Programming Paradigms

Unit 9 — Prolog: Accumulators, Order of Goals/Clauses, and the Cut

J. Gamper

Free University of Bozen-Bolzano
Faculty of Computer Science
IDSE

Outline

Outline

Accumulators

- Frequently we need to traverse a Prolog structure or list and calculate a result which depends on what was found so far
- At intermediate stages of the traversal, we will have an intermediate result
- A common technique to represent the intermediate result is to use an argument of the predicate, termed an accumulator

Example: `listlen/1` without Accumulator

- Consider predicate `listlen(L,N)` that succeeds if `N` is the length of list `L`
  - Some Prolog implementations have a built-in predicate `length(L,N)`
- Lets first look at an implementation without an accumulator

```
listlen([],0).
listlen([H|T],N) :- listlen(T,N1),
                    N is N1 + 1.
```

- The base case is a fact stating that the empty list has length `0`
- The recursive case is a rule stating that the length of a non-empty list can be calculated by adding one to the length of the tail of the list
- Notice that this solution is not tail-recursive

Example: `listlen/1` with Accumulator/1

- Now lets look at a solution with an accumulator

```
listlen(L,N) :- listlen_acc(L,0,N).

listlen_acc([],A,A).
listlen_acc([H|T],A,N) :- A1 is A + 1
                          listlen_acc(T,A1,N).
```

- `listlen_acc(L,A,N)` is an auxiliary predicate that uses an accumulator `A` and has the following meaning:
  - the length of list `L`, when added to the number `A`, is `N`
- The base case of `listacc` states that the length of the list is whatever has been accumulated so far, i.e., `A`
  - Equivalent to adding `0` (length of the empty list) to `A`
- In the recursive clause, `1` is added to the accumulated amount `A`, and recur on the tail of the list with a new accumulator value `A1`
- This solution is tail-recursive

Example: `listlen/1` with Accumulator/2

- Notice that the final argument `N` of the recursive subgoal is the same as the final argument in the head of the clause
- This means that the length `N` returned for the whole list will be the number that the recursive subgoal calculates
  - Thus, the production of the final result is delegated to the recursive subgoal
  - All extra information that is needed to construct the final result is provided by the accumulator
- We get a sequence of `listlen_acc` goals, all sharing the same last argument `N`

```
listlen_acc([a,b,c],0,N).
listlen_acc([b,c],1,N).
listlen_acc([c],2,N).
listlen_acc([],3,N).
```

- The last goal will be matched with the first clause and instantiates `N`=3, which is returned as the final result

Using a List as Accumulator/1

- The accumulator need not to be a number, but can be a list or another structure
- Consider the predicate `revere(L,R)` that is true if list `R` is the reverse of list `L`

```
reverse([],[]).
reverse([H|T], R) :- reverse(T, R1),
                     append(R1, [H], R).
```

- This predicate does not use an accumulator and is expensive because `append` is expensive
- In fact, it runs in $O(n^2)$ time (prove this!)
- Moreover, it is not tail-recursive

Using a List as Accumulator/2

- The predicate `reverse` written with an accumulator

```
reverse(L, R) :- reverse_acc(L, [], R).

reverse_acc([], R, R).
reverse_acc([H|T], R1, R) :- reverse_acc(T, [H|R1], R).
```

- In the predicate `reverse_acc(L, A, R)`, list `A` is used to accumulate partial reversals of `L`.
- `L` holds the part of the list that remains to be reversed
- When `L` is empty, `A` holds the reversed list and the first clause instantiates `R` to `A`.
- Notice the linear complexity of this version!

Outline

Order of Subgoals/1

- Recall our generalized rule about friends: *"X and Y are friends, if X and Y like the same Z and X and Y are not the same"*
- What if we translate this in the following Prolog rule:

```
friend(X,Y) :- \+(X=Y), likes(X,Z), likes(Y,Z).
```

- Might not look like a big change, but this has serious consequences
- If we run the query `friend(wallace,Y)` with the above rule, we get

```
?- friend(wallace,Y).
no
```

- But both of them like cheese! What is going wrong?

Knowledge base

```
likes(wallace, toast).
likes(wallace, cheese).
likes(gromit, cheese).
likes(gromit, cake).
likes(wendolene, sheep).
```

Order of Subgoals/2

- The subgoals are in the wrong order!
- The position of the predicate `\+(X=Y)` has a big impact
- Prolog tries to satisfy subgoals from left to right
- `\+(X=Y)` fails if `X=Y` can be satisfied
  - `X` and `Y` start off uninstantiated in the above case
  - This makes `X=Y` true, resulting in `\+(X=Y)` being false
  - Consequently, the first subgoal always fails

Order of Subgoals/3

- If we arrange the predicates in a different order

```
friend(X, Y) :- likes(X,Z), likes(Y,Z), \+(X=Y).
```

- then `X` and `Y` will be instantiated when reaching the subgoal `\+(X=Y)`
- If `X` and `Y` have a different value at that point, then `\+(X=Y)` will succeed
- It is important to get the order right in which variables are instantiated!

Order of Clauses/1

- Similar as for subgoals in a body of a rule, the order of clauses/rules is crucial
- Consider the following predicate `is_in2` to find/check elements in a list
  - We just swapped the order of the base case and the recursive case

```
is_in2(X, [_|T]) :- is_in2(X, T).
is_in2(X, [X|_]).
```

- Checking membership and the enumeration of the elements works

```
?- is_in2(a,[1,2,a]).
true
```

```
?- is_in2(X,[1,2,a]).
X = 1;
X = 2;
X = a
```

Order of Clauses/2

- But what happens with the query `is_in2(a, L)`?

```
is_in2(X, [_|T]) :- is_in2(X, T).
is_in2(X, [X|_]).
```

- Rules are matched in the order they occur in the program
- In this example, the first predicate/rule always succeeds, leading to a recursive call
- As a consequence, the base case never occurs, and we get a stack error

```
?- is_in2(a, L)
ERROR: Out of local stack
```

Outline

"Cutting" the Number of Solutions/1

- If you ask Prolog to keep looking for further solutions (by answering with ";"), it will go through all possible solutions using backtracking:

```
dance_pairs(X,Y) :- boy(X), girl(Y).
boy(adam).
boy(bert).
...
girl(angela).
girl(betty).
...
```

```
?- dance_pairs(X,Y).
X = adam, Y = angela ;
X = adam, Y = betty ;
...
```

"Cutting" the Number of Solutions/2

- Sometimes we are not interested in exhaustively going through all solutions:
  - We only want to know if a solution exists
  - We are happy with a certain subset of solutions
  - In some recursive cases, there may be an infinite number of solutions
- For instance, the following goal succeeds twice

```
is_in(a, [a,b,a]).
true ;
true ;
false.
```

- However, to check membership, it should succeed only once!
- Prolog provides the cut operator to force it not to consider certain choices

The Cut Operator/1

- The cut operator is denoted by `"!"` and can be inserted into a rule as a subgoal
- What does it do?

```
foo :- a, b.
foo :- c, d, !, e, f.
foo :- g, h.
```

- First of all, the `!` operator always succeeds
  - e.g., if `c` and `d` are satisfied in the second rule, then Prolog will immediately start matching `e`
- Second, the `!` operator stops backtracking

The Cut Operator/2

- There are two levels where backtracking occurs and is blocked by `!`
- First, backtracking within a rule
  - Assuming `c` and `d` are satisfied while checking the second rule, then the choices made for `c` and `d` are "locked in"
    - Prolog may not go back and search for other solutions for `c` and `d`
    - It may still do backtracking for `e` and `f`, though
- Second, backtracking across rules
  - If the second rule fails after the `!` operator, Prolog may not go beyond this rule to try to satisfy `foo`
    - In particular, it will not try out the third rule `foo :- g,h.`

```
foo :- a, b.
foo :- c, d, !, e, f.
foo :- g, h.
```

The `memberchk/2` Predicate

- The built-in predicate `memberchk/2` checks whether the first parameter occurs in the list of elements passed as second parameter

```
memberchk(X,[X|_]) :- !.
memberchk(X,[_|T]):- memberchk(X,T).
```

- The `!` operator is used to avoid that the predicate succeeds for every occurence of the first parameter in the second parameter.

```
memberchk(a, [a,b,a]).
true.
```

Using the Cut Operator

- We will show three common ways how to use the cut operator in practice:
  - to confirm a choice of a rule
  - as a cut-fail combination
  - in the generate and test pattern

Confirming Choice of a Rule/1

- The first use is to tell Prolog that it has found the "right rule" to apply
- Assume we want to add up the numbers from 1 to N

```
sum_to(1,1).
sum_to(N,Result) :-
    TmpN is N-1,
    sum_to(TmpN,TmpRes),
    Result is TmpRes + N.
```

- While this works, it may start an infinite recursion if we ask for a second solution

```
?- sum_to(3,X).
X = 6 ?  ;
Fatal Error:  local stack overflow
```

- So, what happens here?

Confirming Choice of a Rule/2

- The first solution is reported when the first rule matches for the first time, i.e., `sum_to(1,TmpRes)`
- Asking for another solution forces Prolog to search for another solution for `sum_to(1,TmpRes)`, applying the second rule `sum_to(N,Result)` to it
- The second rule executes a recursive call `sum_to(0,TmpRes)`
- This in turn will match again with the second rule, which executes a recursive call `sum_to(-1,TmpRes)` and so on ... yielding an endless loop

Confirming Choice of a Rule/3

- We want to tell Prolog that once it has matched the fact `sum_to(1,1)`, it should not try searching for further solutions
- We can achieve this by rewriting the first rule

```
sum_to(1,1) :- !.
sum_to(N,Result) :-
    TmpN is N-1,
    sum_to(TmpN,TmpRes),
    Result is TmpRes + N.

?- sum_to(3,X).
X = 6.
```

- Notice that Prolog is not offering you a second solution!

Confirming Choice of a Rule/4

- We could just tell Prolog to stop searching for further solutions in the above example
- However, this may not always be under our control

```
go :- sum_to(1,X), foo(apples).

?- go.
```

- If `foo(apples)` fails, then this will trigger backtracking on `sum_to(1,X)`

"Cut-Fail" Combination/1

- The second use of the cut operator is as a "cut-fail" combination
  - i.e., the `!` operator followed by the `fail` predicate: `!, fail`
  - The built-in predicate `fail` cannot be satisfied
- This pattern is used in situations where it is known that, if the current rule fails, there is no need trying further rules for the same predicate since no other solutions exists

"Cut-Fail" Combination/2

- Consider the following step function

$$f(x) = \begin{cases} 0 & \textit{if } X < 3 \\ 2 & \textit{if } X >= 3 \wedge X < 6 \\ 4 & \textit{if } X >= 6 \end{cases}$$

- This is the corresponding Prolog code:

```
f(X,0) :- X < 3.
f(X,2) :- X >= 3, X < 6.
f(X,4) :- X >= 6.
```

- What happens when we ask:

```
?- f(1,Y), 2 < Y.
```

"Cut-Fail" Combination/3

- What happens when we ask:

```
?- f(1,Y), 2 < Y.
```

- The first rule matches and binds variable `Y` to `0`
- The second goal becomes `2 < 0`, which fails
- Prolog then tries through backtracking two other (useless) alternatives for `f(1,Y)`
- Since the 3 rules for `f(.,.)` are mutually exclusive, there is no point in trying another one
- The `!` operator explicitly tells Prolog not to do so and waste time if no other solution exists

"Cut-Fail" Combination – Another Example

- To figure out the correct tax rate for people, we define a predicate to determine average taxpayers
- However, there is a special tax rate for non-residents, i.e., they never pay the average rate

```
average_taxpayer(X) :- non_resident(X), fail.
average_taxpayer(X) :- income(X,Inc), ...
```

- This will not work, as a non-resident will fail the first rule and then one of the following rules will be applied
- However, that's exactly what we don't want to happen
- The following will make sure that none of the following rules will be applied

```
average_taxpayer(X) :- non_resident(X), !, fail.
average_taxpayer(X) :- income(X,Inc), ...
```

Generate and Test/1

- The third major use of cut is in the "generate and test" pattern
- It is a common programming pattern in Prolog and has the form

```
foo :- g1, g2, ..., gn,
       t1, t2, ..., tm.
```

- The sequence of predicates `g1, g2, ..., gn` is called generator and can succeed in many different ways
  - They generate lots of different potential solutions
- The sequence of predicates `t1, t2, ..., tm` is called tester and tests whether something generated by `g1, g2, ..., gn` is actually a solution
  - If something is not a solution, this causes `g1, g2, ..., gn` to backtrack and generate the next candidate

Generate and Test/2

- We want to define integer division just using addition and multiplication
- First we build a predicate that generates all integers

```
is_integer(0).
is_integer(X) :- is_integer(Y), X is Y+1.
```

- Then we check the numbers generated by `is_integer`

```
idiv(X,Y,Result) :-
    is_integer(Result),
    Prod1 is Result*Y,
    Prod2 is (Result+1)*Y,
    Prod1 =< X,
    Prod2 > X,
    !.
```

Generate and Test/3

- The first line in `idiv` is the generator, the other lines implement the tester
- We know that there can only be one possible solution
- After reaching it, we stop the search
- Otherwise, on backtracking `is_integer` would keep on producing integers
- None of these integers would pass the test, resulting in an endless loop

Cutting too Deeply/1

- The cut operator is a dangerous tool and should be used sparingly!
- It can behave in unexpected ways.
- We want to formulate that every person has two parents, except Adam and Eve who have no parents

```
number_of_parent(adam,0) :- !.
number_of_parent(eve,0) :- !.
number_of_parent(X,2).
```

- Works fine if we want to retrieve the number of parents

```
?- number_of_parent(eve,X).
X = 0
```

```
?- number_of_parent(john,X).
X = 2
```

- But does not correctly verify the number of parents

```
?- number_of_parent(eve,2).
yes
```

Cutting too Deeply/2

- A general lesson is that *if we introduce cuts to obtain correct behaviours when the goals are of one form, there is no guarantee that anything sensible will happen if goals of another form start appearing.*
- The goal

  ```
  number_of_parent(eve,X)
  ```

  works fine, while

  ```
  number_of_parent(eve,2)
  ```

  does not!
- How can we address this problem?

Replace Cut by Not/1

- It is considered good programming style to replace cuts by the use of negation, \+ or `not` (if possible)

```
parent(adam,0).
parent(eve,0).
parent(X,2) :- \+(X = adam), \+(X = eve).

?- parent(eve,X).
X = 0 ?  ;
no

?- parent(john,X).
X = 2

?- parent(eve,2).
no
```

Replace Cut by Not/2

- The program computing the sum of the numbers from 1 to N can also be rewritten without a cut operator

```
sum_to(1,1).
sum_to(N,Result) :-
    not(N=1),
    TmpN is N-1,
    sum_to(TmpN,TmpRes),
    Result is TmpRes + N.
```

or

```
sum_to(N,1) :- N =< 1.
sum_to(N,Result) :-
    N > 1,
    TmpN is N-1,
    sum_to(TmpN,TmpRes),
    Result is TmpRes + N.
```

- This programming style makes it clear, which rule to use when

Cut versus Not

- A general pattern with `cut` might look as following

```
A :- B, !, C.
A :- D.
```

- A general pattern with `not` might look as following

```
A :- B, C.
A :- not(B), D.
```

- The `not` operator makes the program more readable
- The `cut` operator makes the program more efficient
  - If backtracking occurs, `B` needs to be evaluated for the second rule as well