Lecture Notes on
Chaining

15-317: Constructive Logic
Frank Pfenning*

Lecture 17
April 22, 2021

1 Introduction

There is a pleasing simplicity of the computation as proof search paradigm when based on inference rules. But how do we now implement it? Proof rules are versatile but the rules themselves are meant to be fixed during proof search. While the rules of each logic program are fixed, they change as we move from one program to another. We want one mechanism that describes logic programming proof search the same way, receiving a syntactic representation of the particular rules of the logic program as input. One way would be to capture the *rules as propositions* so we can represent a program as a collection of propositions, which become the antecedents of a sequent. The query will then be the succedent, and we can hopefully use our usual tools of logical reasoning to obtain an operational semantics.

But what is the connection between rules and propositions, and can we reformulate the bottom-up search strategy using inference rules instead with antecedent propositions? This is the subject of today's lecture.

*Edits by André Platzer

2 Rules as Propositions

Let's review a set of rules for increment in binary (omitting some fine points on standard representations of numbers).

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

The first one inc_e is trivial to express in propositions as just an atomic proposition; the second (inc_0) requires a simple quantifier. The third one has a premise, in which case we can rewrite the inference rule as an implication. Let's write the resulting collection of antecedents as Γ_{inc}.

$$\begin{array}{rcl} \Gamma_{\mathsf{inc}} & = & \mathsf{inc}(\mathsf{e}, \mathsf{b1}(\mathsf{e})), \\ & & \forall m.\, \mathsf{inc}(\mathsf{b0}(m), \mathsf{b1}(m)), \\ & & \forall m.\, \forall n.\, \mathsf{inc}(m, n) \supset \mathsf{inc}(\mathsf{b1}(m), \mathsf{b0}(n)) \end{array}$$

We don't yet have the tools to see in which way this translation is correct, but at least in an intuitive sense it should look plausible.

Now, solving a logic programming query such as $\mathsf{inc}(\mathsf{b1}(\mathsf{e}), \mathsf{b1}(\mathsf{b0}(\mathsf{e})))$ will be modeled as

$$\Gamma_{\mathsf{inc}} \xrightarrow{f} \mathsf{inc}(\mathsf{b1}(\mathsf{e}), \mathsf{b1}(\mathsf{b0}(\mathsf{e})))$$

where "$\xrightarrow{f}$" demands a sequent calculus we have yet to design. Here we use f to suggest *focusing*, although in this lecture we will just introduce *chaining*. Focusing, eventually, will turn out to be *inversion* + *chaining*. The particular form of chaining from this lecture is called *backward chaining*; in the next lecture we also discuss forward chaining.

3 Chaining

We notice first that all propositions in Γ_{inc} are *noninvertible*: neither atoms, nor universal quantification (because of their need for choosing an instantiation), nor implication have invertible rules as antecedents. So the inversion strategy will not be useful. The second thing we notice is that *if we still had rules*, we would know immediately that only the inc_1, represented as the third antecedent, will actually help in the example at the end of the previous section. That is, we need to "look beyond" the quantifiers and to the right of the implication inside the formulas until we find an atomic formula and then decide if it would match our goal.

To formalize this, we *focus* on a particular proposition among the antecedents and then continue to apply rules *only to the proposition in focus* until we can determine whether it matches the succedent. We write $[A]$ for the particular proposition A that is in focus. There can be at most one proposition in focus in a sequent.

The first rule picks a proposition from Γ (which, in our case, would be Γ_{inc}) and puts it into focus. For now, the succedent will always be an atom, so we write P for the succedent.

$$\frac{A \in \Gamma \quad \Gamma, [A] \xrightarrow{f} P}{\Gamma \xrightarrow{f} P}\ \mathsf{focus}L$$

The second rule instantiates a universal quantifier. Of course, at this point we cannot know which term we might want to choose, so we instantiate it with a variable X, whose particular value is to be determined later.

$$\frac{\Gamma, [A(X)] \xrightarrow{f} P}{\Gamma, [\forall x.\, A(x)] \xrightarrow{f} P}\ \forall L$$

When we reach an implication $B \supset A$, we would like to "look past" it at A, but we also have to remember that we still have to prove B in case this A *does* match the succedent.

$$\frac{\Gamma, [A] \xrightarrow{f} P \quad \Gamma \xrightarrow{f} [B]}{\Gamma, [B \supset A] \xrightarrow{f} P}\ {\supset}L$$

Two remarks: (1) we put $\Gamma, [A] \xrightarrow{f} P$ as the first premise (reversing the order we have used so far for implication) because we would like to know if it matches the succedent before we solve B, and (2) the focus is inherited by both subformulas A and B. With this order of premises, we first focus on checking whether the specific result A would help prove P before we focus on trying to prove its required assumption B.

With this process, we may finally come upon an atom Q, in which case it has to unify with the succedent P.

$$\frac{Q = P}{\Gamma, [Q] \xrightarrow{f} P}\ \mathsf{id}$$

Here, we imagine that $Q = P$ is a unification which instantiates the free variables like X we have introduced into the proof. Note that the substitution for such variables has to be *global* throughout the entire partial proof tree. If we wanted to be more formal about this (although I don't believe it really helps understanding at this point), we would "thread through" a substitution or a set of constraints through all the rules.

The following observation is critical: *if Q and P do not unify, then no rule applies here.*

$$\begin{array}{c}\text{no rule if } Q \neq P\\ \Gamma, [Q] \xrightarrow{f} P\end{array}$$

Since the rules only ever apply to propositions in focus, and only one proposition can be in focus, here $[Q]$, and no rule applies if the atoms Q and P do not unify, the entire proof search fails (except for the choice of which proposition from Γ to focus on with rule $\mathsf{focus}L$). Consequently, proof search could only proceed by focusing on another proposition initially.

Along the way we have introduced a second judgment form, $\Gamma \xrightarrow{f} [A]$, where the focus is on the right-hand side. For the example so far, we only need one rule, where we lose focus *at atomic propositions*, which is called blurring.

$$\frac{\Gamma \xrightarrow{f} P}{\Gamma \xrightarrow{f} [P]}\ \mathsf{blur}$$

Here is a summary of the rules so far:

$$\frac{A \in \Gamma \quad \Gamma, [A] \xrightarrow{f} P}{\Gamma \xrightarrow{f} P}\ \mathsf{focus}L$$

$$\frac{\Gamma, [A(X)] \xrightarrow{f} P}{\Gamma, [\forall x.\, A(x)] \xrightarrow{f} P}\ \forall L \qquad \frac{\Gamma, [A] \xrightarrow{f} P \quad \Gamma \xrightarrow{f} [B]}{\Gamma, [B \supset A] \xrightarrow{f} P}\ \supset\! L$$

$$\frac{Q = P}{\Gamma, [Q] \xrightarrow{f} P}\ \mathsf{id} \qquad \begin{array}{c}\text{no rule if } Q \neq P\\ \Gamma, [Q] \xrightarrow{f} P\end{array}$$

$$\frac{\Gamma \xrightarrow{f} P}{\Gamma \xrightarrow{f} [P]}\ \mathsf{blur}$$

4 1+1=2

Now we apply the above rules to check the result of incrementing 1 in binary form. Recall the encoding of the rules

$$\begin{array}{rcl}\Gamma_{\mathsf{inc}} & = & \mathsf{inc}(\mathsf{e}, \mathsf{b1}(\mathsf{e})),\\ & & \forall m.\, \mathsf{inc}(\mathsf{b0}(m), \mathsf{b1}(m)),\\ & & \forall m.\, \forall n.\, \mathsf{inc}(m, n) \supset \mathsf{inc}(\mathsf{b1}(m), \mathsf{b0}(n))\end{array}$$

and the goal sequent, as yet without proof

$$\begin{array}{c}\vdots\\ \Gamma_{\mathsf{inc}} \xrightarrow{f} \mathsf{inc}(\mathsf{b1}(\mathsf{e}), \mathsf{b0}(\mathsf{b1}(\mathsf{e})))\end{array}$$

Sticking with the clause order of logic programming, first we try $\mathsf{inc}(\mathsf{e}, \mathsf{b1}(\mathsf{e}))$:

$$\dfrac{\begin{array}{c}\vdots\\ \Gamma_{\mathsf{inc}}, [\mathsf{inc}(\mathsf{e}, \mathsf{b1}(\mathsf{e}))] \xrightarrow{f} \mathsf{inc}(\mathsf{b1}(\mathsf{e}), \mathsf{b0}(\mathsf{b1}(\mathsf{e})))\end{array}}{\Gamma_{\mathsf{inc}} \xrightarrow{f} \mathsf{inc}(\mathsf{b1}(\mathsf{e}), \mathsf{b0}(\mathsf{b1}(\mathsf{e})))}\ \mathsf{focus}L$$

We fail immediately, because we are in a situation of the form $\Gamma, [Q] \xrightarrow{f} P$ where $Q \neq P$.

Next we try the second proposition in Γ_{inc}:

$$\dfrac{\begin{array}{c}\vdots\\ \Gamma_{\mathsf{inc}}, [\forall m.\, \mathsf{inc}(\mathsf{b0}(m), \mathsf{b1}(m))] \xrightarrow{f} \mathsf{inc}(\mathsf{b1}(\mathsf{e}), \mathsf{b0}(\mathsf{b1}(\mathsf{e})))\end{array}}{\Gamma_{\mathsf{inc}} \xrightarrow{f} \mathsf{inc}(\mathsf{b1}(\mathsf{e}), \mathsf{b0}(\mathsf{b1}(\mathsf{e})))}\ \mathsf{focus}L$$

At this point we use a new variable M for m to postpone a choice until some unification constraints help us determine a good instantiation of the quantifier.

$$\dfrac{\dfrac{\begin{array}{c}\vdots\\ \Gamma_{\mathsf{inc}}, [\mathsf{inc}(\mathsf{b0}(M), \mathsf{b1}(M))] \xrightarrow{f} \mathsf{inc}(\mathsf{b1}(\mathsf{e}), \mathsf{b0}(\mathsf{b1}(\mathsf{e})))\end{array}}{\Gamma_{\mathsf{inc}}, [\forall m.\, \mathsf{inc}(\mathsf{b0}(m), \mathsf{b1}(m))] \xrightarrow{f} \mathsf{inc}(\mathsf{b1}(\mathsf{e}), \mathsf{b0}(\mathsf{b1}(\mathsf{e})))}\ \forall L}{\Gamma_{\mathsf{inc}} \xrightarrow{f} \mathsf{inc}(\mathsf{b1}(\mathsf{e}), \mathsf{b0}(\mathsf{b1}(\mathsf{e})))}\ \mathsf{focus}L$$

Again we fail, this time because $\mathsf{b0}(M)$ does not unify with $\mathsf{b1}(\mathsf{e})$. Third time's a charm:

$$\dfrac{\begin{array}{c}\vdots\\ \Gamma_{\mathsf{inc}}, [\forall m.\, \forall n.\, \mathsf{inc}(m,n) \supset \mathsf{inc}(\mathsf{b1}(m), \mathsf{b0}(n))] \xrightarrow{f} \mathsf{inc}(\mathsf{b1}(\mathsf{e}), \mathsf{b0}(\mathsf{b1}(\mathsf{e})))\end{array}}{\Gamma_{\mathsf{inc}} \xrightarrow{f} \mathsf{inc}(\mathsf{b1}(\mathsf{e}), \mathsf{b0}(\mathsf{b1}(\mathsf{e})))}\;\mathsf{focus}L$$

Combining two consecutive $\forall L$ rules, we arrive at

$$\dfrac{\dfrac{\begin{array}{c}\vdots\\ \Gamma_{\mathsf{inc}}, [\mathsf{inc}(M,N) \supset \mathsf{inc}(\mathsf{b1}(M), \mathsf{b0}(N))] \xrightarrow{f} \mathsf{inc}(\mathsf{b1}(\mathsf{e}), \mathsf{b0}(\mathsf{b1}(\mathsf{e})))\end{array}}{\Gamma_{\mathsf{inc}}, [\forall m.\, \forall n.\, \mathsf{inc}(m,n) \supset \mathsf{inc}(\mathsf{b1}(m), \mathsf{b0}(n))] \xrightarrow{f} \mathsf{inc}(\mathsf{b1}(\mathsf{e}), \mathsf{b0}(\mathsf{b1}(\mathsf{e})))}\;\forall L \times 2}{\Gamma_{\mathsf{inc}} \xrightarrow{f} \mathsf{inc}(\mathsf{b1}(\mathsf{e}), \mathsf{b0}(\mathsf{b1}(\mathsf{e})))}\;\mathsf{focus}L$$

Now we are forced into the $\supset L$ rule. The incomplete proof state is now

$$\dfrac{\dfrac{\dfrac{\begin{array}{c}\vdots\\ \Gamma_{\mathsf{inc}}, [\mathsf{inc}(\mathsf{b1}(M), \mathsf{b0}(N))] \xrightarrow{f} \mathsf{inc}(\mathsf{b1}(\mathsf{e}), \mathsf{b0}(\mathsf{b1}(\mathsf{e})))\end{array} \quad \begin{array}{c}\vdots\\ \Gamma_{\mathsf{inc}} \xrightarrow{f} [\mathsf{inc}(M,N)]\end{array}}{\Gamma_{\mathsf{inc}}, [\mathsf{inc}(M,N) \supset \mathsf{inc}(\mathsf{b1}(M), \mathsf{b0}(N))] \xrightarrow{f} \mathsf{inc}(\mathsf{b1}(\mathsf{e}), \mathsf{b0}(\mathsf{b1}(\mathsf{e})))}\;\supset L}{\Gamma_{\mathsf{inc}}, [\forall m.\, \forall n.\, \mathsf{inc}(m,n) \supset \mathsf{inc}(\mathsf{b1}(m), \mathsf{b0}(n))] \xrightarrow{f} \mathsf{inc}(\mathsf{b1}(\mathsf{e}), \mathsf{b0}(\mathsf{b1}(\mathsf{e})))}\;\forall L \times 2}{\Gamma_{\mathsf{inc}} \xrightarrow{f} \mathsf{inc}(\mathsf{b1}(\mathsf{e}), \mathsf{b0}(\mathsf{b1}(\mathsf{e})))}\;\mathsf{focus}L$$

Now, the first branch succeeds by unification with $M = \mathsf{e}$ and $N = \mathsf{b1}(\mathsf{e})$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\mathsf{inc}(\mathsf{b1}(M), \mathsf{b0}(N)) = \mathsf{inc}(\mathsf{b1}(\mathsf{e}), \mathsf{b0}(\mathsf{b1}(\mathsf{e})))}{\Gamma_{\mathsf{inc}}, \mathsf{inc}(\mathsf{b1}(M), \mathsf{b0}(N)) \xrightarrow{f} \mathsf{inc}(\mathsf{b1}(\mathsf{e}), \mathsf{b0}(\mathsf{b1}(\mathsf{e})))}\;\mathsf{id} \quad \begin{array}{c}\vdots\\ \Gamma_{\mathsf{inc}} \xrightarrow{f} [\mathsf{inc}(M,N)]\end{array}}{\Gamma_{\mathsf{inc}}, [\mathsf{inc}(M,N) \supset \mathsf{inc}(\mathsf{b1}(M), \mathsf{b0}(N))] \xrightarrow{f} \mathsf{inc}(\mathsf{b1}(\mathsf{e}), \mathsf{b0}(\mathsf{b1}(\mathsf{e})))}\;\supset L}{\Gamma_{\mathsf{inc}}, [\forall m.\, \forall n.\, \mathsf{inc}(m,n) \supset \mathsf{inc}(\mathsf{b1}(m), \mathsf{b0}(n))] \xrightarrow{f} \mathsf{inc}(\mathsf{b1}(\mathsf{e}), \mathsf{b0}(\mathsf{b1}(\mathsf{e})))}\;\forall L \times 2}{\Gamma_{\mathsf{inc}} \xrightarrow{f} \mathsf{inc}(\mathsf{b1}(\mathsf{e}), \mathsf{b0}(\mathsf{b1}(\mathsf{e})))}\;\mathsf{focus}L$$

This substitution has to be applied *globally* to the partial proof, which yields

$$\dfrac{\dfrac{\dfrac{}{\Gamma_{\mathsf{inc}}, \mathsf{inc}(\mathsf{b1}(\mathsf{e}), \mathsf{b0}(\mathsf{b1}(\mathsf{e}))) \xrightarrow{f} \mathsf{inc}(\mathsf{b1}(\mathsf{e}), \mathsf{b0}(\mathsf{b1}(\mathsf{e})))}\;\mathsf{id} \qquad \dfrac{\vdots}{\Gamma_{\mathsf{inc}} \xrightarrow{f} [\mathsf{inc}(\mathsf{e}, \mathsf{b1}(\mathsf{e}))]}}{\dfrac{\Gamma_{\mathsf{inc}}, [\mathsf{inc}(\mathsf{e}, \mathsf{b1}(\mathsf{e})) \supset \mathsf{inc}(\mathsf{b1}(\mathsf{e}), \mathsf{b0}(\mathsf{b1}(\mathsf{e})))] \xrightarrow{f} \mathsf{inc}(\mathsf{b1}(\mathsf{e}), \mathsf{b0}(\mathsf{b1}(\mathsf{e})))}{\Gamma_{\mathsf{inc}}, [\forall m.\, \forall n.\, \mathsf{inc}(m, n) \supset \mathsf{inc}(\mathsf{b1}(m), \mathsf{b0}(n))] \xrightarrow{f} \mathsf{inc}(\mathsf{b1}(\mathsf{e}), \mathsf{b0}(\mathsf{b1}(\mathsf{e})))}\;\forall L \times 2}\;{\supset}L}{\Gamma_{\mathsf{inc}} \xrightarrow{f} \mathsf{inc}(\mathsf{b1}(\mathsf{e}), \mathsf{b0}(\mathsf{b1}(\mathsf{e})))}\;\mathsf{focus}L$$

Singling out the remaining subgoal, we can solve this now by focusing on the first of the propositions in Γ_{inc}. Trying to focus on any other one of the antecedents will fail.

$$\dfrac{\dfrac{\dfrac{}{\Gamma_{\mathsf{inc}}, [\mathsf{inc}(\mathsf{e}, \mathsf{b1}(\mathsf{e}))] \xrightarrow{f} \mathsf{inc}(\mathsf{e}, \mathsf{b1}(\mathsf{e}))}\;\mathsf{id}}{\Gamma_{\mathsf{inc}} \xrightarrow{f} \mathsf{inc}(\mathsf{e}, \mathsf{b1}(\mathsf{e}))}\;\mathsf{focus}L}{\Gamma_{\mathsf{inc}} \xrightarrow{f} [\mathsf{inc}(\mathsf{e}, \mathsf{b1}(\mathsf{e}))]}\;\mathsf{blur}R$$

Notice that in the example we could have added many many other propositions to the antecedent and it would not have changed the outcome as long as the eventual proposition at the end of the chain of quantifiers and implications is not of the form $\mathsf{inc}(\_, \_)$. In an implementation of a logic programming language such as Prolog we compile the programs (that is, the antecedents) so that we have direct access to all of those defining the particular predicate in the query (that is, the succedent).

5 Horn Clauses

For which fragment of the logic does this proof search strategy work? Answering this question will give us (a slight generalization) of the important concept of *Horn clauses*, and those are the only ones allowed in backward chaining logic programming languages like Prolog.

A key seems to be that the antecedents all have noninvertible left rules that can be *chained together*. Similarly, we would like the succedent to have noninvertible right rules, again so we can chain them (not visible in this example). Recall that choosing noninvertible rules requires a decision that we may have to take back (but also that Prolog search prescribes a fixed strategy for making those decisions). Remembering polarities: those propositions with invertible right rules are *negative* and have noninvertible left rules, while those with invertible left rules have noninvertible right rules and are *positive*. We write D^- for the negative propositions in the antecedent[1], and G^+ for the positive proposition in the succedent. For this lecture, all atomic propositions are negative P^-, but we will allow positive atomic propositions in the future. We concentrate on the core propositions of Horn clauses in red and leave out some additional propositions, which the theory would predict are compatible with backward chaining.

$$\begin{array}{lcll}
\text{Program formulas} & D^- & ::= & P^- \mid G^+ \supset D^- \mid \forall x.\, D^-(x) \mid D_1^- \wedge D_2^- \mid \top \\
\text{Programs} & \Gamma^- & ::= & \cdot \mid \Gamma^-, D^- \\
\text{Goal formulas} & G^+ & ::= & {\downarrow}P^- \mid G_1^+ \wedge G_2^+ \mid \top \mid \exists x.\, G^+(x) \mid G_1^+ \vee G_2^+ \mid \bot
\end{array}$$

There may be a couple of surprises here. One is that conjunction is both positive and negative. This is because there are two forms of left rules for conjunction, one that decomposes $A \wedge B$ to A, B (which is invertible, and therefore positive, recall $\wedge L$) and one that extracts either A or B (which is noninvertible and therefore negative, recall $\wedge L_1$). But come to think of it, conjunctions separate multiple clauses (which offers a choice), but also separate multiple remaining subgoals in a goal formula (which all need to be proved). The shift operator $\downarrow$ in ${\downarrow}P^-$ can be thought as explicitly indicating that a negative atomic proposition P^- can also be used in a positive position such as as the single original atomic goal in Prolog. We have three judgment forms

$$\begin{array}{ll}
\Gamma^- \xrightarrow{f} P^- & \text{stable sequent} \\
\Gamma^-, [D^-] \xrightarrow{f} P^- & \text{left focus} \\
\Gamma^- \xrightarrow{f} [G^+] & \text{right focus}
\end{array}$$

It is also helpful to consider what we *left out*, which is (syntactically) very little: program formulas D^- cannot be of the form ${\uparrow}G^+$, and goal formulas G^+ can neither be positive atoms P^+ nor of the form ${\downarrow}D^-$ (except ${\downarrow}P^-$).

$$\begin{array}{lcll}
\text{Program formulas} & D^- & ::= & \cdots \mid \cancel{{\uparrow}G^-} \\
\text{Goal formulas} & G^+ & ::= & \cdots \mid \cancel{P^\pm} \mid \cancel{{\downarrow}D^-}
\end{array}$$

By forcing goals to be almost entirely positive and programs to be entirely negative, no inversion will ever be applied. The fact that we omitted pos-

[1] D stands for "definitive clauses" from the early days of Prolog.

itive atoms means chaining is always backwards (see next lecture). Essentially, backward chaining logic programming arises from a pure backward chaining interpretation of intuitionistic logic.

Another interesting aspect is that *Horn clauses are so restricted that classical and intuitionistic logic coincide on them!* That is, a goal P^- is provable from a program Γ^- in classical logic if and only if the sequent is provable in intuitionistic logic. So at least in logic programming, there shouldn't be any arguments between intuitionists and classical logicians.

We now restate the rules for core Horn clauses, using our new notation. Also, for existential goals, we introduce the $\exists R$ rule. For both $\exists R^*$ and $\forall L^*$ we need a globally fresh variable X; the superscript * is supposed to be a notational reminder of this condition.

$$\frac{D^- \in \Gamma^- \quad \Gamma^-, [D^-] \xrightarrow{f} P^-}{\Gamma^- \xrightarrow{f} P^-}\ \mathsf{focus}L$$

$$\frac{\Gamma^-, [D^-(X)] \xrightarrow{f} P^-}{\Gamma^-, [\forall x.\, D^-(x)] \xrightarrow{f} P^-}\ \forall L^* \qquad \frac{\Gamma^-, [D^-] \xrightarrow{f} P^- \quad \Gamma^- \xrightarrow{f} [G^+]}{\Gamma^-, [G^+ \supset D^-] \xrightarrow{f} P^-}\ \supset L$$

$$\frac{Q^- = P^-}{\Gamma^-, [Q^-] \xrightarrow{f} P^-}\ \mathsf{id} \qquad \begin{array}{c}\text{no rule if } Q^- \neq P^- \\ \Gamma^-, [Q^-] \xrightarrow{f} P^-\end{array}$$

$$\frac{\Gamma^- \xrightarrow{f} [G_1^+] \quad \Gamma^- \xrightarrow{f} [G_2^+]}{\Gamma^- \xrightarrow{f} [G_1^+ \wedge G_2^+]}\ \wedge R \qquad \frac{}{\Gamma^- \xrightarrow{f} [\top]}\ \top R$$

$$\frac{\Gamma^- \xrightarrow{f} [G^+(X)]}{\Gamma^- \xrightarrow{f} [\exists x. G^+(x)]}\ \exists R^* \qquad \frac{\Gamma^- \xrightarrow{f} P^-}{\Gamma^- \xrightarrow{f} [{\downarrow} P^-]}\ \mathsf{blur}$$

6 Writing a Meta-Interpreter

Now that we have rewritten the foundation of logic programming in propositional form, can we take advantage of this for an implementation? An ideal language for such an implementation would already support unification and backtracking, since both of these are at the heart of the *computation-as-proof-search* paradigm. We don't have to look far: let's use Prolog! This means we are writing what is called a *meta-circular interpreter*: we provide the semantics of a language in itself. There can be some arguments whether this counts as "definitional" since, for example, the meta-interpreter will not explain the details of unification because they are just inherited from the metalanguage. Nevertheless, we can modify the semantics of chaining by modifying the meta-interpreter, and we will play through an example of this.

Now consider a program Γ^- and a query P^-, which should proceed by searching for a proof of $\Gamma^- \xrightarrow{f} P^-$. In the remainder of this section we will just omit the polarity annotations, remembering the program clauses D and atoms P, Q are negative while goals G are positive.

We begin with program representation, that is, the representation of Γ. The idea is that every element D of Γ is represented in Prolog as a fact `prog(D)`. For example,

```
prog(inc(e,b1(e))).
```

This clause contains no variables. For the next one, $\forall m.\, \mathsf{inc}(\mathsf{b0}(m), \mathsf{b1}(m))$, we have to determine how to represent the universal quantifier. Unfortunately, this is difficult in Prolog because it leaves quantifiers implicit and considers free variables in the program to be universally quantified. We exploit this, by instantiating m with a new free variable M and stating

```
prog(inc(b0(M),b1(M))).
```

For Horn clauses as we have defined them, it is always possible to move the universal quantifier in program formulas D all the way outside, where they correspond to Prolog's implicit universal quantification.

Finally, the last proposition in Γ_{inc}

$$\forall m.\, \forall n.\, \mathsf{inc}(m, n) \supset \mathsf{inc}(\mathsf{b1}(m), \mathsf{b0}(n))$$

requires an implication. We write this in prefix form, $G \supset D$ as `imp(G,D)`, so we don't get confused between the language we are interpreting `imp(G,D)` and the language in which we implement it in, whose implication would be written as `D :- G`.

```
prog(imp(inc(M,N),inc(b1(M),b0(N)))).
```

Again, the quantified variables become free variables.

A priori, we could think the proof search would be through a predicate `solve(Gamma,G)`. But the logic program Γ never changes during proof search in the Horn fragment, so it is sufficient to consider just `solve(G)`. The first two clauses should be clear: they implement $\wedge R$ and $\top R$:

```
solve(true).
solve(and(G1,G2)) :- solve(G1), solve(G2).
```

The third possibility is an atom. We did not complicate the syntax with an explicit atom constructor (perhaps we should have . . .), so we have a predicate `atm/1` that recognizes atoms. In our example:

```
atm(inc(_,_)).
```

After we have recognized that we are trying to prove an atom, we have to nondeterministically select a program clause D and then see if $\Gamma, [D] \xrightarrow{f} P$. Selecting D takes place by calling `prog(D)`, and the $\Gamma, [D] \xrightarrow{f} P$ will be implemented by `focus(D, P)`. Γ can remain implicit, because it is represented as facts in Prolog as explained before.

```
solve(P) :- atm(P), prog(D), focus(D, P).
```

The `focus/2` predicate now distinguishes the cases for its first argument. Since quantifiers are implicit, at this point this only pertains to implications and atoms. For atoms, we unify them and make sure no clause applies if they don't unify.

```
focus(Q,P) :- atm(Q), Q = P.
```

For implications, we continue to focus on D first, and if that eventually succeeds, we solve the subgoal.

```
focus(imp(G,D),P) :- focus(D,P), solve(G).
```

At this point, we already have a complete meta-interpreter for the Horn clause fragment with prefixed quantifiers. Here is the summary.

```
solve(true).
solve(and(G1,G2)) :- solve(G1), solve(G2).
solve(P) :- atm(P), prog(D), focus(D, P).
focus(Q,P) :- atm(Q), Q = P.
focus(imp(G,D),P) :- focus(D,P), solve(G).
```

Our example of binary addition:

```
atm(inc(_,_)).

prog(inc(e,b1(e))).
prog(inc(b0(M),b1(M))).
prog(imp(inc(M,N),inc(b1(M),b0(N)))).
```

We can now exercise the meta-interpreter, again using free variables in the query to model existentially quantified variables in the query.

```
| ?- solve(inc(b1(e),N)).

N = b0(b1(e)) ? ;

(1 ms) no
| ?- solve(inc(M,b1(e))).

M = e ? ;

M = b0(e) ? ;

no
```

With the second example we also see that Prolog backtracking implements backtracking in our small Horn clause language.

7 Repairing Unsound Unification

One unfortunate consequence of using Prolog as our implementation language is that we inherit its unsound unification. This means as a logic-based proof search engine, our meta-interpreter has a severe bug. Fortunately, Prolog also provides us with the means to fix it.

As an example, consider the query which should have no solution: there is no m such that $\mathsf{inc}(\mathsf{b0}(m), \mathsf{b1}(\mathsf{b0}(m)))$! But instead of failing, it uses the second clause and incorrectly succeeds in unifying $M = \mathsf{b0}(M)$, creating a circular term $M = \mathsf{b0}(\mathsf{b0}(\mathsf{b0}(\ldots)))$

```
?- solve(inc(b0(M),b1(b0(M)))).

cannot display cyclic term for M ? ;

no
```

If we want to make the program sound, we have to scrutinize the meta-interpreter for (a) explicit calls to unification, or (b) implicit unifications which arise from repeated variables in the heads of clauses. We see there are no repeated variables, and the only explicit call to unification is in the case of atoms. So we can just replace this with a call to the library predicate `unify_with_occurs_check` that implements sound unification.

```
solve(true).
solve(and(G1,G2)) :- solve(G1), solve(G2).
solve(P) :- atm(P), prog(D), focus(D, P).
% focus(Q,P) :- atm(Q), Q = P.   % unsound unification
focus(Q,P) :- atm(Q), unify_with_occurs_check(Q,P).
focus(imp(G,D),P) :- focus(D,P), solve(G).
```

Now the previous query fails, as we had hoped.

```
?- solve(inc(b0(M),b1(b0(M)))).

no
```

8 Further Variants of the Semantics

We can now add the remaining connectives from our definitions of positive and negative propositions, leaving only the quantifiers implicit. Note that there are two clauses for disjunctive goals, and two clauses for conjunctive programs. We could have implemented this using a disjunctive goal `A ; B` in Prolog, but we prefer to keep the meta-language constructs simple.

Just to show what can be done with a metainterpreter, we also add a minimal tracing facility. We do that via a predicate `display/1` which outputs its argument, and `nl/0` which outputs a newline. Note that `fail` is necessary so that Prolog backtracks after it has printed the goal and uses the other clauses for `solve/1` to actually solve the goal. Other variations on this can be easily imagined.

```
% solve(G) succeeds if Gamma --> G
% focus(D,Q) succeeds if Gamma, [D] --> Q
solve(G) :- display(G), nl, fail.

solve(true).
solve(and(G1,G2)) :- solve(G1), solve(G2).
solve(or(G1,G2)) :- solve(G1).
solve(or(G1,G2)) :- solve(G2).
% no clause for solve(false)
solve(P) :- atm(P), prog(D), focus(D,P).

focus(imp(G,D),P) :- focus(D,P), solve(G).
focus(and(D1,D2),P) :- focus(D1, P).
focus(and(D1,D2),P) :- focus(D2, P).
focus(Q,P) :- atm(Q), unify_with_occurs_check(Q,P).
```

For example, the following query prints the three atomic goals it encounters in sequence (due to the carry bit), in each case with fresh internally named existential variables.

```
?- solve(inc(b1(b1(e)),N)).
inc(b1(b1(e)),_283)
inc(b1(e),_302)
inc(e,_315)

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

no
```

As a final possibility, consider how you might instrument the interpreter to not just present an answer substitution, but a proof term. The `solve/1` predicate would then be generalized to `solve/2` and we would ask, for example

```
solve(M,inc(e,N))
```

for some goal which would not only show `N = b1(e)` but also a proof $M : \mathsf{inc}(\mathsf{e}, \mathsf{b1}(\mathsf{e}))$ that was found by the interpreter.