Lecture Notes on
A Metacircular Interpreter

15-317: Constructive Logic
Frank Pfenning

Lecture 19
Thursday, March 30, 2023

1 Introduction

When developing functional programming, we made it a point to also investigate the differences, say, between proof reduction and computation. The key difference was that for computation we used a notion of observation and the goal of predictability to derive rules for a stepping judgment that embodies a "strategy".

Here we should to the same: inference rules themselves only specify correct derivations and not exactly how they are to be constructed. Various notion, like trying rules in order and backtracking, or deriving premises in left-to-right order should be specified (how else?) via inference rules. These rules could form the basis for an implementation and at the same time for proving properties such as soundness.

One cool way to do this is to start with a *metacircular interpreter*. That is, we write an implementation of a backward chaining logic programming engine in Prolog (so, more-or-less, in itself). As a first step along this path we need to represent programs as data. This is, we reify inference rules as data. But what kind of data? An entirely natural choice here is to use the language of propositions. For example, rules such as

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

would be represented as two propositions

$$\forall y.\, \mathsf{plus}(0, y, y)$$
$$\forall x.\, \forall y.\, \forall z.\, \mathsf{plus}(x, y, z) \supset \mathsf{plus}(\mathsf{s}(x), y, \mathsf{s}(z))$$

in predicate logic. Note that in this form, plus is now a predicate where previously it was a judgment.

Interestingly, the logic programming interpretation of such propositions is closely related to focusing. In order to understand this better, we should return to inversion and focusing and analyze the quantifiers in this context.

2 Inversion and Focusing with Quantifiers

In terms of sequent calculus, it turns out that $\forall x.\, A(x)$ is right invertible and not left invertible, and $\exists x.\, A(x)$ is left invertible and not right invertible. This means we have the following rules in the focusing calculus.

$$\frac{\Gamma, a\ \mathit{elem}\ ;\ \Omega \xrightarrow{\mathsf{R}} A(a)}{\Gamma\ ;\ \Omega \xrightarrow{\mathsf{R}} \forall x.\, A(x)}\ \forall R^a \qquad \frac{\Gamma \vdash t\ \mathit{elem} \quad \Gamma, [A(t)] \xrightarrow{\mathsf{FL}} C}{\Gamma, [\forall x.\, A(x)] \xrightarrow{\mathsf{FL}} C}\ \forall L$$

$$\frac{\Gamma \vdash t\ \mathit{elem} \quad \Gamma \xrightarrow{\mathsf{FR}} [A(t)]}{\Gamma \xrightarrow{\mathsf{FR}} [\exists x.\, A(x)]}\ \exists R \qquad \frac{\Gamma, a\ \mathit{elem}\ ;\ A(a) \cdot \Omega \xrightarrow{\mathsf{L}} C}{\Gamma\ ;\ (\exists x.\, A(x)) \cdot \Omega \xrightarrow{\mathsf{L}} C}\ \exists L^a$$

You may wonder about $\forall x.\, A(x)$ not being available in the premise of the $\forall L$ rules, but that works out because in the choice rule we *copy* the proposition we focus on from Γ. The fact that this remains complete with respect to the sequent calculus (and therefore also natural deduction) is a deep property and by no means obvious.

Like in the predicate calculus (see Lecture 13) we track parameters ranging over elements from a universal domain in Γ and use $\Gamma \vdash t\ \mathit{elem}$ to prove that the term t is well-formed. It turns out that in logic programming we (a) never use any invertible rules, and (b) the set of terms t is just built up from variables, constants, and functions applied to several terms.

$$\text{Terms} \quad t \quad ::= \quad x \mid c \mid f(t_1, \ldots, t_n)$$

So parameters a as such never arise.

3 Rules as Propositions

Part of the encoding of inference rules is that they become *propositions*, as we have seen in the introduction. But what kind of propositions do we need? And which could we allow?

First, and inference rule with now premises would just be a proposition quantified over all the schematic variables of the rule.

$$\frac{}{\mathsf{plus}\ 0\ y\ y}\ \mathsf{p0} \qquad \forall y.\, \mathsf{plus}(0, y, y)$$

From this, we deduce we have at least

$$\text{Clauses} \quad D \quad ::= \quad \forall x.\, D \mid P \mid \cdots$$

The term "clause" comes from the connection of logic programming with a restricted form of logical propositions called *Horn clauses*. For an inference rules with one premise we need $Q \supset P$ for atoms Q and P, but in general it should be $G \supset P$ where G becomes a *subgoal*.

$$\frac{\mathsf{plus}\ x\ y\ z}{\mathsf{plus}\ (\mathsf{s}\, x)\ y\ (\mathsf{s}\, z)}\ \mathsf{ps} \qquad \forall x.\, \forall y.\, \forall z.\, \mathsf{plus}(x, y, z) \supset \mathsf{plus}(\mathsf{s}(x), y, \mathsf{s}(z))$$

Goals G must include conjunction in case there are multiple subgoals.

$$\begin{array}{lrcl} \text{Horn clauses} & D & ::= & \forall x.\, D \mid G \supset P \mid P \\ \text{Goals} & G & ::= & P \mid G_1 \wedge G_2 \mid \exists x.\, G(x) \end{array}$$

We have also included existential quantification among the goals because (as we saw in the last lectures) top-level queries mostly require implicitly existentially quantified variables. Strictly speaking they are not part of the syntax of Horn clauses.

In order to examine the search behavior of Horn clauses and goals we should now consider how to organize proof construction for *propositions* in this fragment. Fortunately, we already have a very powerful tool: *focusing*.

We would like search to be goal-oriented. That is, let's assume we have Horn clause P, $P \supset Q$, and $Q \supset R$ and are trying to prove goal R. As we can see, the search strategy of *backward chaining* is the one that is goal-oriented. In backward chaining, atoms are *negative*. This in turn means we have the following rules, copied over from the general focusing system and specialized to Horn clauses as defined above.

Choice. For the choice rule, we anticipate that the succedent must be an atom P. That's because neither disjunction for falsehood are allowed in the fragment we are considering. As a result, we can only focus on the left, not the right when we have a choice. The key decision is, of course, which proposition in Γ we focus on.

$$\frac{D \in \Gamma \quad \Gamma, [D] \xrightarrow{\mathsf{FL}} P}{\Gamma \xrightarrow{\mathsf{C}} P}\ \mathsf{FLC}$$

Left Focus. In left focus, the key rule is that for implication. We have to solve the subgoal G while in focus, and the right-hand side of the implication has to match the goal. Let's consider the general case (on Horn clauses) and derive its specialized version.

$$\frac{\Gamma \xrightarrow{\mathsf{FR}} [G] \quad \dfrac{Q = P}{\Gamma, [Q] \xrightarrow{\mathsf{FL}} P}\ \mathsf{id}}{\Gamma, [G \supset Q] \xrightarrow{\mathsf{FL}} P}\ \supset L$$

We see that if $Q = P$ the second premise is derivable and only the first premise is left. We also see that if $Q \neq P$ then the second premise will fail, and therefore the rule is not

applicable. This leads to the specialized rule below.

$$\frac{}{\Gamma, [P] \xrightarrow{\mathsf{FL}} P}\ \mathsf{id} \qquad \text{(no rule for } Q \neq P\text{)} \quad \Gamma, [Q] \xrightarrow{\mathsf{FL}} P$$

$$\frac{\Gamma \xrightarrow{\mathsf{FR}} [G]}{\Gamma, [G \supset P] \xrightarrow{\mathsf{FL}} P}\ \supset L \qquad \text{(no rule for } Q \neq P\text{)} \quad \Gamma, [G \supset Q] \xrightarrow{\mathsf{FL}} P$$

$$\frac{\Gamma, [D(t)] \xrightarrow{\mathsf{FL}} P}{\Gamma, [\forall x.\, D(x)] \xrightarrow{\mathsf{FL}} P}\ \forall L$$

In the rule for universal quantification we dispense with the premise $\Gamma \vdash t\ \mathit{elem}$ because every well-formed term will be accepted. In particular, we never introduce any parameters into a derivation because $\forall$ appears only as an antecedent and $\exists$ as a succedent.

Right Focus.

$$\frac{\Gamma \xrightarrow{\mathsf{FR}} [G_1] \quad \Gamma \xrightarrow{\mathsf{FR}} [G_2]}{\Gamma \xrightarrow{\mathsf{FR}} [G_1 \wedge G_2]}\ \wedge R \qquad \frac{\Gamma \xrightarrow{\mathsf{FR}} [G(t)]}{\Gamma \xrightarrow{\mathsf{FR}} [\exists x.\, G(x)]}\ \exists R \qquad \frac{\Gamma \xrightarrow{\mathsf{C}} P}{\Gamma \xrightarrow{\mathsf{FR}} [P]}\ \mathsf{CFR}$$

So the specialized judgments for backward chaining are

$$\begin{array}{l} \Gamma \xrightarrow{\mathsf{C}} P \\ \Gamma, [D] \xrightarrow{\mathsf{FL}} P \\ \Gamma \xrightarrow{\mathsf{FR}} [G] \end{array}$$

where D and G are defined as for Horn clauses shown above and P is an atomic proposition, considered negative.

4 A Metacircular Interpreter

Now that we have the rules for backward chaining in logical form we can think of how to implement these rules in Prolog. Since backward chaining it also the foundation of Prolog, this is called a *metacircular interpreter* of a the language in itself. As we will see, this metacircular interpreter will not answer all the questions about the dynamics of the programs. Instead, some choices for how the object language behaves are mirrored by corresponding questions about how the metalanguage behaves.

We start with the choice judgment, which calls upon membership and invokes focus left.

$$\frac{D \in \Gamma \quad \Gamma, [D] \xrightarrow{\mathsf{FL}} P}{\Gamma \xrightarrow{\mathsf{C}} P}\ \mathsf{FLC}$$

```
choose(Gamma, atom(P)) :-
    mem(D, Gamma),
    focusL(Gamma, D, atom(P)).
```

Here we use a representation where there object language proposition $G_1 \wedge G_2$ is the metalanguage term `and(G1,G2)`, P is `atom(P)`, $G \supset P$ is `imp(G,atom(P))`. Also, Γ is just a Prolog list.

We'll some back to the `mem` predicate in a bit. Let's move on to left focus.

$$\frac{}{\Gamma, [P] \xrightarrow{\mathsf{FL}} P}\ \mathsf{id} \qquad \frac{\Gamma \xrightarrow{\mathsf{FR}} [G]}{\Gamma, [G \supset P] \xrightarrow{\mathsf{FL}} P}\ \supset L$$

The rule id suggests

```
focusL(Gamma, atom(P), atom(P)).
```

This would be fine if we would be willing to accept the unsound unification of Prolog, and maybe you are, but I couldn't bring myself to do that. To make it sound, we have to bind two different variables and then unify them soundly using the built-in `unify_with_occurs_check`. Because we will need it again in the next clause, we factor out a `unify` predicate.

```
unify(P,Q) :- unify_with_occurs_check(P,Q).

focusL(Gamma, atom(Q), atom(P)) :- unify(Q,P).
focusL(Gamma, imp(G,atom(Q)), atom(P)) :-
    unify(Q,P),
    focusR(Gamma, G).
```

To prove the subgoal, we call upon the right focus judgment, represented as the predicate `focusR`. Postponing the issue of quantifiers, the rules for right focus are straightforward.

$$\frac{\Gamma \xrightarrow{\mathsf{FR}} [G_1] \quad \Gamma \xrightarrow{\mathsf{FR}} [G_2]}{\Gamma \xrightarrow{\mathsf{FR}} [G_1 \wedge G_2]}\ \wedge R \qquad \frac{\Gamma \xrightarrow{\mathsf{C}} P}{\Gamma \xrightarrow{\mathsf{FR}} [P]}\ \mathsf{CFR}$$

```
focusR(Gamma, and(G1,G2)) :-
    focusR(Gamma, G1),
    focusR(Gamma, G2).
focusR(Gamma, atom(P)) :-
    choose(Gamma, atom(P)).
```

The case of conjunction is a case where we see that the subgoals on the object language will be proved left to right precisely if the goals in the metalanguage (Prolog) will be proved left to right. So, somehow, this metacircular interpreter doesn't precisely fix subgoal order, except if we already know Prolog's subgoal order. But in a sense we were just trying to define that via the metacircular interpreter! In the next lecture we will see how to avoid this

kind of object/meta dependency so that the object language goes left-to-right no matter how the metalanguage proves its subgoals.

This bring us to the issue of quantifiers. In logic programming languages such as λ-Prolog [Miller and Nadathur, 2012] and Twelf [Pfenning and Schürmann, 1999] there is intrinsic support for bound variables, so it is easy to represent the propositions $\forall x.\, D(x)$ and $\exists x.\, G(x)$. In Prolog, there is no such support. Explicitly programming it is certainly possible, but tedious and takes some of the elegance out of the interpreter.

Fortunately, Prolog has an another mechanism we can use. We can represent a proposition with *free* variables and then use the build-in `copy_term` to make a copy with fresh variables substituted for the free variables. Using this technique we can eliminate quantifiers altogether. When we try the members of the antecedents Γ in turn, we have to create fresh copy each time. So:

```
mem(Dcopy, [D | Gamma]) :- copy_term(D, Dcopy).
mem(Dcopy, [_ | Gamma]) :- mem(Dcopy, Gamma).

choose(Gamma, atom(P)) :-
    mem(D, Gamma),
    focusL(Gamma, D, atom(P)).
```

Here is another instance where the metalanguage dynamics (it tries the clauses for `mem` in the given order) means that the object language dynamics does the same thing (it tries the elements of Γ in order). For example, if we flipped the two clauses for `mem`, then computation in the object language would try the its program clauses from right to left.

Now, when interpreting an example of, say, addition, we have to be careful regarding the interpretation of free variables. Consider:

```
ex3([atom(plus(0, Y, Y)),
     imp(atom(plus(X, Y, Z)), atom(plus(s(X), Y, s(Z))))]).

query3(Z) :- ex3(Gamma),
             choose(Gamma, atom(plus(s(0), s(s(0)), Z))).
query4(X,Y) :- ex3(Gamma),
               choose(Gamma, atom(plus(X, Y, s(s(s(s(0))))))).
```

Here `ex3(Gamma)` will bind `Gamma` to the list in the previous clause. In this list we see two propositions D, one the case for `0` and one the case for `s(X)`. It looks like the variable `Y` is shared between them, but that's not the case, because is copied separately by the `mem` predicate. Similarly, the use of `Z` in `query3` and `X` and `Y` in `query4` is unrelated to the variables of the same name in `ex3`.

You can find the complete live code for the meta-interpreter and examples in the file meta.pl. We also summarize the code (without the examples) in Listing 2 and show and interaction inf Listing **??**.

```
mem(Dcopy, [D | Gamma]) :- copy_term(D, Dcopy).
mem(Dcopy, [_ | Gamma]) :- mem(Dcopy, Gamma).

unify(P,Q) :- unify_with_occurs_check(P,Q).
% unify(P,P).  % Prolog, but logically unsound

choose(Gamma, atom(P)) :-
    mem(D, Gamma),
    focusL(Gamma, D, atom(P)).

focusL(Gamma, atom(Q), atom(P)) :- unify(Q,P).
focusL(Gamma, imp(G,atom(Q)), atom(P)) :-
    unify(Q,P),
    focusR(Gamma, G).

focusR(Gamma, and(G1,G2)) :-
    focusR(Gamma, G1),
    focusR(Gamma, G2).
focusR(Gamma, atom(P)) :-
    choose(Gamma, atom(P)).
```

Listing 1: Metacircular Horn clause interpreter

```
| ?- [meta].

(2 ms) yes
| ?- query4(X,Y).

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

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

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

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

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

no
```

Listing 2: Running the metacircular interpreter

References

Dale Miller and Gopalan Nadathur. *Programming with Higher-Order Logic*. Cambridge University Press, 2012.

Frank Pfenning and Carsten Schürmann. System description: Twelf — a meta-logical framework for deductive systems. In H. Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction (CADE-16)*, pages 202–206, Trento, Italy, July 1999. Springer-Verlag LNAI 1632.