Lecture Notes on
Prolog

15-317: Constructive Logic
Frank Pfenning*

Lecture 15
April 13, 2021

1 Introduction

Prolog is the first and still standard backward-chaining logic programming language. While it is somewhat dated, it contains many interesting ideas and is appropriate for certain types of applications that involve symbolic representations, backtracking, or unification. On the other, it has more than its typical share of pitfalls and problems, due to its dynamically typed nature, the prevalence of failure and backtracking, and the interactions between logical and extralogical predicates.

In this lecture we give a somewhat nonstandard introduction to Prolog by introducing a number of critical features using two examples: (1) basic computation on binary numbers in little endian representation, and (2) a proof checker based on certifying theorem provers Lecture 13.

2 Binary Numbers

Unary numbers, such as used in the Peano's axioms, are foundationally adequate, but not useful for practical computation due to the size of their representation. Instead, we use binary numbers. Representing them as *terms* in logic is straightforward, we just have to decide on the particulars[1].

*Edits by André Platzer

[1]Practical logic programming languages such as Prolog use machine arithmetic instead.

It turns out a so-called "little endian" representation where the least significant bits is the outermost constructor is most convenient. This is because when defining operations on two numbers the least significant bits of both numbers always line up correctly, and then we can recurse on the remainder of the numbers representing the higher bits.

$$\text{Binary numbers} \quad M \quad ::= \quad \mathsf{b0}(M) \mid \mathsf{b1}(M) \mid \mathsf{e}$$

where the (mathematical) translations between (mathematical) values and their representations are given by functions $\ulcorner\urcorner$ and $\llcorner\lrcorner$, respectively:

$$\begin{array}{lcl@{\qquad}lcl}
\ulcorner 0 \urcorner & = & \mathsf{e} & \llcorner \mathsf{e} \lrcorner & = & 0 \\
\ulcorner 2n \urcorner & = & \mathsf{b0}(\ulcorner n \urcorner) \text{ for } n > 0 & \llcorner \mathsf{b0}(M) \lrcorner & = & 2\,\llcorner M \lrcorner \\
\ulcorner 2n+1 \urcorner & = & \mathsf{b1}(\ulcorner n \urcorner) & \llcorner \mathsf{b1}(M) \lrcorner & = & 2\,\llcorner M \lrcorner + 1
\end{array}$$

Now we can specify the successor relation $\mathsf{inc}(M, N)$ such that N represents the successor of M by the following three rules:

$$\frac{}{\mathsf{inc}(\mathsf{e}, \mathsf{b1}(\mathsf{e}))}\ \mathsf{inc}_e \qquad \frac{}{\mathsf{inc}(\mathsf{b0}(M), \mathsf{b1}(M))}\ \mathsf{inc}_0 \qquad \frac{\mathsf{inc}(M, N)}{\mathsf{inc}(\mathsf{b1}(M), \mathsf{b0}(N))}\ \mathsf{inc}_1$$

In these rule, we use upper case variables for schematic variables in the rules, which is consistent with the Prolog syntax. We use lower case identifiers for predicates (inc, so far), function symbols (b0 and b1), including arity 0 function symbols or constants (e). In Prolog syntax, we write a rule

$$\frac{J_1 \ldots J_n}{J}$$

with :- as separator between the conclusion and its resulting premises:

$$J \ \texttt{:-}\ J_1, \ldots, J_n.$$

which we read as "*J if J_1 through J_n*". We call the rule a *clause*, J the *head of the clause* and $J_1, \ldots, J_n$ the *body of the clause*. If there are zero premises, the rules is simply written as '*J.*'. Transcribing the rules then yields the following Prolog program. We call the predicate `inc_` (with a trailing underscore) to distinguish it from the later predicate `inc` which fixes some of its problems.

```
inc_(e,b1(e)).
inc_(b0(M),b1(M)).
inc_(b1(M),b0(N)) :- inc_(M,N).
```

As we will see, there are some problems with this program. But first, let's fire up the Gnu Prolog interpreter to run this program on some inputs. The first line after the prompt `| ?-` is Prolog's notation for loading a program from a file, here `bin.pl`.

```
% gprolog
GNU Prolog 1.4.4 (64 bits)
Compiled Apr 23 2013, 17:26:17 with /opt/local/bin/gcc-apple-4.2
By Daniel Diaz
Copyright (C) 1999-2013 Daniel Diaz
| ?- ['bin.pl'].
...
(1 ms) yes
| ?-
```

At the prompt we can now type *queries* and have the interpreter simultaneously search for a proof and an instantiation of the free variables in the query. Once a solution has been found, the interpreter may give you the opportunity to search for other solutions, or simply return to the prompt if it can see no further solutions are possible. For example, we can increment 5 to get 6.

```
| ?- inc_(b1(b0(b1(e))),N).

N = b0(b1(b1(e)))

yes
| ?-
```

We can also give several goals conjunctively, and they will be solved in sequence. The following increments 5 three times to obtain 8.

```
| ?- inc_(b1(b0(b1(e))),N1), inc_(N1,N2), inc_(N2,N3).

N1 = b0(b1(b1(e)))
N2 = b1(b1(b1(e)))
N3 = b0(b0(b0(b1(e))))

yes
| ?-
```

The fact that Prolog is dynamically typed, leads to some unexpected and meaningless answers:

```
| ?- inc_(b0(some_random_junk), N).

N = b1(some_random_junk)
```

This comes under the heading of "*garbage-in, garbage-out*", but it is still disconcerting this would be claimed as true rather than meaningless!

Now let's try to run the predicate "in reverse" to calculate the predecessor of a binary number, in this case 6.

```
| ?- inc_(M,b0(b1(b1(e)))).

M = b1(b0(b1(e))) ? ;

no
| ?-
```

Indeed, we obtain 5! Prolog asks if we would like to search for another solution, we type the semicolon verb';' and it confirms there are no further solutions. So far, things look good. Let't try the predecessor of 0, which should not exist.

```
| ?- inc_(M,e).

no
| ?-
```

Once again correct. Let's try the predecessor of 1:

```
| ?- inc_(M,b1(e)).

M = e ? ;

M = b0(e) ? ;

no
| ?-
```

Here we have a surprise: we get two answers! The second one also morally represents the number $2 \cdot 0 = 0$, but it is not in standard form since it has a leading bit b0. The problem is that both the first and second rules apply to this query

```
inc_(e,b1(e)).
inc_(b0(M),b1(M)).
inc_(b1(M),b0(N)) :- inc_(M,N).
```

Returning an answer not in standard form is a problem only if we want to always maintain standard form (which seems like a good idea). But even if we do not, the fact that the innocuous looking predicate returns a second answer upon backtracking will almost certainly lead to unintended consequences wherever this predicate is used.

If we want to run this predicate with the mode `inc(-N, +M)` then we need to prevent the second solution by distinguishing the cases for M in the middle clause.

```
inc(e,b1(e)).
inc(b0(b0(M)),b1(b0(M))).
inc(b0(b1(M)),b1(b1(M))).
inc(b1(M),b0(N)) :- inc(M,N).
```

Now b0(e) is ruled out, and the problem disappears:

```
| ?- inc(M,b1(e)).

M = e ? ;

no
| ?-
```

Here is one way to define what it means for a binary to be in standard form

```
std(e).
% no case for std(b0(e))
std(b0(b0(N))) :- std(b0(N)).
std(b0(b1(N))) :- std(b1(N)).
std(b1(N)) :- std(N).
```

Next lecture shows another way. Now `inc(+std,-std)` and `inc(-std,+std)`.

3 Checking Proof Terms

We now move on to another example, which introduces a number of other features of Prolog: proof checking. The rules were introduced in Lecture 13 and we only summarize the fragment with implication and conjunction:

Checkable terms	M, N	$::=$	$\langle M, N\rangle \mid (\mathsf{fn}\ u \Rightarrow M) \mid R$
Synthesizing terms	R	$::=$	$u \mid \textbf{fst}\,R \mid \textbf{snd}\,R \mid R\,M$
Ordered contexts	Ω	$::=$	$. \mid (u : A\downarrow) \cdot \Omega$

We use the (list) *ordered* context so we can check terms such as

$$\vdash (\mathsf{fn}\ x \Rightarrow \mathsf{fn}\ x \Rightarrow x) : a \supset (b \supset b)$$

where x refers to the innermost binding of x, but not to any other ones:

$$\not\vdash (\mathsf{fn}\ x \Rightarrow \mathsf{fn}\ x \Rightarrow x) : a \supset (b \supset a)$$

We have the following rules.

$$\frac{\Omega \vdash M : A \uparrow \quad \Omega \vdash N : B \uparrow}{\Omega \vdash \langle M, N\rangle : A \wedge B \uparrow} \wedge I \qquad \frac{(u : A \downarrow) \cdot \Omega \vdash M : B \uparrow}{\Omega \vdash (\mathsf{fn}\ u \Rightarrow M) : A \supset B \uparrow} \supset I^u$$

$$\frac{\Omega \vdash R : A \downarrow}{\Omega \vdash R : A \uparrow} \downarrow\uparrow$$

$$\frac{}{(u : A \downarrow) \cdot \Omega \vdash u : A \downarrow} \mathsf{var}_{=} \qquad \frac{w \neq u \quad \Omega \vdash u : A \downarrow}{(w : B \downarrow) \cdot \Omega \vdash u : A \downarrow} \mathsf{var}_{\neq}$$

$$\frac{\Omega \vdash R : A \wedge B \downarrow}{\Omega \vdash \mathbf{fst}\, R : A \downarrow} \wedge E_1 \qquad \frac{\Omega \vdash R : A \wedge B \downarrow}{\Omega \vdash \mathbf{snd}\, R : B \downarrow} \wedge E_2$$

$$\frac{\Omega \vdash R : A \supset B \downarrow \quad \Omega \vdash M : A \uparrow}{\Omega \vdash R\, M : B \downarrow} \supset E$$

We had the intuition that these rules describe an algorithm, but now they (almost) really do describe a program, in Prolog! We have two predicates, one (`check/3`, which means `check` with 3 arguments) in which Ω, M, and A must all be given and it succeeds or fails, and `synth/3` in which Ω and R are given and it synthesizes A or fails (where Ω is rendered as O).

```
check(+O, +M, +A)
synth(+O, +R, -A)
```

The first rule is easy to translate, using the constructors `pair(M,N)` for $\langle M, N\rangle$ and `and(A,B)` for $A \wedge B$:

```
check(O, pair(M,N), and(A,B)) :-
    check(O, M, A),
    check(O, N, B).
```

In the second rule, $\supset I$, we have to add a variable and its type to the front of the list Ω. The syntax for lists in Prolog allows several different notations. We have the empty list `[]`, and we have a constructor which takes an element `X` and adds it to the front of the list `Xs` with `'.'(X,Xs)` Here, use of "dot" requires single quotes so it is not confused with the end-of-clause period. An alternative notation for `'.'(X,Xs)` is `[X | Xs]`, which is commonly used. Finally we can also write out lists as with `[1,2,3,4]`. All of the following denote the same list:

```
[1,2,3,4]   [1|[2,3,4]]   [1,2|[3,4]]
'.'(1,[2,3,4]) '.'(1,'.'(2,'.'(3,'.'(4,[]))))
[1|[2|[3|[4|[]]]]]
```

We use the most common syntax to add the term `tp(X,A)` to the front of the context `O`, where `tp/2` is a new Prolog term constructor.

```
check(O, fun(X,M), imp(A,B)) :-
    check([tp(X,A)|O], M, B).
```

If neither of these two clauses match, we could be looking at a synthesizable term R, so we should try to synthesize a type for it and compare it to the given one.

```
check(O, R, A) :- synth(O, R, B), A = B.
```

Here we use the built-in equality predicate to *unify* A and B. In this case, for a mode-correct query, A is given and input to `check` and B will be returned by `synth`, so the comparison is just an equality test, not full unification.

The synthesis judgment is again straightforward for pairs.

```
synth(O, fst(R), A) :- synth(O, R, and(A,B)).
synth(O, snd(R), B) :- synth(O, R, and(A,B)).
```

We can see this is mode-correct for `synth(+, +, -)`. In the head of the clause, we know `O` and `fst(R)` and therfore `R`. Now we can appeal to the hypothesis that `and(A,B)` will be known if the subgoal succeeds, which means `A` is known, which is what we needed to show.

In the case of application `app(R,M)` we just need to be careful to solve the two subgoals in the right order.

```
synth(O, app(R,M), B) :-
    synth(O, R, imp(A,B)),
    check(O, M, A).
```

If we had switched them, as in

```
synth(O, app(R,M), B) :-
    check(O, M, A),        % bug here!
    synth(O, R, imp(A,B)).
```

then it would not be mode correct: when we call `check(O, M, A)` we do not yet know `A`, which is required for the mode of `check/3`.

Finally we have two rules for variables, where we either find `X` at the head of the list, or look for it in the tail.

```
% warning: these have a bug!
synth([tp(X,A)|O], X, A).
synth([tp(Y,B)|O], X, A) :- synth(O, X, A).
```

Let's run some examples with this code to test it. The first one checks the identity function, then we check the first and second projections. All succeed and fail as expected.

```
| ?- check([], fun(x,x), imp(a,a)).

true ? ;

no
| ?- check([], fun(x,fun(y,x)), imp(a,imp(b,a))).

true ? ;

no
| ?- check([], fun(x,fun(y,y)), imp(a,imp(b,b))).

true ? ;

no
| ?- check([], fun(x,fun(y,y)), imp(a,imp(b,a))).

no
| ?-
```

However, there is a bug in the program. Consider

```
| ?- check([], fun(x,fun(x,x)), imp(a,imp(b,b))).

true ? ;
```

```
no
| ?- check([], fun(x,fun(x,x)), imp(a,imp(b,a))).

true ? ;

no
| ?-
```

The first query succeeds as expected, because `x` should refer to its innermost enclosing binder. The second query shows that we incorrectly allow `x` to also refer to the outer binder, violating all scoping principles!

```
% warning: these have a bug!
synth([tp(X,A)|O], X, A).
synth([tp(Y,B)|O], X, A) :- synth(O, X, A).
```

If we trace the execution we can see the problem

```
| ?- trace.
The debugger will first creep -- showing everything (trace)

yes
{trace}
| ?- check([], fun(x,fun(x,x)), imp(a,imp(b,a))).
      1    1  Call: check([],fun(x,fun(x,x)),imp(a,imp(b,a))) ?
      2    2  Call: check([tp(x,a)],fun(x,x),imp(b,a)) ?
      3    3  Call: check([tp(x,b),tp(x,a)],x,a) ?
      4    4  Call: synth([tp(x,b),tp(x,a)],x,_416) ?
      4    4  Exit: synth([tp(x,b),tp(x,a)],x,b) ?
      4    4  Redo: synth([tp(x,b),tp(x,a)],x,b) ?
      5    5  Call: synth([tp(x,a)],x,_441) ?
      5    5  Exit: synth([tp(x,a)],x,a) ?
      4    4  Exit: synth([tp(x,b),tp(x,a)],x,a) ?
      3    3  Exit: check([tp(x,b),tp(x,a)],x,a) ?
      2    2  Exit: check([tp(x,a)],fun(x,x),imp(b,a)) ?
      1    1  Exit: check([],fun(x,fun(x,x)),imp(a,imp(b,a))) ?

true ?
```

Trace line 5 exits the query `synth([tp(x,b),tp(x,a)],x,_416) ?` with `_416 = b`. However, this fails to unify with the `a` we are supposed to check against in line 3! So Prolog backtracks (see `Redo`) and synthesizes another type from the remainder of the context `O`, namely `a`. This now works and the query incorrectly succeeds.

The fix requires that we *only* continue to look through the context Ω if the variable x we are trying to find is *different* from the variable at the head of the list:

```
synth([tp(X,A)|O], X, A).
synth([tp(Y,B)|O], X, A) :- Y \= X, synth(O, X, A).
```

The new goal `Y \= X` stands for "`Y` *and* `X` *are not unifiable*", although here both `Y` and `X` will be known and it just comes down to checking equality between two terms.

Here is the summary of the repaired program

```
% check(+O, +M, +A)
% synth(+O, +R, -A)

check(O, pair(M,N), and(A,B)) :-
    check(O, M, A),
    check(O, N, B).
check(O, fun(X,M), imp(A,B)) :-
    check([tp(X,A)|O], M, B).
check(O, R, A) :- synth(O, R, B), A = B.

synth(O, fst(M), A) :- synth(O, M, and(A,B)).
synth(O, snd(M), B) :- synth(O, M, and(A,B)).
synth(O, app(R,M), B) :-
    synth(O, R, imp(A,B)),
    check(O, M, A).
synth([tp(X,A)|O], X, A).
synth([tp(Y,B)|O], X, A) :- Y \= X, synth(O, X, A).
```

Let's run a few more queries. First the uncurrying function, which proves $(A \supset (B \supset C)) \supset ((A \wedge B) \supset C)$ and then currying, which goes in the opposite direction.

```
| ?- check([], fun(f,fun(p,app(app(f,fst(p)),snd(p)))),
               imp(imp(a,imp(b,c)),imp(and(a,b),c))).

true ? ;

no
| ?- check([], fun(f,fun(x,fun(y,app(f,pair(x,y))))),
               imp(imp(and(a,b),c),imp(a,imp(b,c)))).

true ? ;

(1 ms) no
```

4 Unification

This has worked out extremely well so far, so now we are getting greedy. What about running `check/3` in a mode such as `check(+O, +M, -A)` that it was not intended for, which amounts to type inference? Let's try!

```
| ?- check([], fun(x,x), A).

A = imp(B,B) ? ;

no
```

The interpreter finds one solution, namely $B \supset B$ for any B. In type inference we can see this is *the most general type* since any other type of $\mathsf{fn}\ x \Rightarrow x$ is an instance of it. Let's try something more complicated, such as $\mathsf{fn}\ f \Rightarrow \mathsf{fn}\ x \Rightarrow \mathsf{fn}\ y \Rightarrow f\,(x, y)$:

```
| ?- check([], fun(f,fun(x,fun(y,app(f,pair(x,y))))), A).

A = imp(imp(and(B,C),D),imp(B,imp(C,D))) ? ;

no
```

Again, the interpreter finds a single most general solution, namely $((B \wedge C) \supset D) \supset (B \supset (C \supset D))$ for any propositions (types) B, C, and D. During the search for a proof of the goal, the interpreter accumulates constraints on A. These are then solved by a process called *unification*, which fails if there are no solutions or simplifies the constraints to a minimal form where it is easy to read off the solution in the form of a substitution.

Unfortunately, Prolog's algorithm for unification is *unsound* for reasons of efficiency. This is really inexcusable, especially since the overhead together with other optimizations is not very high, but we now have to live with that bad decision. To see the problem consider the term $\mathsf{fn}\ x \Rightarrow x\,x$. There should not be a type for this term, because after a couple of steps we are in the situation (writing unknown types as greek letters):

$$\dfrac{\dfrac{}{(x:\alpha\downarrow)\vdash x:\alpha\downarrow}\,x \qquad \dfrac{\dfrac{}{x:\alpha\downarrow\vdash x:\alpha\downarrow}\,x \qquad \alpha=\beta}{x:\alpha\downarrow\vdash x:\beta\uparrow}\,{\downarrow\uparrow} \qquad \alpha=\beta\supset\gamma}{x:\alpha\downarrow\vdash x\,x:\gamma\downarrow}\,{\supset}E$$

The reason that type inference fails is that there is no solution to the equations $\alpha = \beta, \alpha = \beta \supset \gamma$ because $\beta = \beta \supset \gamma$ has no solution.

Surprisingly, Prolog fails to notice that! Or, more precisely, it builds a cyclic term to "solve" this equation.

```
check([], fun(x, app(x,x)), A).

cannot display cyclic term for A ? ;

no
```

Essentially, when it processes the equation $\beta = B$ for some type B it just sets β to be equal to B without checking if this would introduce a cyclic term. In Prolog terminology we say that it *omits the the occurs check* which would verify that B does not contain the β that is chosen as the value for β.

Because of this shortcoming, Prolog has an explicit predicate `unify_with_occurs_check/2` that makes sure the two arguments are unified properly, so that the problem $\beta = \beta \supset \gamma$ fails. Wherever in the program a variable is repeated in the head of a clause, or we call on unification, we should call on unification with the occurs check instead. Fortunately, we only have to consider three lines. First

```
check(O, R, A) :- synth(O, R, B), A = B.
```

Now that we have a more general mode (allowing the third argument to be partially instantiated, but contain free variables), we can rewrite it as

```
check(O, R, A) :- synth(O, R, A).
```

Second, we look at the two lines where variables are considered

```
synth([tp(X,A)|O], X, A).
synth([tp(Y,B)|O], X, A) :- Y \= X, synth(O, X, A).
```

In the first line, the two occurrences of `A` are unified, which could be unsound so we need to appeal to `unify_with_occurs_check/2` instead. There is no such problem in the second line, because `B` and `A` are unrelated. Here is the complete revised program:

```
check(O, pair(M,N), and(A,B)) :-
    check(O, M, A),
    check(O, N, B).
check(O, fun(X,M), imp(A,B)) :-
    check([tp(X,A)|O], M, B).
check(O, R, A) :- synth(O, R, A).

synth(O, fst(M), A) :- synth(O, M, and(A,B)).
synth(O, snd(M), B) :- synth(O, M, and(A,B)).
synth(O, app(R,M), B) :-
    synth(O, R, imp(A,B)),
    check(O, M, A).
synth([tp(X,B)|O], X, A) :- unify_with_occurs_check(B, A).
synth([tp(Y,B)|O], X, A) :- Y \= X, synth(O, X, A).
```

Now we find that self-application is not typable, as expected.

```
| ?- check([], fun(x, app(x,x)), A).

no
```

while the other, positive examples continue to work as expected.

In a future lecture we will show the specifics about how unification in Prolog as well as sound unification works.

This particular example is a remarkable compact implementation of full type inference for a small language. Intrinsic pattern matching is critical, as it the built-in notion of logic variable and unification. Backtracking does not particularly come into play here, but it will if you implement the G4ip decision procedure based on Dyckhoff's contraction-free sequent calculus.

An attempt to use this proof checker as a theorem prover in the mode `check(+O, -M, +A)` predictably fails:

```
| ?- check([], M, imp(a,a)).

Fatal Error: local stack overflow ...
```