was written by committee in real-time during lecture. Another somewhat more efficient version can be found with code that accompanies the lecture at `http://www.cs.cmu.edu/~fp/courses/lp/code/11-diff/`.

`hole(`ij`)` if the location ij is an empty hole. We have a prediate `init(`S_0`)` which holds for the initial state S_0.

```
init([
                peg(20),peg(30),peg(40),
                peg(21),peg(31),peg(41),
peg(02),peg(12),peg(22),peg(32),peg(42),peg(52),peg(62),
peg(03),peg(13),peg(23),hole(33),peg(43),peg(53),peg(63),
peg(04),peg(14),peg(24),peg(34),peg(44),peg(54),peg(64),
                peg(25),peg(35),peg(45),
                peg(26),peg(36),peg(46)
]).
```

We also have a predicate `between/3` which holds between three places A, B, and C whenever there is a possible jump to the right or up. This means that if `between(`C`,`B`,`A`)` is true then a jump left or up from A to C is possible. We show a few cases in the definition of `between`.

```
between(20,30,40).
between(20,21,22).
between(30,31,32).
between(40,41,42).
...
```

There are 38 such clauses altogether.

Next we have a predicate to flip a peg to a hole in a given state returning the new state.

```
swap([peg(A)|State], peg(A), [hole(A)|State]).
swap([hole(A)|State], hole(A), [peg(A)|State]).
swap([Place|State1], Place0, [Place|State2]) :-
        swap(State1, Place0, State2).
```

This fails if the requested place is not a peg or hole, respectively.

In order to make a single move, we find all candidate triples A, B, or C using the `between` relation and then swap the two pegs and hole to be two holes and a peg.

```
move1(State1,State4) :-
        ( between(A,B,C) ; between(C,B,A) ),
        swap(State1, peg(A), State2),
        swap(State2, peg(B), State3),
        swap(State3, hole(C), State4).
```

To see if we can make n moves from a given state we make one move and then see if we can make $n - 1$ moves from the resulting state. If this fails, we backtrack, trying another move.

```
moves(0, _).
moves(N, State1) :-
        N > 0,
        move1(State1,State2),
        N1 is N-1,
        moves(N1, State2).
```

Finally, to solve the puzzle we have to make n moves from the initial state. To have a full solution, we would need $n = 31$, since we start with 32 pegs so making 31 moves will win.

```
solve(N) :-
        init(State0),
        moves(N, State0).
```

Since in practice we cannot solve the puzzle this way, it is interesting to see how many sequences of moves of length n are possible from the initial state. For example, there are 4 possible sequences of a single move and, 12 possible sequences of two moves, and 221072 sequences of seven moves.

There we encounter a difficulty, namely that we cannot maintain any information about the number of solutions upon backtracking in pure Prolog (even though as you have seen in a homework assignments, it is easy to count the number of solutions in the meta-interpreter).

This inability to preserve information is fundamental, so Prolog implementations offer several ways to circumvent it. One class of solutions is represented by `findall` and related predicates which can collect the solutions to a query in a list. A second possibility is to use `assert` and `retract` to change the program destructively while it is running—a decidedly non-logical solution. Final, modern Prolog implementations offer global variables that can be assigned to and incremented in a way that survives backtracking.

We briefly show the third solution in GNU Prolog. A global variable is addressed by an atom, here `count`. We can assign to it with `g_assign/2`, increment it with `g_inc/1` and read its value with `g_read/2`. The idea is to let a call to `solve` succeed, increment a global variable `count`, then fail and backtrack into `solve` to find an another solution, etc., until there are no further solutions.

```
test(N) :-
        g_assign(count, 0),
        solve(N),
        g_inc(count),
        fail.
test(N) :-
        g_read(count, K),
        format("At ~p, ~p solutions\n", [N,K]).
```

This works, because the effect of incrementing `count` remains, even when backtracking fails past it.

Although global variables are clearly non-logical, their operational semantics is not so difficult to specify (see Exercise 11.4).

This example reveals imperative or potentially imperative operations on several levels. In the next lecture we will explore one of them.

11.6 Historical Notes

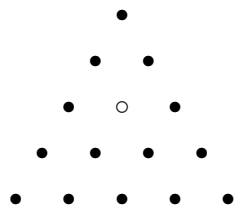
Okasaki has given a perceptive analysis of purely functional queues [2]. According to Sterling and Shapiro [3], difference lists have been Prolog folklore since the early days, with a first published description in a paper by Clark and Tärnlund [1].

11.7 Exercises

Exercise 11.1 *Reconsider the Dutch national flag problem introduced in Exercise 2.4 and give an efficient solution using difference lists.*

Exercise 11.2 *Think of another interesting application of difference lists or related incomplete data structures and write and explain your implementation.*

Exercise 11.3 *Give a Prolog implementation of the following triangular version of peg solitaire*



where jumps can be made in 6 directions: east, northeast, northwest, west, southwest, and southeast (but not directly north or south). Use your program to determine the number of solutions (you may count symmetric ones), and in which locations the only remaining peg may end up in. Also, what is the maximal number of pegs that may be left on the board without any possible further moves?

Exercise 11.4 *Give an extension of the operational semantics with goal stacks, failure continuations, and explicit unification to model global variables (named by constants) which can be assigned, incremented, decremented, and read. Glean their intended meaning from the use in the* `test/1` *predicate for peg solitaire.*

You may assume that the values assigned to global variables in this manner are ground and remain unaffected by backtracking.

11.8 References

[1] K. L. Clark and S.-A. Tärnlund. A first order theory of data and programs. In B. Gilchrist, editor, *Proceedings of the IFIP Congress*, pages 939–944, Toronto, Canada, 1977. North Holland.

[2] Chris Okasaki. Simple and efficient purely functional queues and deques. *Journal of Functional Programming*, 5(4):583–592, October 1995.

[3] Leon Sterling and Ehud Shapiro. *The Art of Prolog*. The MIT Press, Cambridge, Massachusetts, 2nd edition edition, 1994.