Lecture Notes on Prolog

15-317: Constructive Logic
Frank Pfenning*

Lecture 15
October 20, 2016

In this lecture we introduce some simple data structures such as lists, and simple algorithms on them such as as quicksort or mergesort and how they are implemented as recursive functions in Prolog. We also introduce some first considerations of types and modes for logic programs. Then we examine some basic non-logical features for controlling search in Prolog.

1 Lists

Lists are defined by two constructors: the empty list nil and the constructor cons which takes an element and a list, generating another list. For example, the list a, b, c would be represented as $\mathsf{cons}(a, \mathsf{cons}(b, \mathsf{cons}(c, \mathsf{nil})))$. The official Prolog notation for nil is `[]`, and for $\mathsf{cons}(h, t)$ is `.(`h`, `t`)`, overloading the meaning of the period '`.`' as a terminator for clauses and a binary function symbol. In practice, however, this notation for cons is rarely used. Instead, most Prolog programs use `[`h`|`t`]` for $\mathsf{cons}(h, t)$.

There is also a sequence notation for lists, so that the list a, b, c consisting of these three elements can be written as `[a, b, c]`. It could also be written as `[a | [b | [c | []]]]` or `[a, b | [c, []]]`. Note that all of these notations will be parsed into the same internal form, using nil and cons. We generally follow Prolog list notation in these notes.

*With edits by André Platzer

2 Type Predicates

We now return to the definition of plus from the previous lecture, except that we have reversed the order of the two clauses. That makes the clause that does not involve any further search first so that it will be used first.

```
plus(z, N, N).
plus(s(M), N, s(P)) :- plus(M, N, P).
```

In view of the new list constructors for terms, the first clause now looks wrong. For example, with this clause we can prove

```
plus(s(z), [a, b, c], s([a, b, c])).
```

This is absurd: what does it mean to add 1 and a list? What does the term `s([a, b, c])` denote? It is clearly neither a list nor a number.

From the modern programming language perspective the answer is clear: the definition above lacks *types*. Unfortunately, Prolog (and traditional predicate calculus from which it was originally derived) do not distinguish terms of different types. The historical answer for why these languages have just a single type of terms is that types can be defined as unary predicates. While this is true, it does not account for the pragmatic advantage of distinguishing meaningful propositions from those that are not. To illustrate this, the standard means to correct the example above would be to define a predicate nat with the rules

$$\frac{}{\mathsf{nat}(\mathsf{0})}\ \mathsf{nz} \qquad\qquad \frac{\mathsf{nat}(N)}{\mathsf{nat}(\mathsf{s}(N))}\ \mathsf{ns}$$

and modify the base case of the rules for addition

$$\frac{\mathsf{nat}(N)}{\mathsf{plus}(\mathsf{0}, N, N)}\ \mathsf{pz} \qquad\qquad \frac{\mathsf{plus}(M, N, P)}{\mathsf{plus}(\mathsf{s}(M), N, \mathsf{s}(P))}\ \mathsf{ps}$$

Since its premise recursively requires M, N, P to be natural numbers via pz already, the ps rule does not need $\mathsf{nat}(M)$, $\mathsf{nat}(N)$ and $\mathsf{nat}(P)$ as premises.

One of the problems is that now, for example, $\mathsf{plus}(\mathsf{0}, \mathsf{nil}, \mathsf{nil})$ is *false*, when it should actually be meaningless. Many problems in debugging Prolog programs can be traced to the fact that propositions that should be meaningless will be interpreted as either true or false instead, incorrectly succeeding or failing. If we transliterate the above into Prolog, we get:

```
nat(z).
nat(s(N)) :- nat(N).

plus(z, N, N) :- nat(N).
plus(s(M), N, s(P)) :- plus(M, N, P).
```

No self-respecting Prolog programmer would write the `plus` predicate this way. Instead, he or she would omit the type test in the first clause leading to the earlier program. The main difference between the two is whether meaningless clauses are false (with the type test) or true (without the type test). One should then annotate the predicate with the intended domain.

```
% plus(m, n, p) iff m + n = p for nat numbers m, n, p.
plus(z, N, N).
plus(s(M), N, s(P)) :- plus(M, N, P).
```

It would be much preferable from the programmer's standpoint if this informal comment were a formal type declaration, and an illegal invocation of `plus` were a compile-time error rather than leading to silent success or failure. There has been some significant research on types systems and type checking for logic programming languages [5] and we will talk about types more later in this course.

3 List Types

We begin with the type predicates defining lists.

```
list([]).
list([X|Xs]) :- list(Xs).
```

Unlike languages such as ML, there is no test whether the elements of a list all have the same type. We could easily test whether something is a list of natural numbers.

```
natlist([]).
natlist([N|Ns]) :- nat(N), natlist(Ns).
```

The generic test, whether we are presented with a homogeneous list, all of whose elements satisfy some predicate `P`, would be written as:

```
plist(P, []).
plist(P, [X|Xs]) :- P(X), plist(P, Xs).
```

While this is legal in some Prolog implementations, it can not be justified from the underlying logical foundation, because `P` stands for a predicate and is an argument to another predicate, `plist`. This is the realm of higher-order logic, and a proper study of it requires a development of *higher-order logic programming* [3, 4]. In Prolog the goal `P(X)` is a *meta-call*, often written as `call(P(X))`.[1] We will avoid its use, unless we develop higher-order logic programming later in this course.

4 List Membership and Disequality

As a second example, we consider membership of an element in a list.

$$\frac{}{\mathsf{member}(X, \mathsf{cons}(X, Ys))} \qquad \frac{\mathsf{member}(X, Ys)}{\mathsf{member}(X, \mathsf{cons}(Y, Ys))}$$

In Prolog syntax:

```
% member(X, Ys) iff X is a member of list Ys
member(X, [X|Ys]).
member(X, [Y|Ys]) :- member(X, Ys).
```

Note that in the first clause we have omitted the check whether `Ys` is a proper list, making it part of the presupposition that the second argument to `member` is a list.

Already, this very simple predicate has some subtle points. To show the examples, we use the Prolog notation `?-` A`.` for running a query A. After presenting the first answer substitution, Prolog interpreters issue a prompt to see if another solution is desired. If the user types `';'` the interpreter will backtrack to see if another solution can be found. For example, the query

```
?- member(X, [a,b,a,c]).
```

has four solutions, in the order

```
X = a;
X = b;
X = a;
X = c.
```

Perhaps surprisingly, the query

[1] GNU Prolog has the special syntax `G=..[P, X], call(G)` for meta-call `P(X)`.

```
?- member(a, [a,b,a,c]).
```

succeeds twice (both with the empty substitution), once for the first occurrence of `a` and once for the second occurrence.

If `member` is part of a larger program, backtracking of a later goal could lead to unexpected surprises when `member` succeeds again. There could also be an efficiency issue. Assume you keep the list in alphabetical order. Then when we find the first matching element there is no need to traverse the remainder of the list, although the `member` predicate above will always do so.

So what do we do if we want to only check membership, or find the first occurrence of an element in a list? Unfortunately, there is no easy answer, because the most straighforward solution

$$\frac{}{\mathsf{member}(X, \mathsf{cons}(X, Ys))} \qquad \frac{X \neq Y \quad \mathsf{member}(X, Ys)}{\mathsf{member}(X, \mathsf{cons}(Y, Ys))}$$

requires disequality which is problematic in the presence of variables. In Prolog notation:

```
member1(X, [X|Ys]).
member1(X, [Y|Ys]) :- X \= Y, member1(X, Ys).
```

When both arguments are ground (i.e., have no variables), this works as expected, giving just one solution to the query

```
?- member1(a, [a,b,a,c]).
```

However, when we ask

```
?- member1(X, [a,b,a,c]).
```

we only get one answer, namely `X = a`. The reason is that when we come to the second clause, we instantiate `Y` to `a` and `Ys` to `[b,a,c]`, and the body of the clause becomes

```
X \= a, member1(X, [b,a,c]).
```

Now we have the problem that we cannot determine if `X` is different from `a`, because `X` is still a variable. Prolog interprets $s \neq t$ as *non-unifiability*, that is, $s \neq t$ succeeds if s and t are not unifiable. But `X` and `a` are unifiable, so the subgoal fails and no further solutions are generated.[2]

[2]One must remember, however, that in Prolog unification is not sound because it omits the occurs-check, as hinted at in the previous lecture. This also affects the correctness of disequality, because the result does not have to be correct.

There are two attitudes we can take to handle disequalities. One is to restrict the use of disequality (and, therefore, here also the use of `member1`) to the case where both sides are ground so have no variables in them. In that case disequality can be easily checked without problems. This is the suggested solution adopted by Prolog programmers, and one which we adopt for now.

The second one is to postpone the disequality $s \neq t$ until we can tell from the structure of s and t that they will be different (in which case we succeed) or the same (in which case the disequality fails). The latter solution requires a much more complicated operational semantics because some goals must be postponed until their arguments become instantiated. This is the general topic of *constructive negation*[3] [1] in the setting of *constraint logic programming* [2, 6].

Disequality is related to the more general question of negation, because $s \neq t$ is the negation of equality, which is a simple predicate that is either primitive, or could be defined with the one clause `X = X`.

5 Simple List Predicates

Now let's explore some other list operations and explore recursive definitions and their flexibility in more detail. We start with $\mathsf{prefix}(xs, ys)$ which is supposed to hold when the list xs is a prefix of the list ys. The definition is relatively straightforward.

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

Conversely, we can test for a suffix.

```
suffix(Xs, Xs).
suffix(Xs, [Y|Ys]) :- suffix(Xs, Ys).
```

Interestingly, these predicates can be used in a variety of ways. We can check if one list is a prefix of another, we can enumerate prefixes, and we can even enumerate prefixes and lists. For example:

[3] The use of the word "constructive" here is unrelated to its use in logic.

```
?- prefix(Xs,[a,b,c,d]).
Xs = [];
Xs = [a];
Xs = [a,b];
Xs = [a,b,c];
Xs = [a,b,c,d].
```

enumerates all prefixes, while

```
?- prefix(Xs,Ys).
Xs = [];

Xs = [A]
Ys = [A|_];

Xs = [A,B]
Ys = [A,B|_];

Xs = [A,B,C]
Ys = [A,B,C|_];

Xs = [A,B,C,D]
Ys = [A,B,C,D|_];
...
```

enumerates lists together with prefixes. Note that `A`, `B`, `C`, and `D` are variables, as is the underscore _ (called anonymous variable) so that for example `[A|_]` represents any list with at least one element.

A more general prediate is append(xs, ys, zs) which holds when zs is the result of appending xs and ys.

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

append can also be used in different directions, and we can also employ it for alternative definitions of prefix and suffix.

```
prefix2(Xs, Ys) :- append(Xs, _, Ys).
suffix2(Xs, Ys) :- append(_, Xs, Ys).
```

Here we have used anonymous variables '_'. The prefix2 clause specifies that Xs is a prefix of Ys if appending something (represented by the anonymous variable _) to Xs gives Ys. Conversely, suffix clause specifies that Xs

is a suffix of Ys if there is something (represented by _) to which appending Xs gives Ys.

Note that when several underscores appear in one clauses, each one stands for a different anonymous variable. For example, if we want to define a sublist as a suffix of a prefix, we have to name the intermediate variable instead of leaving it anonymous.

```
sublist(Xs, Ys) :- prefix(Ps, Ys), suffix(Xs, Ps).
```

A sublist of Ys is any suffix Xs of any prefix Ps. A definition with anonymous variables would be incorrect:

```
%% THIS IS INCORRECT CODE
sublistBug(Xs, Ys) :- prefix(_, Ys), suffix(Xs, _).
```

6 Sorting

As a slightly larger example, we use a recursive definition of quicksort. This is particularly instructive as it clarifies the difference between a specification and an implemention. A specification for sort(xs, ys) would simply say that ys is an ordered permutation of xs. However, this specification is not useful as an implementation: we do not want to cycle through all possible permutations until we find one that is ordered.

Instead we implement a non-destructive version of quicksort, modeled after similar implementations in functional programming. We use here the built-in Prolog integers, rather than the unary representation from the previous lecture. Prolog integers can be compared with n =< m (n is less or equal to m) and n > m (n is greater than m) and similar predicates, written in infix notation. In order for these comparisons to make sense, the arguments must be instantiated to actual integers and are not allowed to be variables, which constitute a run-time error. This combines two conditions: the first, which is called a *mode*, is that =< and < require their arguments to be *ground* upon invocation, that is not contain any variables. The second condition is a type condition which requires the arguments to be integers. Since these conditions cannot be enforced at compile time, they are signaled as run-time errors.

Quicksort proceeds by partitioning the tail of the input list into those elements that are smaller than or equal to its first element and those that are larger than its first element. It then recursively sorts the two sublists and appends the results.

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

Partitioning a list about the pivot element `X0` into the list Ls of elements less-or-equal `X0` and the list Gs of elements greater than `X0` is also straightforward.

```
partition([], _, [], []).
partition([X|Xs], X0, [X|Ls], Gs) :-
        X =< X0, partition(Xs, X0, Ls, Gs).
partition([X|Xs], X0, Ls, [X|Gs]) :-
        X > X0, partition(Xs, X0, Ls, Gs).
```

Note that the second and third case are both guarded by comparisons. This will fail if either `X` or `X0` are uninstantiated or not integers. The predicate `partition(`xs`, `x_0`, `ls`, `gs`)` therefore inherits a mode and type restriction: the first argument must be a ground list of integers and the second argument must be a ground integer. If these conditions are satisfied and `partition` succeeds, the last two arguments will always be lists of ground integers. In a future lecture we will discuss how to enforce conditions of this kind to discover bugs early. Here, the program is small, so we can get by without mode checking and type checking.

It may seem that the check `X > X0` in the last clause is redundant. However, that is not the case because upon backtracking we might select the second clause, even if the first one succeeded earlier, leading to an incorrect result. For example, without this guard the query

```
?- quicksort([2,1,3], Ys)
```

would incorrectly return `Ys = [2,1,3]` as its second solution.

In this particular case, the test is trivial so the overhead is acceptable. Sometimes, however, a clause is guarded by a complicated test which takes a long time to evaluate. In that case, there is no easy way to avoid evaluating it twice, in pure logic programming. Prolog offers several ways to work around this limitation which we discuss in the next section.

Finally note the significant performance advantage of the order of subgoals chosen in `partition`. If the comparisons with `X0` and the recursive call to `partition` were swapped, the subgoal order from left to right

would cause substantial computation for partition that might be discarded if the subsequent comparison fails.

7 Conditionals

We use the example of computing the minimum of two numbers as an example analogous to `partition`, but shorter.

```
minimum(X, Y, X) :- X =< Y.
minimum(X, Y, Y) :- X > Y.
```

In order to avoid the second, redundant test we can use Prolog's *conditional* construct, written as

```
A -> B ; C
```

which solves goal A. If A succeeds we commit to the solution, removing all choice points created during the search for a proof of A and then solve B. If A fails we solve C instead. There is also a short form `A -> B` which is equivalent to `A -> B ; fail` where `fail` is a goal that always fails.

Using the conditional, `minimum` can be rewritten more succinctly as

```
minimum(X, Y, Z) :- X =< Y -> Z = X ; Z = Y.
```

The price that we pay here is that we have to leave the realm of pure logic programming.

Because the conditional is so familiar from imperative and functional program, there may be a tendency to overuse the conditional when it can easily be avoided.

8 Cut

The conditional combines two ideas: commiting to all choices so that only the first solution to a goal will be considered, and branching based on that first solution.

A more powerful primitive is *cut*, written as '`!`', which is unrelated to the use of the word "cut" in proof theory. A cut appears in a goal position and commits to all choices that have been made since the clause it appears in has been selected, including the choice of that clause. For example, the following is a correct implementation of `minimum` in Prolog.

```
minimum(X, Y, Z) :- X =< Y, !, Z = X.
minimum(X, Y, Y).
```

The first clause states that if x is less or equal to y then the minimum is equal to x. Moreover, we commit to this clause in the definition of minimum and on backtracking we do not attempt to use the second clause (which would otherwise be incorrect, of course).

If we permit meta-calls in clauses, then we can define the conditional `A -> B ; C` using cut with

```
if_then_else(A, B, C) :- A, !, B.
if_then_else(A, B, C) :- C.
```

The use of cut in the first clause removes all choice points created during the search for a proof of A when it succeeds for the first time, and also commits to the first clause of `if_then_else`. The solution of B will create choice points and backtrack as usual, except when it fails the second clause of `if_then_else` will never be tried.

If A fails before the cut, then the second clause will be tried (we haven't committed to the first one) and C will be executed.

Cuts can be very tricky and are the source of many errors, because their interpretation depends so much on the operational behavior of the program rather than the logical reading of the program. One should resist the temptation to use cuts excessively to improve the efficiency of the program unless it is truly necessary.

Cuts are generally divided into *green cuts* and *red cuts*. Green cuts are merely for efficiency, to remove redundant choice points, while red cuts change the meaning of the program entirely. Revisiting the earlier code for `minimum` we see that it is a red cut, since the second clause does not make any sense by itself, but only because the first clause was attempted before and must have failed its `X =< Y` test to ever proceed to the second clause. The cut in

```
minimum(X, Y, Z) :- X =< Y, !, Z = X.
minimum(X, Y, Y) :- X > Y.
```

is a green cut: removing the cut does not change the meaning of the program. It still serves some purpose here, however, because it prevents the second comparison to be carried out if the first one succeeds (although it is still performed redundantly if the first one fails).

A common error is exemplified by the following attempt to make the `minimum` predicate more efficient.

```
% THIS IS INCORRECT CODE
minimumBug(X, Y, X) :- X =< Y, !.
minimumBug(X, Y, Y).
```

At first this seems completely plausible, but it is nonetheless incorrect. Think about it before you look at the counterexample at the end of these notes—it is quite instructive.

9 Negation as Failure

One particularly interesting use of cut is to implement negation as finite failure. That is, we say that A is false if the goal A fails. Using higher-order techniques and we can implement `\+(A)`, read negation-as-failure of A, with

```
\+(A) :- A, !, fail.
\+(A).
```

The second clause seems completely contrary to the definition of negation, so we have to interpret this program operationally. To solve `\+(`A`)` we first try to solve A. If that fails we go the second clause which always succeeds. This means that if A fails then `\+(`A`)` will succeed without instantiating any variables. If A succeeds then we commit and fail, so the second clause will never be tried. In this case, too, no variables are instantiated, this time because the goal fails.

One of the significant problem with negation as failure is the treatment of variables in the goal. That is, `\+(`A`)` succeeds if there is no instance of A that is true. On the other hand, it fails if there is an instance of A that succeeds. This means that free variables may not behave as expected. For example, the goal

```
?- \+(X = a).
```

will fail. According to the usual interpretation of free variables this would mean that there is no term t such that $t \neq a$ for the constant a. Clearly, this interpretation is incorrect, as, for example,

```
?- \+(b = a).
```

will succeed.

This problem is similar to the issue we identified for disequality. When goals may not be ground, negation as failure should be viewed with distrust and is probably wrong more often than it is right.

There is also the question on how to reason about logic programs containing disequality, negation as failure, or cut. I do not consider this to be a solved research question.

10 Prolog Arithmetic

As mentioned and exploited above, integers are a built-in data type in Prolog with some predefined predicates such as =< or >. You should consult your Prolog manual for other built-in predicates. There are also some built-in operations such as addition, subtraction, multiplication, and division. Generally these operations can be executed using a special goal of the form t `is` e which evaluates the arithmetic expression e and unifies the result with term t. If e cannot be evaluated to a number, a run-time error will result. As an example, here is the definition of the length predicate for Prolog using built-in integers.

```
% length(Xs, N) iff Xs is a list of length N.
length([], 0).
length([X|Xs], N) :- length(Xs, N1), N is N1+1.
```

As is often the case, the left-hand side of the `is` predicate is a variable, the right-hand side an expression.

11 Answer

The problem with `minimumBug` is that a query such as

```
?- minimumBug(5,10,10).
```

will succeed because it fails to match the first clause head due to its pattern of variables and then accepts the second clause which has no subgoals. Contrast this to the previous, correct, implementation of `minimum/3` which first accepted any input pattern and then subsequently unified after the cut to check whether or make sure that the arguments match.

The general rule of thumb is to leave output variables (here: in the third position) unconstrained free variables and unify it with the desired output after the cut. This leads to the earlier version of `minimum` using cut.

12 References

References

[1] D. Chan. Constructive negation based on the complete database. In R.A. Kowalski and K.A. Bowen, editors, *Proceedings of the 5th International Conference and Symposium on Logic Programming (ICSLP'88)*, pages 111–125, Seattle, Washington, September 1988. MIT Press.

[2] Joxan Jaffar and Jean-Louis Lassez. Constraint logic programming. In *Proceedings of the 14th Annual Symposium on Principles of Programming Languages*, pages 111–119, Munich, Germany, January 1987. ACM Press.

[3] Dale Miller and Gopalan Nadathur. Higher-order logic programming. In Ehud Shapiro, editor, *Proceedings of the Third International Logic Programming Conference*, pages 448–462, London, June 1986.

[4] Gopalan Nadathur and Dale Miller. Higher-order logic programming. In D.M. Gabbay, C.J. Hogger, and J.A. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 5, chapter 8. Oxford University Press, 1998.

[5] Frank Pfenning, editor. *Types in Logic Programming*. MIT Press, Cambridge, Massachusetts, 1992.

[6] Peter J. Stuckey. Constructive negation for constraint logic programming. In *Proceedings of the 6th Annual Symposium on Logic in Computer Science (LICS'91)*, pages 328–339, Amsterdam, The Netherlands, July 1991. IEEE Computer Society Press.