Lecture Notes on Unification

15-317: Constructive Logic
Frank Pfenning

Lecture 21
Thursday, April 6, 2023

1 Introduction

We have resolved one of three questions of how backward chaining on Horn clauses should be interpreted as a logic programming language, namely subgoal ordering. Clause selection (including backtracking) and unification remain. We will talk about unification and how it is implemented in Prolog.

We also start on *forward chaining* in Section 5, an entirely different computational interpretation of Horn clauses. As the foundation of Datalog, it has found many applications for tasks such as program analysis, compilation, access control, and many others.

2 Existential Quantification

First, we note that we can eliminate the existential quantifier from the definition of Horn clauses, although it is certainly useful to explain free logic variables in a query. We can eliminate it because

$$(\exists x.\, A(x)) \supset B \equiv \forall x.\, (A(x) \supset B)$$

is a valid logic equivalence. Based on our scoping conventions, writing it in this way implies that x does not occur in B. Let's quickly prove this in the sequent calculus. As usual, we implicitly apply weakening, dropping proposition from among the antecedents when they are no longer needed.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{a\ \mathit{elem} \vdash a\ \mathit{elem}} \quad \dfrac{\cdots}{a\ \mathit{elem}, A(a) \Longrightarrow A(a)}\,\mathsf{id}}{a\ \mathit{elem}, A(a) \Longrightarrow \exists x.\, A(x)}\,\exists R \quad \dfrac{\cdots}{B \Longrightarrow B}\,\mathsf{id}}{(\exists x.\, A(x)) \supset B, a\ \mathit{elem}, A(a) \Longrightarrow B}\,\supset L}{(\exists x.\, A(x)) \supset B, a\ \mathit{elem} \Longrightarrow A(a) \supset B}\,\supset R}{(\exists x.\, A(x)) \supset B \Longrightarrow \forall x.\, (A(x) \supset B)}\,\forall I^a$$

$$\dfrac{\dfrac{}{a\ \mathit{elem} \vdash a\ \mathit{elem}} \quad \dfrac{\dfrac{\dots\dots\dots\dots\dots\dots\dots\dots\dots\dots}{a\ \mathit{elem}, A(a) \Longrightarrow A(a)}\ \mathsf{id} \quad \dfrac{\dots\dots\dots\dots}{B \Longrightarrow B}\ \mathsf{id}}{A(a) \supset B, a\ \mathit{elem}, A(a) \Longrightarrow B}\ \supset L}{\dfrac{\forall x.\,(A(x) \supset B), a\ \mathit{elem}, A(a) \Longrightarrow B}{\dfrac{\forall x.\,(A(x) \supset B), \exists x.\,A(x) \Longrightarrow B}{\forall x.\,(A(x) \supset B) \Longrightarrow (\exists x.\,A(x)) \supset B}\ \supset I}\ \exists L^a}\ \forall L$$

So if we have a clause $(\exists x.\,G(x)) \supset P$ we can transform it into the equivalent $\forall x.\,G(x) \supset P$. Another interesting observation: if we pick $B = \bot$, then the law says

$$\neg\exists x.\,A(x) \equiv \forall x.\,\neg A(x)$$

Of course, the dual does *not* hold: we cannot push a negation through a universal quantifier because a proof of $\exists x.\,\neg A(x)$ will have to exhibit a witness for x, while a proof of $\neg\forall x.\,A(x)$ does not.

3 Substitution and Unification

If we ignore existentials, then the only rule for quantification in backward chaining is $\forall L$.

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

The idea is to simply instantiate the quantifier with a fresh variable X and accumulate and solve constraints imposed upon X and other variables in the goals and clauses.

$$\dfrac{\Gamma, [D(X)] \xrightarrow{\mathsf{FL}} P \quad (X \text{ not in } P \text{ or } D(x))}{\Gamma, [\forall x.\,D(x)] \xrightarrow{\mathsf{FL}} P}\ \forall L$$

Unfortunately, this freshness condition doesn't turn out to be correct. That's because we solve subgoals of $G_1 \wedge G_2$ independently, but we want X chosen while solving G_1 also to be fresh with respect to G_2. Furthermore, when we have multiple subgoals we need to communicate the substitution created by solving the first one to the second. This has come up in several examples such as the times judgment for multiplication.

One solution seems to be to use our goal stack judgment because then a later subgoal G_2 is available for us to check. In any case, we need to *return* a substitution as a result of computation, so we should incrementally compute is as we are proving the goal.

Slightly differently from what we did in lecture, our new judgment then becomes something like

$$\theta \;/\; \Gamma, [D] \xrightarrow{\mathsf{FL}} P \;/\; \theta'$$

where θ is an input substitution and θ' is an output substitution. We don't show the rules for this judgment, leaving them to a later lecture or as an exercise, so we can focus our attention on the process of unification itself.

$$\begin{array}{lrcl} \text{Terms} & t & ::= & c \mid f(t_1, \ldots, t_n) \mid x \mid X \\ \text{Substitutions} & \theta & ::= & \cdot \mid \theta, t/X \end{array}$$

Note that we distinguish between variables x that are bound by quantifiers and free (logic) variables X that are subject to substitution.

Our main judgment for unification is

$$s \doteq t \;/\; \theta$$

Soundness of unification requires that $[\theta]s = [\theta]t$, where $[\theta]s$ is, as before, our notation for applying a substitution to the free variables in a term. Equality here is meant to be ordinary mathematical equality (the two terms are equal), while $s \doteq t \;/\; \theta$ should be read as "*θ is a unifier for s and t*".

We consider a simple example, without making the substitution explicit.

$$f(X, X) \doteq f(g(a, Y), g(Y, a))$$

We see that for this equation to hold we must have

$$X \doteq g(a, Y) \quad \text{and} \quad X \doteq g(Y, a)$$

Therefore $g(a, Y) \doteq g(Y, a)$ which requires $Y \doteq a$ and $a \doteq Y$. Putting it all together the final unifying substitution would be

$$(g(a, a)/X, a/Y)$$

There are many algorithms for unification, and even more implementations. The one we show in this lecture is more-or-less Robinson's original algorithm [Robinson, 1971]. In order to handle terms $f(t_1, \ldots, t_n)$ we write $\bar{t}$ for a sequence of terms and add a new judgment to unify sequences $\bar{s} \doteq \bar{t}$.

$$\frac{}{c = c \;/\; (\cdot)} \qquad \text{no rule for } c \neq d$$

$$c = d \;/\; \theta$$

$$\frac{\bar{s} \doteq \bar{t} \;/\; \theta}{f(\bar{s}) \doteq f(\bar{t}) \;/\; \theta} \qquad \text{no rule for } f \neq g$$

$$f(\bar{s}) \doteq g(\bar{t}) \;/\; \theta$$

$$\frac{s \doteq t \;/\; \theta_1 \quad [\theta_1]\bar{s} \doteq [\theta_1]\bar{t} \;/\; \theta_2}{(s, \bar{s}) \doteq (t, \bar{t}) \;/\; \theta_2 \circ \theta_1} \qquad \frac{}{() \doteq () \;/\; (\cdot)}$$

The second to last rule is the trickiest so far. Consider our specification. If $s \doteq t \;/\; \theta_1$ then $[\theta_1]s = [\theta_1]t$. This may require substitution for some variables that are free in $\bar{s}$ and $\bar{t}$. We therefore apply θ_1 to the remaining sequent of terms before unifying them. Measuring against our spec, we have, from the two premises:

$$\begin{array}{lcl} [\theta_1]s & = & [\theta_1]t \\ [\theta_2]([\theta_1]\bar{s}) & = & [\theta_2]([\theta_1]\bar{t}) \end{array}$$

Therefore, the composition $[\theta_2 \circ \theta_1]\bar{s} = [\theta_2 \circ \theta_1]\bar{t}$. Moreover, since already $[\theta_1]s = [\theta_1]t$ we can apply a further substitution to its free variables and maintain the equality, so $[\theta_2 \circ \theta_1]s = [\theta_2][\theta_1]s = [\theta_2][\theta_1]t = [\theta_2 \circ \theta_1]t$

Next, we come to the rules for variables. For two equal variables, the empty substitution is sufficient. When we equate $X \doteq Y$, we can substitute X for Y or Y for X, as we wish. The case for constant is also clear, so only functions are somewhat interesting.

$$\frac{}{X \doteq X \;/\; (\cdot)} \qquad \frac{X \neq Y}{X \doteq Y \;/\; (Y/X)} \qquad \frac{}{X \doteq c \;/\; (c/X)} \qquad \frac{}{c \doteq X \;/\; (c/X)}$$

Let's consider one of the last two remaining cases, since other is symmetric. It is tempting to conjecture the rule

$$\frac{}{X \doteq f(\bar{t}) \;/\; (f(\bar{t})/X)} \;??$$

But let's check by applying the substitution to both sides:

$$\begin{array}{lcl} [f(\bar{t})/X]X & = & f(\bar{t}) \\ [f(\bar{t})/X]f(\bar{t}) & = & f([f(\bar{t})/X]\bar{t}) \end{array}$$

The two are only equal if X does not occur in $\bar{t}$, usually abbreviated as $X \notin \bar{t}$. This gives us our final two rules:

$$\frac{X \notin \bar{t}}{X \doteq f(\bar{t}) \;/\; (f(\bar{t})/X)} \qquad \frac{X \notin \bar{t}}{f(\bar{t}) \doteq X \;/\; (f(\bar{t})/X)}$$

The proof that these rules are *sound* with respect to the specification is not difficult. We already embedded it in our explanation of the rules. What is more difficult to is that it is also *complete*. That is, if there is a unifier the algorithm will find one. The crucial property, needed for the rule concerning pairs, is the following

> *Consider s and t such that there is a substitution σ such that $[\sigma]s = [\sigma]t$. Then there is a substitution θ such that $s \doteq t \;/\; \theta$ and $\sigma = \theta' \circ \theta$ for some θ'.*

This means that any other unifying substitution can be seen as an instance of the one determined by the rules. In the terminology of the field, we say that if $s \doteq t \;/\; \theta$ then θ is a *most general unifier* of s and t.

4 Unification in Prolog

The so-called *occurs-check* in the last two rules of unification is omitted by Prolog unless you call `unify_with_occurs_check(s,t)`. This is because it was deemed to expensive in the early days of Prolog, so we sacrificed logical correctness to the god of efficiency (never a good idea, in my opinion).

To understand a little bit better what happens in Prolog, we need to talk about the term representation. We can think of a variable X as holding a pointer. Initially, when X is created, it points to itself (perhaps inspired by the identity substitution X/X). When

it is unified with a term t which is allocated somewhere on the heap, we just change the pointer to the address of t, which is extremely fast. Doing an occurs-check here would be an $O(n)$ operation, where n is the size of the term t. That is clearly unacceptable because in a procedure call such as `plus(m, n, W)` the occurs-check would require traversing m and n at each level in the recursion.

However, consider the clause

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

The variables X, Y, and Z are implicitly universally quantified and we make a fresh copy of the clause before we try to unify `plus(s(X), Y, s(Z))` with a goal `plus(m, n, W)` for some terms m and n.

When we unify `s(X)` with m there is *no need for an occurs-check*. That's because X was just created as a fresh variable, so it cannot occur anywhere in m. The same is true for Y and Z. In other words, no occurs-check needs to be done at all for this clause.

Next consider the clause

```
plus(0, Y, Y).
```

Here, we don't need any occurs-check for the first occurrence of Y, but when we encounter the second we cannot simply assign it or avoid unification. Consider, for example,

```
?- plus(0, X, s(X)).
```

Clearly, this should fail, but without the occurs-check it will eventually unify $X \doteq s(X)$ which succeeds in Prolog.

So if we anticipate that `plus` could be called with arguments that have their own free variables as second and third arguments, we should rewrite this clause as

```
plus(0, Y1, Y2) :- unify_with_occurs_check(Y1,Y2).
```

Now that the head of the clause `plus(0, Y1, Y2)` no longer contains any repeated variables, Prolog will just assign them to the corresponding terms in the goal. Sound unification will then fail as appropriate.

There is another important case where Prolog unification is sound, namely when we know in a unification problem such as $X = t$ that t has no free variables. This is often known because many Prolog programs only run correctly when certain arguments to predicates are known to be ground. Unfortunately, in standard Prolog there is no way to declare this property and have it checked for correctness before running a query, although some implementations support so-called *mode declarations* and static checks.

5 Saturation

We now move on to a new subject, namely the idea of using inference rules for Horn clauses, interpreted from the top down, as a kind of logic program. Why is this even reasonable? Consider a very simple set of two rules:

$$\frac{}{\mathsf{even}\ 0} \qquad \frac{\mathsf{even}\ x}{\mathsf{even}\ (\mathsf{s}\,(\mathsf{s}\,x))}$$

If we tried to use that in the top-down direction we would start with $\mathsf{even}\ 0$, then learn $\mathsf{even}\ (\mathsf{s}\,(\mathsf{s}\,0))$, and so on. This process would never terminate.

As a different style of example, let's consider reachability in a directed graph. We can specify this with

$$\frac{\mathsf{edge}\ x\ y}{\mathsf{path}\ x\ y}\ \mathsf{edge}_1 \qquad \frac{\mathsf{path}\ x\ y \quad \mathsf{path}\ y\ z}{\mathsf{path}\ x\ z}\ \mathsf{trans}$$

As we saw in recitation, these rules do not make a good backward-chaining (that is, Prolog) logic program because it will not terminate on many graphs.

However, given an edge relation we can run these rules in the forward direction to compute paths. Here is an example evolution of our knowledge, given the first two facts.

$\mathsf{edge}\ a\ b$	given
$\mathsf{edge}\ b\ c$	given
$\mathsf{path}\ a\ b$	by rule edge_1
$\mathsf{path}\ b\ c$	by rule edge_1
$\mathsf{path}\ a\ c$	by rule trans

At this point we can of course still apply some rules, but they will always result in facts we already know. We say the database of facts is *saturated*. So in forward-chaining proof construction computation stops when the database is saturated. We can then query the saturated database, for example, to find out if $\mathsf{path}\ a\ c$ (which succeeds) or $\mathsf{path}\ a\ a$ (which fails, because paths here don't include self-loops). The facts are just looked up—there is no further computation besides that.

The problems posed by termination in backward chaining and forward chaining are quite different. In backward chaining we have to make sure a measure on some ground terms get smaller in recursive calls. In forward chaining we typically show termination by showing that the collection of facts that could potentially in the database is finite. For example, in the program for computing paths, there are at most n^2 facts of the form $\mathsf{path}\ x\ y$ if there are n vertices. Therefore saturation has to terminate. In many cases we can characterize asymptotic complexity from the form of the rules, using some fascinating *meta-complexity theorems* [Ganzinger and McAllester, 2001, 2002].

6 Forward Chaining

In a surprising twist, we can obtain logic programming using top-down inference from Horn clauses by giving all atomic predicates *positive* polarity [Andreoli, 1992, Chaudhuri et al., 2008]. As for backward chaining, we start by simply specializing the focusing rules for Horn clauses in Figure 1.

Due to the usual interpretation of quantifiers as creating fresh logic variables instead guessing terms t, the identity rule should unify the two sides instead of testing for equality.

The computational behavior is further constrained as follows:

(1) In the rule CFL, where we transition from left focus to a choice sequent, we actually fail when $P \in \Gamma$, that is, when the new antecedent P is already known. This is expressed in the rule CFL'. With logic variables, we fail if P is *an instance* of some fact Q with variables.

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

Choice. For the choice left rule FLC, D^- cannot be an atom, because atoms are positive.

$$\frac{D^- \in \Gamma \quad \Gamma, [D^-] \xrightarrow{\mathsf{FL}} G}{\Gamma \xrightarrow{\mathsf{C}} G}\ \mathsf{FLC} \qquad \frac{\Gamma \xrightarrow{\mathsf{FR}} [G]}{\Gamma \xrightarrow{\mathsf{C}} G}\ \mathsf{FRC}$$

Right Focus.

$$\frac{}{\Gamma, P \xrightarrow{\mathsf{FR}} [P]}\ \mathsf{id} \qquad \begin{array}{c}\text{(no rule if } P \notin \Gamma\text{)} \\ \Gamma \xrightarrow{\mathsf{FR}} [P]\end{array}$$

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

Left Focus.

$$\frac{\Gamma \xrightarrow{\mathsf{FR}} [G] \quad \Gamma, [P] \xrightarrow{\mathsf{FL}} G}{\Gamma, [G \supset P] \xrightarrow{\mathsf{FL}} G}\ \supset L \qquad \frac{\Gamma, P \xrightarrow{\mathsf{C}} G}{\Gamma, [P] \xrightarrow{\mathsf{FL}} G}\ \mathsf{CFL} \qquad \frac{\Gamma, [D(t)] \xrightarrow{\mathsf{FL}} G}{\Gamma, [\forall x.\, D(x)] \xrightarrow{\mathsf{FL}} G}\ \forall L$$

Left Focus with Saturation. Replace CFL with CFL'

$$\frac{P \notin \Gamma \quad \Gamma, P \xrightarrow{\mathsf{FL}} G}{\Gamma, [P] \xrightarrow{\mathsf{FL}} G}\ \mathsf{CFL}' \qquad \begin{array}{c}\text{(no rule if } P \in \Gamma\text{)} \\ \Gamma, [P] \xrightarrow{\mathsf{FL}} G\end{array}$$

Figure 1: Forward chaining

(2) In the choice rules, we always try all possible ways to focus on the left. Only if all of them fail (either because they don't apply or the conclusion is already among the antecedent) do we focus on the right. This will either succeed or fail outright—it does not call back to the choice or left focus judgments.

We walk through an example to see how this represents logical inference. Consider

$$\Gamma_0 \xrightarrow{\mathsf{C}} R \quad \text{for} \quad \Gamma_0 = (P, P \supset Q, (P \wedge Q) \supset R)$$

We can try to left focus on $P \supset Q$ and on $(P \wedge Q) \supset R$. Focusing on the latter will fail, but let's see how:

$$\dfrac{\dfrac{\dfrac{P \in \Gamma_0}{\Gamma_0 \xrightarrow{\mathsf{FR}} [P]}\,\mathsf{id} \qquad \text{fails, since } Q \notin \Gamma_0}{\Gamma_0 \xrightarrow{\mathsf{FR}} [P \wedge Q]}\,\wedge R \qquad \begin{array}{c}\vdots\\ \Gamma_0, [R] \xrightarrow{\mathsf{FL}} R\end{array}}{\Gamma_0, [(P \wedge Q) \supset R] \xrightarrow{\mathsf{FL}} R}\,\supset L}{\Gamma_0 \xrightarrow{\mathsf{C}} R}\,\mathsf{FLC}$$

Since the first subgoal of $\supset L$ fails, we don't consider the second one. It would eventually succeed.

But we can successfully focus on $P \supset Q$ as follows:

$$\dfrac{\dfrac{\dfrac{P \in \Gamma_0}{\Gamma_0 \xrightarrow{\mathsf{FR}} [P]}\,\mathsf{id} \qquad \dfrac{\begin{array}{c}\vdots\\ \Gamma_0, Q \xrightarrow{\mathsf{FL}} R\end{array}}{\Gamma_0, [Q] \xrightarrow{\mathsf{FL}} R}\,\mathsf{CFL}'}{\Gamma_0, [P \supset Q] \xrightarrow{\mathsf{FL}} R}\,\supset L}{\Gamma_0 \xrightarrow{\mathsf{C}} R}\,\mathsf{FLC}$$

Now that we have added Q to our antecedents we can successfully focus on $(P \wedge Q) \supset R$ since P is already present. In the abstract system of forward chaining, we could also focus again on $P \supset Q$, but in the presence of saturation we would fail since $Q \in (\Gamma_0, Q)$.

To summarize the first two phases of focusing:

$$\begin{array}{c}\Gamma_0, Q, R \xrightarrow{\mathsf{C}} R\\ \vdots\\ \Gamma_0, Q \xrightarrow{\mathsf{C}} R\\ \vdots\\ \Gamma_0 \xrightarrow{\mathsf{C}} R\end{array}$$

At this point, computationally, left focusing on any proposition in Γ_0 would fail since it woulc only add either Q or R, both of which we already have among the antecedents. The antecedents (representing our database of facts) are *saturated*.

So we attempt to focus on the succedent R, which closes the computation immediately.

$$\dfrac{\dfrac{}{\Gamma_0, Q, R \xrightarrow{\mathsf{FR}} [R]}\,\mathsf{id}}{\Gamma_0, Q, R \xrightarrow{\mathsf{C}} R}\,\mathsf{FRC}$$
$$\vdots$$
$$\Gamma_0, Q \xrightarrow{\mathsf{C}} R$$
$$\vdots$$
$$\Gamma_0 \xrightarrow{\mathsf{C}} R$$

If we think of these as "big steps", each is essentially forced. Ultimately, in this case, there is just one derivation of the conclusion.

Even though the order of steps may vary, if all facts are *ground* (no free variables), then the saturated database is unique even if the order of the focusing steps may differ. For example, consider

$$P, P \supset Q, P \supset R \xrightarrow{\mathsf{C}} R$$

In the saturated database we will have Q and R, regardless of which we infer first.

In the next lecture we will explore forward chaining logic programming further.

References

Jean-Marc Andreoli. Logic programming with focusing proofs in linear logic. *Journal of Logic and Computation*, 2(3):197–347, 1992.

Kaustuv Chaudhuri, Frank Pfenning, and Greg Price. A logical characterization of forward and backward chaining in the inverse method. *Journal of Automated Reasoning*, 40(2–3): 133–177, 2008. Special issue with selected papers from IJCAR 2006.

Harald Ganzinger and David A. McAllester. A new meta-complexity theorem for bottom-up logic programs. In T.Nipkow R.Goré, A.Leitsch, editor, *Proceedings of the First International Joint Conference on ArAutomated Reasoning (IJCAR'01)*, pages 514–528, Siena, Italy, June 2001. Springer-Verlag LNCS 2083.

Harald Ganzinger and David A. McAllester. Logical algorithms. In P.Stuckey, editor, *Proceedings of the 18th International Conference on Logic Programming*, pages 209–223, Copenhagen, Denmark, July 2002. Springer-Verlag LNCS 2401.

J. A. Robinson. Computational logic: The unification computation. *Machine Intelligence*, 6: 63–72, 1971.