Notes for Lecture 25 (Fall 2022 week 11, part 3): More on Prolog cuts

Jana Dunfield

November 24, 2022

1 Cuts in nub?

Let's look at the code for nub, which is a predicate that removes duplicate elements (like Haskell's library function nub):

```
nub([],        []).
nub([X | Xs], Ys)       :- member(X, Xs), nub(Xs, Ys).
nub([X | Xs], [X | Ys]) :- \+ member(X, Xs), nub(Xs, Ys).
```

There are 7 sites in this code where we could try to add a cut:

1. by creating a new premise in the first clause;
2. in the second clause, by adding a cut before the first premise;

3. in the second clause, by adding a cut between the premises;
4. in the second clause, by adding a cut after the last premise;

5. in the third clause, by adding a cut before the first premise;
6. in the third clause, by adding a cut between the premises;
7. in the third clause, by adding a cut after the last premise.

When thinking about the behaviour of a cut, we often need to think about how we intend the predicate to be used. I only care about queries to nub where the first argument is an actual list (not a Prolog variable), so I won't consider what would happen for a query like nub(Xs, [1,2,3]). In the discussion below, I assume the first argument to a query involving nub is *not* a Prolog variable (and contains no Prolog variables). I will not assume anything about the second argument. So we could give a query like

```
?- nub([5,4,3,4,4,3], Answer).
```

or one like

```
?- nub([5,4,3,4,4,3], [4,3,4]).
```

(That one should be false. The 5 is missing from the second argument.)

1.1 Possible cut 1

Adding a cut to the first clause has no effect:

```
nub([],        [])       :- !.
nub([X | Xs], Ys)       :- member(X, Xs), nub(Xs, Ys).
nub([X | Xs], [X | Ys]) :- \+ member(X, Xs), nub(Xs, Ys).
```

Prolog will only see the cut if the first argument in the query has matched `[]`. If the first argument is the empty list, then it will not match the non-empty lists in the second and third clauses. There are no other paths to explore, so there are no paths to cut.

1.2 Possible cut 2

Let's add a cut to the start of the 2nd clause.

```
nub([],        []).
nub([X | Xs], Ys)       :- !,
                            member(X, Xs), nub(Xs, Ys).
nub([X | Xs], [X | Ys]) :- \+ member(X, Xs), nub(Xs, Ys).
```


Now, whenever our query gives a non-empty list as the first argument to `nub`:

- Prolog will figure out which paths need to be explored. The 2nd clause and the 3rd clause both have a non-empty list `[X | Xs]` as the first argument in the head (the conclusion of the clause, to the left of the `:-`), so these two paths are added to Prolog's "to do" list.

- Prolog will try the 2nd clause first, because it appears above the 3rd clause in the source file.
- Prolog will see the cut, and will forget about alternative paths. There is one alternative to the current path: the 3rd clause. Prolog will forget about the third clause. But we need the third clause to handle any list element that is *not* repeated. (Try it.)

So, this cut is Not Good. More precisely, it is a "red" cut that changes the meaning of the predicate, and in a way that creates a major bug.

1.3 Possible cut 3

Let's add a cut to the *middle* of the 2nd clause.

```
nub([],        []).
nub([X | Xs], Ys)       :- member(X, Xs), !, nub(Xs, Ys).
nub([X | Xs], [X | Ys]) :- \+ member(X, Xs), nub(Xs, Ys).
```

Now, whenever our query gives a non-empty list as the first argument to `nub`:

- Prolog will figure out which paths need to be explored. The 2nd clause and the 3rd clause both have a non-empty list `[X | Xs]` as the first argument in the head (the conclusion of the clause, to the left of the `:-`), so these two paths are added to Prolog's "to do" list.

- Prolog will try the 2nd clause first, because it appears above the 3rd clause in the source file.
- Prolog will check the premise `member(X, Xs)`.
  - If the premise `member(X, Xs)` is *false* (`X` does not appear in `Xs`, that is, `X` is not duplicated in the input list), Prolog will give up on this path, and try the 3rd clause. Prolog will not see the cut, so the cut has no effect and we have not created a bug (yet?).
  - If the premise `member(X, Xs)` is *true* (`X` does appear in `Xs`, so `X` is duplicated), Prolog will move on to the next premise, which is the cut. It will forget about the unexplored path through the 3rd clause.

    (It also forgets about unexplored paths within the `member` predicate! See the discussion below about repeated solutions.)

    But this unexplored path is *not* a problem: The premise `member(X, Xs)` is true. If we explore the path through the 3rd clause, the first premise is `\+ member(X, Xs)`, which is the *negation* of `member(X, Xs)`. This first premise of the 3rd clause is true only if `member(X, Xs)` is false. But `member(X, Xs)` is true so the first premise of the 3rd clause is false. The 3rd clause will always fail, so there is no need to try it.

So we have not created a bug with this cut: if `member(X, Xs)` is true, we cut off a path that would definitely fail, which is harmless. In fact, it saves time, since Prolog will not have to search `Xs` twice. If `member(X, Xs)` is false, the second clause fails and we try the third clause.

This is a *green* cut: it does not change the results to queries.

This cut does reduce the number of repeated solutions, however. If `X` appears *more than once* in `Xs` (so it actually appears a total of three or more times in the argument `[X | Xs]`), then there are multiple ways for `member` to be true, and typing a semicolon will give extra solutions.

Using the original version of nub, without this cut, we get:


```
?- nub([10,10,10], Answer).
Answer = [10]  % type a ;
Answer = [10]
false.
```

After adding the cut, we get:

```
?- nub([10,10,10], Answer).
Answer = [10].
```

Prolog does not wait for us to type anything, because the cut eliminated a useless extra path within the `member` predicate.

1.4 Possible cut 4

We move the cut to the end of the 2nd clause:

```
nub([],       []).
nub([X | Xs], Ys)       :- member(X, Xs), nub(Xs, Ys), !.
nub([X | Xs], [X | Ys]) :- \+ member(X, Xs), nub(Xs, Ys).
```

When Prolog sees this cut, it will forget all the unexplored paths. That includes forgetting the unexplored path through the 3rd clause, which is fine. It will also forget the unexplored paths within the recursive call to `nub(Xs, Ys)`. Is that okay?

Actually, yes. If we reach this cut, all the "real" (non-cut) premises of the 2nd clause were true. So the clause is about to succeed. `nub` is supposed to behave like a function: for every list we give as the first argument, we should get a unique list as the second argument. If the input is, say, `[1,2,2,1,5,3,5,6,5,3]`, the output should always be the same. (The behaviour of `nub` as to *which* duplicate element it removes might be a little surprising—it removes the first occurrence, not the last—but it is consistent.)

The "Possible cut 3" already did the job of eliminating repeated solutions, so moving the cut to the end of the clause doesn't help more than "Possible cut 3" did.

Both "Possible cut 3" and "Possible cut 4" are really useful. To see why, remove the cut and try queries like

```
?- nub([3,3,3,3], Answer).
```

with increasing numbers of duplicated `3` elements.

(If you like discrete math, try to figure out how the number of repeated identical solutions increases with the number of duplicated elements. We get 2 solutions, 6 solutions, 24 solutions, then 120 solutions. Yes, I *did* type semicolon 120 times.)

1.5 Possible cut 5


```
nub([],          []).
nub([X | Xs], Ys)        :- member(X, Xs), nub(Xs, Ys).
nub([X | Xs], [X | Ys]) :- !, \+ member(X, Xs), nub(Xs, Ys).
```


When we reach this cut, we are exploring the last clause matching the query, and we haven't checked any real premises yet, so there are no other paths to explore. This cut does nothing.

1.6 Possible cut 6

```
nub([],          []).
nub([X | Xs], Ys)        :- member(X, Xs), nub(Xs, Ys).
nub([X | Xs], [X | Ys]) :- \+ member(X, Xs), !, nub(Xs, Ys).
```

When we reach this cut, we are exploring the last clause matching the query, so there are no other clauses to explore. However, we are cutting off any unexplored paths within `\+ member(X, Xs)`.

But there are no such paths. For a predicate to be *false* (the `\+` negates the predicate), Prolog must have already explored all possible paths within it. Since we reached the cut, the premise `\+ member(X, Xs)` was true, meaning `member(X, Xs)` was false, meaning there were zero solutions to `member(X, Xs)`.

So this cut also does nothing.

1.7 Possible cut 7

```
nub([],          []).
nub([X | Xs], Ys)        :- member(X, Xs), nub(Xs, Ys).
nub([X | Xs], [X | Ys]) :- \+ member(X, Xs), nub(Xs, Ys), !.
```

This one is really interesting.

I ran the same query I've been using to test: `?- nub([3,3,3], Answer).` Just as with "Possible cut 5" and "Possible cut 6", I got duplicate solutions. I was about to write "this cut also has no effect", but that seemed wrong—since this cut is after the recursive call `nub(Xs, Ys)`, it should eliminate the duplicate paths *within the second clause*.

So:

```
?- nub([3,3,3], Answer).
Answer = [3]       % type ;
Answer = [3]
false.

?- nub([1,3,3,3], Answer).
Answer = [1, 3].   % finishes immediately
```

In the first query, `nub([3,3,3], Answer).`, we always go through the second clause: `member(X, Xs)` always succeeds, except for the last element.

In the second query, `nub([1,3,3,3], Answer).`, the second clause fails because `1` is *not* a member of `[3,3,3]`. We try the third clause:

- Check premise `\+ member(X, Xs)`. Succeeds, because `member(X, Xs)` is false, so `\+ member(X, Xs)` is true.
- Check premise `nub(Xs, Ys)`. This succeeds with `Ys = [3]`.
- Check premise `!`. This succeeds (a cut, as a premise, is always true), so we cut off the paths within `nub(Xs, Ys)` that weren't taken.

This made sense to me, but something bothered me about the first query.

Every successful query to `nub` eventually goes through the third clause. When we get to the last element of a list, the tail is empty. The `member` premise in the second clause will fail, so the third clause will be tried, and will always succeed. That means the cut will be reached.

If a cut eliminates all the paths we haven't yet taken, why didn't the cut eliminate the paths within the second clause?

1.8 Possible cut 7, continued

To understand what's going on, it may help to understand why we actually get 2, then 6, then 24, then 120 solutions as we increase the number of repeated elements in the list.

(The following should work for `nub` with possible cuts 5, 6, and 7.)

```
?- nub([1,1,1], Answer).
Answer = 1  % type ;
Answer = 1  % type ;
false.
```

We get two solutions because the second clause asks "is `member(1, [1,1])` true?" and there are two ways for it to be true, depending on whether `member` finds the first 1 in `[1,1]` or the second 1 in `[1,1]`.

When we get to the second 1, in the recursive call nub(Xs, Ys), which is nub([1,1], Ys), we call member(1, [1]). There is only one way for that to be true, because [1] contains only one 1.

If we add one more 1, then:

```
?- nub([1,1,1,1], Answer).
Answer = 1  % type ;
Answer = 1  % type ;
Answer = 1  % type ;
Answer = 1  % type ;
Answer = 1  % type ;
Answer = 1  % type ;
false.
```

There are three ways to find 1 in [1,1,1]. But then the recursive call is nub([1,1,1], Ys). We then ask "is the head of [1,1,1] in the tail", that is, "is member(1, [1,1]) true?" There are two ways for that to be true.

The tree on the next page is my attempt to represent the paths taken by Prolog as we type semicolon.

```
             nub([1,1,1,1], Answer)
                     |
                     |
                member(1,
                  [1,1,1])
                   A B C
                  /  |  \
                _/   |   \_
               /     |     \
          found    found  found
            A        B       C
           /         |        \
          /          |         \
  nub([1,1,1],  nub([1,1,1],  nub([1,1,1],
      Ys)           Ys)           Ys)
         /           |             \
        /            |              \
    member(1,    member(1,          member(1,
      [1,1])       [1,1])                  [1,1])
       B C          B C                    B C
      /   \        /   \                  /   \
 found   found  found  found          found   found
   B       C      B      C              B       C
   |       |      |      |              |       |
  ...     ...    ...  nub([1,1], Ys2)  ...     ...
                         |
                      member(1, [1])
                                 C
                                 |
                              found C
                                 |
                            nub([1], Ys3)
                              /      \
                (2nd clause fails   try 3rd clause
                 because                |
                 member(1, [])    \+ member(1, [])
                 is false)              |
                                     nub([], Ys4)
                                        |
                                   1st clause succeeds
```


In all of these paths, Prolog starts by calling `member(1, [1,1,1])` finds the first 1 (marked `A`, this is not a Prolog variable, it's just in this diagram). After `member` returns, the last premise of the second clause defining `nub` calls `nub([1,1,1], Ys)`, which calls `member(1,[1,1])`. Then we call `nub([1,1], Ys2)`, which calls `member(1, [1])`, and then we call `nub([1], Ys3)`, which succeeds with `Ys3 = [1]`.

Once we get down to the `nub([1,1], Ys2)`, there is only one path down to a solution. But at the beginning (top of the diagram), there are 3 paths because there are 3 ways for `member(1,[1,1,1])` to be true. Going down, there are 2 ways for `member(1,[1,1])` to be true. So there are a total of 6 paths.

If we add another `1` to the input to `nub`, we will add a 4-way branching at the root, which gives $4 \cdot 3 \cdot 2 = 24$ paths.

One more query (with possible cut 7):

```
?- nub([3,3,3,6,6,6,6], Answer).
Answer = [3, 4]    % type ;
Answer = [3, 4]    % type ;
false.
```

There are three 3's, and four 6's.

This query gives two solutions. These two solutions are caused by the call `member(3, [3,3,6,6,6,6])`, which has two solutions (there are two 3's in `[3,3,6,6,6,6]`). When we reach the *last* 3, Prolog finally sees the cut in the last clause, which forgets all the other paths within `nub(Xs, Ys)`, that is, `nub([6,6,6,6], Ys)`.

It doesn't matter how many extra 6's we put in the input—we will still have only two solutions.

1.9 Conclusion

In the diagram (if it made any sense), each path does the same things in the same order. It doesn't matter which element `member` notices; the important thing is that if `member` is true, `X` is duplicated and we don't include it at the head of the output.

Possible cuts 3 or 4 are the best ones to add: adding a cut in the second clause (but *after* the call to `member`) cuts off all the other paths.

Possible cut 7 is better than nothing but it only gets some of the extra paths, not all.

2 Cuts in `first/second`?

Discussion to be added. (I decided not to talk about this. Discussion of a different example is below.)

3 Cuts in `add1or2/times10or100`

The predicates `add1or2` and `times10or100` take in a number and do something with it.

```
?- add1or2(50, Z).
Z = 51    % type ;
Z = 52.

?- times10or100( 3, Z).
Z = 30    % type ;
Z = 300.
```

Both predicates are defined with semicolons, which mean "or". Each semicolon is a choice. For example, in `add1or2`, Prolog will try `X is A + 1` first. If we type a semicolon to look for more solutions, Prolog backtracks and tries `X is A + 2`.

```
add1or2(A, X) :- (X is A + 1; X is A + 2).

times10or100(B, Y) :- (Y is B * 10; Y is B * 100).

addtimes(A, Y) :- add1or2(A, X), times10or100(X, Y).
```

The predicate `addtimes` uses both `add1or2` and `times10or100`:

```
?- addtimes(50, Z).
Z = 510     % type ;
Z = 5100    % type ;
Z = 520     % type ;
Z = 5200.   % type ;
```

In the first solution, Prolog used `X is A + 1` and `Y is B * 10`.
In the second solution, Prolog used `X is A + 1` and `Y is B * 100`.
In the third solution, Prolog used `X is A + 2` and `Y is B * 10`.
In the fourth solution, Prolog used `X is A + 2` and `Y is B * 100`.
What happens if we add a cut to the *end* of `times10or100`?


```
add1or2(A, X) :- (X is A + 1; X is A + 2).

times10or100(B, Y) :- (Y is B * 10; Y is B * 100), !.

addtimes(A, Y) :- add1or2(A, X), times10or100(X, Y).

?- addtimes(50, Z).
Z = 510
Z = 520.
```


By comparing these to the earlier solutions, the first solution `Z = 510` used `X is A + 1` and `Y is B * 10`.

The second solution `Z = 520` used `X is A + 2` and `Y is B * 10`.

We are missing the solutions that used `Y is B * 100`. The cut we added to `times10or100` only affects the paths created *after we started calling* `times10or100`. It doesn't affect the paths created earlier, in `add1or2`.