Lecture Notes on Logic Programming

15-317: Constructive Logic
Frank Pfenning

Lecture 18
Tuesday, March 28, 2023

1 Introduction

There are multiple roles for judgments and the inference rules defining them. For one, we used them to define fundamental notions such as *truth* and *verification*. For another, we used them to communicate basic algorithmic ideas such as bidirectional type-checking or decision procedures for intuitionistic propositional logic. In the latter case, we typically interpreted the rules bottom-up and extracted a functional implementation. More formally, this functional implementation was the computational contents of an intuitionistic proof *about* the judgment and its rules.

We now take this second role a step further: we define a programming language where construction of a derivation according to inference rules is the basic notion of computation. Proof construction must then adhere to a fixed strategy, otherwise the computational behavior of a program would be unpredictable. This echoes a similar relationship between proof reduction and steps of computation: the latter must follow a fixed strategy for predictable outcomes.

The class of languages operating via proof search are called *logic programming languages*. They can be roughly classified as either *backward chaining* (like Prolog [Kowalski, 1988, Colmerauer and Roussel, 1993]) or *forward chaining* (like Datalog [Maier et al., 2018]). These have quite different characteristics, and we examine them in turn. In this lecture we consider backward chaining and Prolog.

We now examine the various decisions on how to interpret a collection of inference rules as programs.

2 Goal-Directed Proof Construction

We use some very simple program to illustrate the fundamental concepts of how proof construction in a backward chaining logic programming language proceeds. Let's consider the judgment plus $x\ y\ z$ to express that $x + y = z$ on the unary natural numbers defined from 0 and the successor $\mathsf{s}(x)$ for a natural number x.

$$\frac{}{\mathsf{plus}\ 0\ y\ y}\ \mathsf{p0} \qquad \frac{\mathsf{plus}\ x\ y\ z}{\mathsf{plus}\ (\mathsf{s}\,x)\ y\ (\mathsf{s}\,z)}\ \mathsf{ps}$$

We can start by asking

$$?\!-\ \mathsf{plus}\ (\mathsf{s}\,0)\ (\mathsf{s}\,(\mathsf{s}\,0))\ (\mathsf{s}\,(\mathsf{s}\,(\mathsf{s}\,0))).$$

It will match this *query* against the conclusions of the rules in the order they are given. The query is *not* an instance of the $\mathsf{plus}\ 0\ y\ y$ because $0 \neq \mathsf{s}\,0$ so $\mathsf{p0}$ will fail immediately and it will try ps. The conclusion matches our goal under the *substitution* $x = 0$, $y = \mathsf{s}\,(\mathsf{s}\,0)$, and $z = \mathsf{s}\,(\mathsf{s}\,(\mathsf{s}\,0))$. So our partial derivation looks like

$$\frac{\begin{array}{c}\vdots\\ \mathsf{plus}\ 0\ (\mathsf{s}\,(\mathsf{s}\,0))\ (\mathsf{s}\,(\mathsf{s}\,0))\end{array}}{\mathsf{plus}\ (\mathsf{s}\,0)\ (\mathsf{s}\,(\mathsf{s}\,0))\ (\mathsf{s}\,(\mathsf{s}\,(\mathsf{s}\,0)))}\ \mathsf{ps}$$

Now rule $\mathsf{p0}$ applies with substitution $y = \mathsf{s}\,0$ and the derivation is complete.

$$\frac{\dfrac{}{\mathsf{plus}\ 0\ (\mathsf{s}\,(\mathsf{s}\,0))\ (\mathsf{s}\,(\mathsf{s}\,0))}\ \mathsf{p0}}{\mathsf{plus}\ (\mathsf{s}\,0)\ (\mathsf{s}\,(\mathsf{s}\,0))\ (\mathsf{s}\,(\mathsf{s}\,(\mathsf{s}\,0)))}\ \mathsf{ps}$$

A Prolog interpreter would not show this proof, but only say "yes" to indicate that the query is successful.

If we instead ask

$$?\!-\ \mathsf{plus}\ 0\ (\mathsf{s}\,0)\ 0.$$

the query would fail immediately because it does not match the conclusion of any rule. So the answer would be "no".

We can also compute answers beyond "yes" by leaving free variables in the goal. If the search is successful, the interpreter will report the substitution for this variable. Coming back to our original query, we can instead ask

$$?\!-\ \mathsf{plus}\ (\mathsf{s}\,0)\ (\mathsf{s}\,(\mathsf{s}\,0))\ Z.$$

which we interpret *existentially*: is there a term Z such that $\mathsf{plus}\ (\mathsf{s}\,0)\ (\mathsf{s}\,(\mathsf{s}\,0))\ Z$ can be derived by the rules? Again the first rule does not match, but the second does. While matching the conclusion of the rule against the goal, it *instantiates* variables, both in the query and the conclusion of the rule it tries to use.

$$\frac{\begin{array}{c}\vdots\\ \mathsf{plus}\ 0\ (\mathsf{s}\,(\mathsf{s}\,0))\ Z_2\end{array}}{\mathsf{plus}\ (\mathsf{s}\,0)\ (\mathsf{s}\,(\mathsf{s}\,0))\ Z}\ \mathsf{ps}\ (Z = \mathsf{s}\,Z_2)$$

Now it matches the first rule (p0) with the substitution $y = \mathsf{s}\,(\mathsf{s}\,0)$ and $Z_2 = \mathsf{s}\,(\mathsf{s}\,0)$. Since there is no premise there is no subgoal and we succeed.

$$\dfrac{\dfrac{}{\mathsf{plus}\ 0\ (\mathsf{s}\,(\mathsf{s}\,0))\ Z_2}\ \mathsf{p0}\ (Z_2 = \mathsf{s}\,(\mathsf{s}\,0))}{\mathsf{plus}\ (\mathsf{s}\,0)\ (\mathsf{s}\,(\mathsf{s}\,0))\ Z}\ \mathsf{ps}\ (Z = \mathsf{s}\,Z_2)$$

We elided the substitution for y since it doesn't appear in the original query. Now we see that $Z = \mathsf{s}\,Z_2$ and $Z_2 = \mathsf{s}\,(\mathsf{s}\,0)$ which works out to $Z = \mathsf{s}\,(\mathsf{s}\,(\mathsf{s}\,0))$. This is the answer returned overall in lieu of the full proof.

Logically, logic programming is generally based (on a fragment of) the predicate calculus, but one where elements t (satisfying the judgment t *elem* from our presentation of quantifiers) can be constructed from arbitrary constants (like 0) and function symbols (like s).

In particular, there is no static type-checking and this leads to some counterintuitive results. For example, the query

$$?\!-\ \mathsf{plus}\ 0\ \mathsf{junk}\ Z$$

will *succeed*(!) with answer substitution $Z = \mathsf{junk}$. Also, a query such as

$$?\!-\ \mathsf{plus}\ \mathsf{foo}\ 0\ 0$$

will simply fail with the result "no" rather than being rejected as meaningless.

As discussed in lecture, the unexpected success could be fixed. For example, we could write

$$\dfrac{}{\mathsf{nat}\ 0}\ \mathsf{n0} \qquad \dfrac{\mathsf{nat}\ x}{\mathsf{nat}\ (\mathsf{s}\,x)}\ \mathsf{ns}$$

$$\dfrac{\mathsf{nat}\ y}{\mathsf{plus}\ 0\ y\ y}\ \mathsf{p0} \qquad \dfrac{\mathsf{plus}\ x\ y\ z}{\mathsf{plus}\ (\mathsf{s}\,x)\ y\ (\mathsf{s}\,z)}\ \mathsf{ps}$$

However, this is not generally done because when the query has proper natural numbers, then this check is redundant and expensive. Also, in logic the idea is to *relativize the quantifiers* in order to simulate types with logical predicates. However, a query such as

$$?\!-\ \mathsf{nat}\ Z \wedge \mathsf{plus}\ t_1\ t_2\ Z$$

for terms t_1 and t_2 would have entirely the wrong behavior. As we will see shortly, it would enumerate the natural numbers until it finds one that is the sum of t_1 and t_2.

3 Backtracking

The rules in the program (called *clauses*) are tried in the order they are presented. This sometimes leads to unexpected results. Consider

$$\dfrac{p}{p} \qquad \dfrac{}{p}$$

If presented in this order, then

$$?\!\!-\ p.$$

will loop infinitely because it reduces the goal of proving p to the subgoal of proving p, etc. On the other hand, if we switch these two clauses then it will succeed infinitely often. This may be clearer using the nat predicate from the previous section:

$$?\!\!-\ \mathsf{nat}\ X.$$

will succeed with $X = 0$, then $X = \mathsf{s}\,0$ (if more solutions are needed), then $X = \mathsf{s}\,(\mathsf{s}\,0)$, etc.

Such backtracking behavior is occasionally quite cool, but often the programmer has to fight against over-generation of solutions.

Can we use plus in order to compute subtraction? Yes: a query such as $\mathsf{plus}\ n\ Y\ m$ will find a Y such that $n + Y = m$ if n and m are representations of concrete numbers. You may try this out in Prolog, or you can simulate it with goal-directed search as the other examples. Essentially, it will count down n to 0, meanwhile removing one successor from m on each step. When $n = 0$ then the third argument will be $n - m$ which is then copied to Y.

We can also split a natural number n into x and y such that $x + y = n$. Consider a query

$$?\!\!-\ \mathsf{plus}\ X\ Y\ (\mathsf{s}\,(\mathsf{s}\,0)).$$

Trying the rules in order, the first answer is $X = 0$ and $Y = \mathsf{s}\,(\mathsf{s}\,0)$. After backtracking, we get $X = \mathsf{s}\,0$ and $Y = \mathsf{s}\,0$, then $X = \mathsf{s}\,(\mathsf{s}\,0)$ and $Y = 0$. Further backtracking will fail because no more rules will match.

4 Subgoal Ordering

Let's consider a rules with more than one premises. Here are two for multiplication which uses the defining equation $(x + 1) \times y = x \times y + y$.

$$\frac{}{\mathsf{times}\ 0\ y\ 0}\ \mathsf{t0} \qquad \frac{\mathsf{times}\ x\ y\ w \quad \mathsf{plus}\ w\ y\ z}{\mathsf{times}\ (\mathsf{s}\,x)\ y\ z}\ \mathsf{ts}$$

Let's say we want to multiply $\mathsf{times}\ (\mathsf{s}\,0)\ (\mathsf{s}\,(\mathsf{s}\,0))\ Z$. If we construct the proof of the first premise first, it will solve and succeed with $W = 0$, and then find a proof of $\mathsf{plus}\ 0\ (\mathsf{s}\,(\mathsf{s}\,0))\ Z$ in a straightforward way.

However, if we tried to prove the second subgoal first, it would try to prove $\mathsf{plus}\ W\ (\mathsf{s}\,(\mathsf{s}\,0))\ Z$, essentially enumerating the numbers W and Z such that $W + 2 = Z$. For each such pair it will then check the multiplication in the first premise. This is clearly the wrong behavior in general, so it is important to keep subgoal ordering in mind.

Returning to an earlier example, the query

$$?\!\!-\ \mathsf{nat}\ Z \wedge \mathsf{plus}\ m\ n\ Z.$$

for unary numbers m and n will actually *enumerate* all possible natural numbers as Z and check until is finds one Z that satisfies $\mathsf{plus}\ m\ n\ Z$. Again, this is clearly not the right behavior (even if the answer may ultimately be correct).

So: *subgoals are ordered left to right*.

5 Unification

The process of solving the equations as they arise during search is called *unification*. We will discuss the details of an algorithm in a future lecture. You can think of the equations being solved by substitution, as we often do in algebra. When an answer substitution is presented, it is *projected* onto the free variables occurring in the query. Other variables represent intermediate constraints that arise during search that are not shown to the user.

Whenever a rule is being used, we instantiate its schematic variables with fresh variables, so when a rule is used multiple times during search it allows fresh instances every time it is applied.

There is one strange phenomenon you should be aware of. For efficiency reason (which played a big role in the early days of logic programming), problems such as $X = t$ are solved by just binding X to the term t. This is actually *logically incorrect* in some circumstances. For example, the equation $X = \mathsf{s}\,X$ does not have a (finite) solution because the right-hand side will always be strictly bigger than the left-hand side no matter what X is. However, Prolog will happily bind X to $\mathsf{s}\,X$, creating a circular term.

As we explain with the example of type inference in Section 8, we can work around this unsoundness, but it takes care.

6 Summary So Far

We refer to the logic programming interpreter as Prolog, also interpreters for similar backward-chaining languages are based on similar decision.

- When attempting to prove a judgment (= *solve a goal*) the rules (= *clauses*) are tried in a bottom-up manner in the order they are presented.
- We *backtrack* to the most recent decision when a subgoal fails.
- If there are multiple premises (= *multiple subgoals*) their proofs are attempted in left-to-right order.
- Conceptually, the result of computation is a derivation, although only the substitution for the free variables in the query is presented upon success together with “yes”.
- When proof construction fails without further alternatives for search, the result of computation is presented as “no”.
- Equations between schematic variables in the rules and the goals are solved by *unification* (although in Prolog in an unsound manner).
- Because classical logic programming is based on the predicate calculus, there is no static type checking. All terms belong to the same universal domain. Typographical errors often lead to failure (answer “no”) or unexpected, incorrect answers rather than static errors.

7 Prolog Syntax and Interpreter

The examples in this and the following section, can be found in the file lec18.pl.

In Prolog, rules such as

$$\frac{}{\mathsf{plus}\ 0\ y\ y}\ \mathsf{p0} \qquad \frac{\mathsf{plus}\ x\ y\ z}{\mathsf{plus}\ (\mathsf{s}\,x)\ y\ (\mathsf{s}\,z)}\ \mathsf{ps}$$

$$\frac{}{\mathsf{times}\ 0\ y\ 0}\ \mathsf{t0} \qquad \frac{\mathsf{times}\ x\ y\ w \quad \mathsf{plus}\ w\ y\ z}{\mathsf{times}\ (\mathsf{s}\,x)\ y\ z}\ \mathsf{ts}$$

are presented as

```
plus(0, Y, Y).
plus(s(X), Y, s(Z)) :- plus(X, Y, Z).

times(0, Y, 0).
times(s(X), Y, Z) :- times(X, Y, W), plus(W, Y, Z).
```

We see variables are in uppercase, while constant and function symbols are in lowercase. The inference rule (seen as a reverse implication) is shown as `:-`, and they are not named.

To load a file (*consult*, in Prolog terminology) we write its name, leaving out the `.pl` extension, as list at the top level. We used here GNU Prolog (gprolog), but Ciao (available on the Andrew machines) has a similar interface.

```
% gprolog
GNU Prolog 1.4.5 (64 bits)
Compiled Jul 14 2018, 19:58:18 with clang
By Daniel Diaz
Copyright (C) 1999-2018 Daniel Diaz
| ?- [lec18].
compiling lec18.pl for byte code...
lec18.pl:4: warning: singleton variables [Y] for times/3
lec18.pl:17-18: warning: singleton variables [Gamma] for tpof/3
lec18.pl:20-22: warning: singleton variables [B] for tpof/3
lec18.pl compiled, 22 lines read - 2979 bytes written, 7 ms

(1 ms) yes
| ?-
```

Now we can ask queries as sketched in the lecture on the board, but using Prolog syntax. We chain a few of them together. The line after the prompt `| ?-` is typed by the user, as is the semicolon (`;`) in order to obtain further solutions.

```
| ?- plus(s(0), s(s(0)), Z).

Z = s(s(s(0)))
```

```
yes
| ?- plus(X, Y, s(s(s(0)))).

X = 0
Y = s(s(s(0))) ? ;

X = s(0)
Y = s(s(0)) ? ;

X = s(s(0))
Y = s(0) ? ;

X = s(s(s(0)))
Y = 0 ? ;

no
| ?-
```

8 Implementing Type Inference

In Lecture 6 we implemented a bidirectional type checker based on verifications. Since verifications are at the very heart of logic, this approach is highly robust and extends to many languages with advanced features [Dunfield and Krishnaswami, 2022]. Functional languages such as ML and Haskell are instead based on *type inference* [Milner, 1978]. The basic observation is that the typing rules are *syntax-directed* (there is exactly one rule for each program constructs), so we always know the shape of a typing derivation. It remains to solve the equations between the types imposed by the rules. This is precisely what *unification* does! This means we can rewrite the typing rules as a logic program and have it perform type inference for us.

We use a list Γ to record types for all variables. It has the form of a Prolog list

```
  [tp(x1,a1), ..., tp(xn,an)]
```

where `tp` is a constructor, the x_i are variables, and a_i are types. We show the rules only for functions `lam(x,m)`, applications `app(m,n)`, and variables `x`. It is straightforward to extend to all the constructs in our language. The types on this fragment just have the form `arrow(a,b)`, but they may also contain free variables that represent polymorphism.

We begin with the first approximation of the code with a two significant bugs in Listing 1. The notation of a list with head `head` and tail `tail` is `[head | tail]`.

The first bug pertains to the last two clauses. The problem is that an expression such as `lam(x,lam(x,x))` could be assigned type `arrow(A,arrow(B,A))` even though `x` should only be able to refer to the inner binding (and therefore only have type `arrow(A,arrow(B,B))`). The solution is to *guard* the second clause with a test $X \neq Y$ and can be found in Listing 2.

```
%%% Warning: this code has bugs!

tpof(Gamma, lam(X,M), arrow(A,B)) :-
    tpof([tp(X,A)|Gamma], M, B).

tpof(Gamma, app(M,N), B) :-
    tpof(Gamma, M, arrow(A,B)),
    tpof(Gamma, N, A).

tpof([tp(X,A)|Gamma], var(X), A).

tpof([tp(Y,B)|Gamma], var(X), A) :-
    tpof(Gamma, var(X), A).
```

Listing 1: Type inference with two bugs

```
%%% Warning: this code has bugs!

%%% ...
tpof([tp(X,A)|Gamma], var(X), A).

tpof([tp(Y,B)|Gamma], var(X), A) :-
    X \= Y,
    tpof(Gamma, var(X), A).
```

Listing 2: Type inference with one bug

```
eq(X,Y) :- unify_with_occurs_check(X,Y).

tpof(Gamma, lam(X,M), arrow(A,B)) :-
    tpof([tp(X,A)|Gamma], M, B).

tpof(Gamma, app(M,N), B) :-
    tpof(Gamma, M, arrow(A1,B)),
    tpof(Gamma, N, A2),
    eq(A1,A2).

tpof([tp(X,A1)|Gamma], var(X), A2) :-
    eq(A1,A2).

tpof([tp(Y,B)|Gamma], var(X), A) :-
    X \= Y,
    tpof(Gamma, var(X), A).
```

Listing 3: Type inference corrected

The second problem has to do with the unsoundness of unification in Prolog. Indeed, if we asked (and we did)

```
| ?- tpof([], lam(x, app(var(x), var(x))), A).
```

we would find that Prolog thinks there is such an `A` but that it cannot be printed because it is circular. In languages such ML this will indeed fail to type check because it leads to the equations $A = B \supset C$ and $B = B \supset C$ which has no solution.

Prolog has a workaround, namely a built-in predicate `unify_with_occurs_check` which implements a sound unification algorithm. We need this in two places: when looking up variable in the context and in an application. The corrected code is in Listing 3.

You can find a few queries testing our code in Listing 4.

References

Alain Colmerauer and Philippe Roussel. The birth of Prolog. *ACM SIGPLAN Notices*, 28 (3):37–52, 1993.

Jana Dunfield and Neel Krishnaswami. Bidirectional typing. *ACM Computing Surveys*, 98 (5):1–38, 2022.

Robert. A. Kowalski. The early years of logic programming. *Communications of the ACM*, 31(1):38–43, 1988.

David Maier, K. Tuncay Tekle, Michael Kifer, and David S. Warren. Datalog: Concepts, history, and outlook. In Michael Kifer and Yanhong Annie Liu, editors, *Declarative Logic*

```
| ?- tpof([], lam(x, var(x)), A).

A = arrow(B,B) ? ;

no
| ?- tpof([], lam(x, lam(y, var(x))), A).

A = arrow(B,arrow(_,B)) ? ;

no
| ?- tpof([], lam(x, app(var(x), var(x))), A).

no
| ?- tpof([], lam(x, lam(y, lam(z, app(app(x,z), app(y,z))))), A).

no
| ?- tpof([], lam(x, lam(y, lam(z, app(app(var(x),var(z)),
                                         app(var(y),var(z)))))), A).

A = arrow(arrow(B,arrow(C,D)),arrow(arrow(B,C),arrow(B,D))) ? ;

no
| ?-
```

Listing 4: Testing type inference

Programming: Theory, Systems, and Applications, pages 3–100. ACM and Morgan & Claypool, 2018.

Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, August 1978.