Lecture Notes on Logic Programming

15-317: Constructive Logic
Frank Pfenning*

Lecture 13
October 2, 2015

1 Computation vs. Deduction

The previous lectures explored a connection between logic and computation based on the observation that once we have a proof, it corresponds to a functional program. In this lecture we switch to an entirely different connection between logic and computation. The starting point is that the search for a proof has a computational interpretation. We interpret logical rules as programs that are executed by proof search according to a fixed strategy.

Logic programming is a particular way to approach programming. Other paradigms we might compare it to are imperative programming or functional programming. The divisions are not always clear-cut—a functional language may also have some imperative aspects, for example—but the mindset of various paradigms is quite different and determines how we design and reason about programs.

To understand logic programming, we first examine the difference between computation and deduction. To *compute* we start from a given expression and, according to a fixed set of rules (the program) generate a result. For example, $25 + 46 \to (2 + 4 + 1)1 \to (6 + 1)1 \to 71$ for a computation of decimal addition with carry. To *deduce* we start from a conjecture and, according to a fixed set of rules (the axioms and inference rules), try to construct a proof of the conjecture. So computation is mechanical and

*With edits by André Platzer

requires no ingenuity, while deduction is a creative process. For example, for all $n > 2$: $a^n + b^n \neq c^n$, ... *357 years of hard work ...*, QED.

Philosophers, mathematicians, and computer scientists have tried to unify the two, or at least to understand the relationship between them for centuries. For example, George Boole[1] succeeded in reducing a certain class of logical reasoning to computation in so-called Boolean algebras. Since the fundamental undecidability results of the 20th centuries we know that not everything we can reason about is in fact mechanically computable, even if we follow a well-defined set of formal rules.

In this course we are interested in a connection of a different kind. A first observation is that computation can be seen as a limited form of deduction, because computation actually establishes theorems, too. For example, $15+26 = 41$ is both the result of a computation, and a theorem of arithmetic. Conversely, deduction can be considered a form of computation if we fix a strategy for proof search, removing the guesswork (and the possibility of employing ingenuity) from the deductive process.

This latter idea is the foundation of logic programming. Logic program computation proceeds by proof search according to a fixed strategy. By knowing what this strategy is, we can implement particular algorithms in logic, and execute the algorithms by proof search according to this fixed strategy.

2 Judgments and Proofs

Since logic programming computation is proof search, to study logic programming means to study proofs. We adopt here the approach by Martin-Löf [3]. Although he studied logic as a basis for functional programming rather than logic programming, his ideas are more fundamental and therefore equally applicable in both paradigms.

The most basic notion is that of a *judgment*, which is an object of knowledge. We know a judgment because we have evidence for it. The kind of evidence we are most interested in is a *proof*, which we display as a *deduction* using *inference rules* in the form

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

where R is the name of the rule (often omitted), J is the judgment established by the inference (the *conclusion*), and $J_1, \ldots, J_n$ are the *premisses* of

[1] 1815–1864

the rule. We can read it as

If J_1 and $\cdots$ and J_n then we can conclude J by virtue of rule R.

By far the most common judgment is the truth of a proposition A, which we write as A *true*. Because we will be occupied almost exclusively with the thruth of propositions for quite some time in this course we generally omit the trailing "*true*". Other examples of judgments on propositions are A *false* (A is false), A *true at* t (A is true at time t, the subject of temporal logic), or K *knows* A (K knows that A is true, the subject of epistemic logic).

To give some simple examples we need a language to express propositions. We start with *terms* t that have the form $f(t_1, \ldots, t_n)$ where f is a *function symbol* of arity[2] n and $t_1, \ldots, t_n$ are the arguments. Terms can have variables in them, which we generally denote by upper-case letters. *Atomic propositions* P have the form $p(t_1, \ldots, t_n)$ where p is a *predicate symbol* of arity n and $t_1, \ldots, t_n$ are its arguments. Later we will introduce more general forms of propositions, built up by logical connectives and quantifiers from atomic propositions.

In our first set of examples we represent natural numbers $0, 1, 2, \ldots$ as terms of the form $\mathsf{0}, \mathsf{s}(\mathsf{0}), \mathsf{s}(\mathsf{s}(\mathsf{0})), \ldots$, using two function symbols ($\mathsf{0}$ of arity 0 and s of arity 1).[3] The first predicate we consider is the predicate even of arity 1. Its meaning is defined by two inference rules:

$$\frac{}{\mathsf{even}(\mathsf{0})}\ \mathsf{evz} \qquad\qquad \frac{\mathsf{even}(N)}{\mathsf{even}(\mathsf{s}(\mathsf{s}(N)))}\ \mathsf{evs}$$

The first rule, evz, expresses that $\mathsf{0}$ is even. It has no premiss and therefore is like an axiom. The second rule, evs, expresses that if N is even, then $\mathsf{s}(\mathsf{s}(N))$ is also even. Here, N is a *schematic variable* of the inference rule: every instance of the rule where N is replaced by a concrete term represents a valid inference. We have no more rules, so we think of these two as defining the predicate even completely in the sense that there are no other circumstances under which we would know $\mathsf{even}(N)$ except those justified by a series of uses of both rules.

[2] A function f of arity n expects the n arguments $t_1, \ldots, t_n$.

[3] This is not how numbers are represented in practical logic programming languages such as Prolog, but it is a convenient source of examples.

The following is a trivial example of a deduction, showing that 4 is even:

$$\cfrac{\cfrac{\cfrac{}{\mathsf{even}(0)}\ \mathsf{evz}}{\mathsf{even}(\mathsf{s}(\mathsf{s}(0)))}\ \mathsf{evs}}{\mathsf{even}(\mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{s}(0)))))}\ \mathsf{evs}$$

Here, we used the rule evs twice: once with $N = \mathsf{0}$ and once with $N = \mathsf{s}(\mathsf{s}(0))$.

3 Proof Search

To make the transition from inference rules to logic programming we need to impose a particular strategy. Two fundamental ideas suggest themselves: we could either search backward from the conjecture, growing a (potential) proof tree upwards until all resulting premises are proved, or we could work forwards from the axioms applying rules until we arrive at the conjecture. We call the first one *goal-directed* and the second one *forward-reasoning*.

goal-directed search $\uparrow$

$$\cfrac{\cfrac{\cfrac{}{\mathsf{even}(0)}\ \mathsf{evz}}{\mathsf{even}(\mathsf{s}(\mathsf{s}(0)))}\ \mathsf{evs}}{\mathsf{even}(\mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{s}(0)))))}\ \mathsf{evs}$$

$\downarrow$ forward-reasoning search

In the logic programming literature we find the terminology *top-down* for goal-directed, and *bottom-up* for forward-reasoning, but this goes counter to the direction in which the proof tree is constructed. Logic programming was conceived with goal-directed search, and this is still the dominant direction since it underlies Prolog, the most popular logic programming language. Later in the class, we will also have an opportunity to consider forward reasoning.

Goal-directed Proof Search. In the first approximation, the goal-directed strategy we apply is very simple: given a conjecture (called the *goal*) we determine which inference rules might have been applied to arrive at this conclusion. We select one of them and then recursively apply our strategy to all the premisses as subgoals. If there are no premisses we have completed the proof of the goal. We will consider many refinements and more precise descriptions of search in this course.

For example, consider the conjecture $\mathsf{even}(\mathsf{s}(\mathsf{s}(0)))$. We now execute the logic program consisting of the two rules evz and evs to either prove or refute this goal. We notice that the only rule with a matching conclusion is evs. Our partial proof now looks like

$$\dfrac{\begin{array}{c}\vdots\\ \mathsf{even}(0)\end{array}}{\mathsf{even}(\mathsf{s}(\mathsf{s}(0)))}\ \mathsf{evs}$$

with $\mathsf{even}(0)$ as the only subgoal.

Considering the subgoal $\mathsf{even}(0)$ we see that this time only the rule evz could have this conclusion. Moreover, this rule has no premisses so the computation terminates successfully, having found the proof

$$\dfrac{\dfrac{}{\mathsf{even}(0)}\ \mathsf{evz}}{\mathsf{even}(\mathsf{s}(\mathsf{s}(0)))}\ \mathsf{evs}.$$

Actually, most logic programming languages will not show the proof in this situation, but only answer `yes` if a proof has been found, which means the conjecture was true.

Failing Proof Search. Now consider the goal $\mathsf{even}(\mathsf{s}(\mathsf{s}(\mathsf{s}(0))))$. Clearly, since 3 is not even, the computation must fail to produce a proof. Following our strategy, we first reduce this goal using the evs rule to the subgoal $\mathsf{even}(\mathsf{s}(0))$, with the incomplete proof

$$\dfrac{\begin{array}{c}\vdots\\ \mathsf{even}(\mathsf{s}(0))\end{array}}{\mathsf{even}(\mathsf{s}(\mathsf{s}(\mathsf{s}(0))))}\ \mathsf{evs}.$$

At this point we note that there is no rule whose conclusion matches the goal $\mathsf{even}(\mathsf{s}(0))$. We say proof search *fails*, which will be reported back as the result of the computation, usually by printing `no`.

Since we think of the two rules as the complete definition of even we conclude that $\mathsf{even}(\mathsf{s}(0))$ is *false*. This example illustrates *negation as failure*, which is a common technique in logic programming. Notice, however, that there is an asymmetry: in the case where the conjecture was true, search constructed an explicit proof which provides evidence for its truth. In the case where the conjecture was false, no evidence for its falsehood is immediately available since all we can say is that we tried to find a proof in all possible ways and failed in each. This means that negation does not have first-class status in logic programming.

4 Answer Substitutions

In the first example the response to a goal is either `yes`, in which case a proof has been found, or `no`, if all attempts at finding a proof fail finitely. It is also possible that proof search does not terminate. But how can we write logic programs to compute values?

Since every natural number is either even or odd, the only expected answers are `yes` or `no` in that case. So let's look at an example where we actually expect a computed value as an answer. As an example we consider programs to compute sums and differences of natural numbers in the representation from the previous section. We start by specifying the underlying *relation* and then illustrate how it can be used for computation. The relation in this case is $\mathsf{plus}(m, n, p)$ which should hold if $m + n = p$. We use the recurrence

$$\begin{array}{lcl} (m+1)+n & = & (m+n)+1 \\ 0+n & = & n \end{array}$$

as our guide because it counts down the first argument to 0. We obtain

$$\frac{\mathsf{plus}(M, N, P)}{\mathsf{plus}(\mathsf{s}(M), N, \mathsf{s}(P))}\ \mathsf{ps} \qquad\qquad \frac{}{\mathsf{plus}(\mathsf{0}, N, N)}\ \mathsf{pz}.$$

Now consider a goal of the form $\mathsf{plus}(\mathsf{s}(\mathsf{0}), \mathsf{s}(\mathsf{0}), R)$ where R is an unknown. This represents the question if there exists an R such that the relation $\mathsf{plus}(\mathsf{s}(\mathsf{0}), \mathsf{s}(\mathsf{0}), R)$ holds. Search not only constructs a proof, but, with some bookkeeping, also a term t for R such that $\mathsf{plus}(\mathsf{s}(\mathsf{0}), \mathsf{s}(\mathsf{0}), t)$ is true.

For the original goal, $\mathsf{plus}(\mathsf{s}(\mathsf{0}), \mathsf{s}(\mathsf{0}), R)$, only the rule ps could apply because of a mismatch between $\mathsf{0}$ and $\mathsf{s}(\mathsf{0})$ in the first argument to plus in the conclusion. We also see that the R must have the form $\mathsf{s}(P)$ for some P, although we do not know yet what P should be.

$$\frac{\begin{array}{c}\vdots\\ \mathsf{plus}(0, \mathsf{s}(0), P)\end{array}}{\mathsf{plus}(\mathsf{s}(0), \mathsf{s}(0), R)}\ \mathsf{ps} \quad \text{with } R = \mathsf{s}(P)$$

For the subgoal only the pz rule applies and we see that P must equal $\mathsf{s}(0)$.

$$\text{proof search}\ \left\uparrow\ \frac{\dfrac{}{\mathsf{plus}(0, \mathsf{s}(0), P)}\ \mathsf{pz} \quad \text{with } P = \mathsf{s}(0)}{\mathsf{plus}(\mathsf{s}(0), \mathsf{s}(0), R)}\ \mathsf{ps} \quad \text{with } R = \mathsf{s}(P)\ \right\downarrow\ \text{substitute answers}$$

If we carry out the substitutions and put $P = \mathsf{s}(0)$ into $R = \mathsf{s}(P)$ giving $R = \mathsf{s}(\mathsf{s}(0))$, we obtain the complete proof

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

which is explicit evidence that $1 + 1 = 2$. Instead of the full proof, implementations of logic programming languages mostly just print the substitution for the unknowns in the original goal, in this case $R = \mathsf{s}(\mathsf{s}(0))$.

Some terminology of logic programming: the original goal is called the *query*, its unknowns are *logic variables*, and the result of the computation is an *answer substitution* for the logic variables, suppressing the proof.

5 Backtracking

Sometimes during proof search the goal matches the conclusion of more than one rule. This is called a *choice point*. When we reach a choice point we pick the first among the rules that match, in the order they were presented. If that attempt at a proof fails, we try the second one that matches, and so on. This process is called *backtracking*.

As an example, consider the query $\mathsf{plus}(M, \mathsf{s}(0), \mathsf{s}(\mathsf{s}(0)))$, intended to compute an m such that $m + 1 = 2$, that is, $m = 2 - 1$. This demonstrates that we can use the same logic program (here: the definition of the plus predicate) in different ways (before: addition, now: subtraction).

The conclusion of the rule pz, $\mathsf{plus}(0, N, N)$, does not match because the second and third argument of the query are different. However, the rule ps applies and we obtain

$$\dfrac{\begin{array}{c}\vdots\\ \mathsf{plus}(M_1, \mathsf{s}(0), \mathsf{s}(0))\end{array}}{\mathsf{plus}(M, \mathsf{s}(0), \mathsf{s}(\mathsf{s}(0)))}\;\mathsf{ps} \quad \text{with } M = \mathsf{s}(M_1)$$

At this point both rules, ps and pz, match. We use the rule ps because it is listed first, leading to

$$\dfrac{\dfrac{\begin{array}{c}\vdots\\ \mathsf{plus}(M_2, \mathsf{s}(0), 0)\end{array}}{\mathsf{plus}(M_1, \mathsf{s}(0), \mathsf{s}(0))}\;\mathsf{ps} \quad \text{with } M_1 = \mathsf{s}(M_2)}{\mathsf{plus}(M, \mathsf{s}(0), \mathsf{s}(\mathsf{s}(0)))}\;\mathsf{ps} \quad \text{with } M = \mathsf{s}(M_1)$$

At this point no rule applies at all and this attempt fails. So we return to our earlier choice point and try the second alternative, pz.

$$\dfrac{\dfrac{}{\mathsf{plus}(M_1, \mathsf{s}(0), \mathsf{s}(0))}\;\mathsf{pz} \quad \text{with } M_1 = 0}{\mathsf{plus}(M, \mathsf{s}(0), \mathsf{s}(\mathsf{s}(0)))}\;\mathsf{ps} \quad \text{with } M = \mathsf{s}(M_1)$$

At this point the proof is complete, with the answer substitution $M = \mathsf{s}(0)$.

Note that with even a tiny bit of foresight we could have avoided the failed attempt by picking the rule pz first. But even this small amount of ingenuity cannot be permitted: in order to have a satisfactory programming language we must follow every step prescribed by the search strategy.

6 Subgoal Order

Another kind of choice arises when an inference rule has multiple premises, namely the order in which we try to find a proof for them. Of course, logically the order should not be relevant because the final proof is a proof no matter in which order we went to find it. But operationally the behavior of a program can be quite different.

As an example, we define $\mathsf{times}(m, n, p)$ which should hold if $m \times n = p$. We implement the recurrence

$$\begin{array}{rcl} 0 \times n & = & 0 \\ (m+1) \times n & = & (m \times n) + n \end{array}$$

in the form of the following two inference rules.

$$\frac{}{\mathsf{times}(0, N, 0)}\ \mathsf{tz} \qquad\qquad \frac{\mathsf{times}(M, N, P) \quad \mathsf{plus}(P, N, Q)}{\mathsf{times}(\mathsf{s}(M), N, Q)}\ \mathsf{ts}$$

As an example we compute $1 \times 2 = Q$. The first step is determined.

$$\frac{\begin{array}{c}\vdots\\ \mathsf{times}(0, \mathsf{s}(\mathsf{s}(0)), P)\end{array} \quad \begin{array}{c}\vdots\\ \mathsf{plus}(P, \mathsf{s}(\mathsf{s}(0)), Q)\end{array}}{\mathsf{times}(\mathsf{s}(0), \mathsf{s}(\mathsf{s}(0)), Q)}\ \mathsf{ts}$$

Now if we solve the left subgoal first, there is only one applicable rule which forces $P = 0$

$$\frac{\dfrac{}{\mathsf{times}(0, \mathsf{s}(\mathsf{s}(0)), P)}\ \mathsf{tz}\ (P = 0) \quad \begin{array}{c}\vdots\\ \mathsf{plus}(P, \mathsf{s}(\mathsf{s}(0)), Q)\end{array}}{\mathsf{times}(\mathsf{s}(0), \mathsf{s}(\mathsf{s}(0)), Q)}\ \mathsf{ts}$$

Now since $P = 0$ from the first subgoal, which we, thus, know also for the second subgoal, there is only one rule that applies to the second subgoal and we obtain correctly

$$\frac{\dfrac{}{\mathsf{times}(0, \mathsf{s}(\mathsf{s}(0)), P)}\ \mathsf{tz}\ (P = 0) \quad \dfrac{}{\mathsf{plus}(P, \mathsf{s}(\mathsf{s}(0)), Q)}\ \mathsf{pz}\ (Q = \mathsf{s}(\mathsf{s}(0)))}{\mathsf{times}(\mathsf{s}(0), \mathsf{s}(\mathsf{s}(0)), Q)}\ \mathsf{ts}.$$

On the other hand, if we solve the right subgoal $\mathsf{plus}(P, \mathsf{s}(\mathsf{s}(0)), Q)$ first we have no information on P and Q, so both rules for plus apply. Since ps is given first, the strategy discussed in the previous section means that we try it first, which leads to

$$\frac{\begin{array}{c}\vdots\\ \mathsf{times}(0, \mathsf{s}(\mathsf{s}(0)), P)\end{array} \quad \dfrac{\begin{array}{c}\vdots\\ \mathsf{plus}(P_1, \mathsf{s}(\mathsf{s}(0)), Q_1)\end{array}}{\mathsf{plus}(P, \mathsf{s}(\mathsf{s}(0)), Q)}\ \mathsf{ps}\ (P = \mathsf{s}(P_1), Q = \mathsf{s}(Q_1))}{\mathsf{times}(\mathsf{s}(0), \mathsf{s}(\mathsf{s}(0)), Q)}\ \mathsf{ts}.$$

Again, rules ps and ts are both applicable, with ps listed first, so we continue:

$$
\dfrac{\dfrac{\vdots}{\mathsf{times}(0, \mathsf{s}(\mathsf{s}(0)), P)} \quad \dfrac{\dfrac{\dfrac{\vdots}{\mathsf{plus}(P_2, \mathsf{s}(\mathsf{s}(0)), Q_2)}}{\mathsf{plus}(P_1, \mathsf{s}(\mathsf{s}(0)), Q_1)}\ \mathsf{ps}\ (P_1 = \mathsf{s}(P_2), Q_1 = \mathsf{s}(Q_2))}{\mathsf{plus}(P, \mathsf{s}(\mathsf{s}(0)), Q)}\ \mathsf{ps}\ (P = \mathsf{s}(P_1), Q = \mathsf{s}(Q_1))}{\mathsf{times}(\mathsf{s}(0), \mathsf{s}(\mathsf{s}(0)), Q)}\ \mathsf{ts}
$$

It is easy to see that this will go on indefinitely, and computation will not terminate.

In fact, in light of the backtracking we observed here, we might want to reorder the rules so that pz comes before ps since pz gives short proofs. Likewise, the right premise of ts has two schema variables that are still unknown while the left premise has only one. That serves as a heuristic indication that tz might have the appropriate order. These are heuristic considerations, however, and a more detailed analysis is necessary to determine the computationally most suitable form.

This examples illustrate that the order in which subgoals are solved can have a strong impact on the computation. Here, proof search either completes in two steps or does not terminate. This is a consequence of fixing an operational reading for the rules. The standard solution is to attack the subgoals in left-to-right order. We observe here a common phenomenon of logic programming: two definitions, entirely equivalent from the logical point of view, can be very different operationally. Actually, this is also true for functional programming: two implementations of the same function can have very different complexity. This debunks the myth of "declarative programming"—the idea that we only need to specify the problem rather than design and implement an algorithm for its solution. However, we can assert that both specification and implementation can be expressed in the language of logic. As we will see later when we come to logical frameworks, we can integrate even correctness proofs into the same formalism!

7 Prolog Notation

By far the most widely used logic programming language is Prolog, which actually is a family of closely related languages. There are several good textbooks, language manuals, and language implementations, both free and commercial. A good resource is the FAQ of the Prolog newsgroup[4].

[4] https://groups.google.com/forum/#!forum/comp.lang.prolog

For this course we use GNU Prolog[5] although the programs should run in just about any Prolog since we avoid the more advanced features.

The two-dimensional presentation of inference rules does not lend itself to a textual format. The Prolog notation for a rule

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

is

$$J \leftarrow J_1, \ldots, J_n.$$

where the name of the rule is omitted and the left-pointing arrow is rendered as ':-' in a plain text file.

$$J :\!- J_1, \ldots, J_n.$$

We read this as

$$J \textit{ if } J_1 \textit{ and } \cdots \textit{ and } J_n.$$

Prolog terminology for an inference rule is a *clause*, where J is the *head* of the clause and $J_1, \ldots, J_n$ is the body. Therefore, instead of saying that we "*search for an inference rule whose conclusion matches the conjecture*", we say that we "*search for a clause whose head matches the goal*".

As an example, we show the earlier programs in Prolog notation.

```
even(z).
even(s(s(N))) :- even(N).

plus(s(M), N, s(P)) :- plus(M, N, P).
plus(z, N, N).

times(z, N, z).
times(s(M), N, Q) :-
        times(M, N, P),
        plus(P, N, Q).
```

Clauses are tried in the order they are presented in the program. Subgoals are solved in the order they are presented in the body of a clause.

[5] http://www.gprolog.org/

8 Unification

One important operation during search is to determine if the conjecture matches the conclusion of an inference rule (or, in logic programming terminology, if the goal unifies with the head of a clause). This operation is a bit subtle, because the the rule may contain schematic variables, and the the goal may also contain logical variables.

As a simple example (which we glossed over before), consider the goal

$$\mathsf{plus}(\mathsf{s}(0), \mathsf{s}(0), R)$$

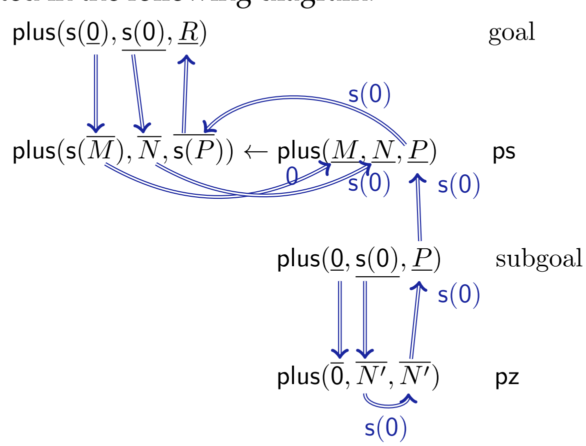
and the clause

$$\mathsf{plus}(\mathsf{s}(M), N, \mathsf{s}(P)) \leftarrow \mathsf{plus}(M, N, P)$$

We need to find some way to instantiate M, N, and P in the clause head and R in the goal such that $\mathsf{plus}(\mathsf{s}(0), \mathsf{s}(0), R) = \mathsf{plus}(\mathsf{s}(M), N, \mathsf{s}(P))$, by which we man that $\mathsf{plus}(\mathsf{s}(0), \mathsf{s}(0), R)$ and $\mathsf{plus}(\mathsf{s}(M), N, \mathsf{s}(P))$ become syntactically identical.

Without formally describing an algorithm yet, the intuitive idea is to match up corresponding subterms. If one of them is a variable, we set it to the other term. Here we set $M = 0$, $N = \mathsf{s}(0)$, and $R = \mathsf{s}(P)$. P is arbitrary and remains a variable. Applying these equations to the body of the clause we obtain $\mathsf{plus}(0, \mathsf{s}(0), P)$ which will be the subgoal with another logic variable, P.

In order to use the other clause for plus to solve this goal we have to solve $\mathsf{plus}(0, \mathsf{s}(0), P) = \mathsf{plus}(0, N, N)$ which sets $N = \mathsf{s}(0)$ and $P = \mathsf{s}(0)$. The basic idea behind unification and the intuitive order how it works in this case is illustrated in the following diagram:



This process is called *unification*, and the equations for the variables we generate represent the *unifier*. There are some subtle issues in unification. One is that the variables in the clause (which really are schematic variables in an inference rule) should be renamed to become fresh variables each time a clause is used so that the different instances of a rule are not confused with each other. Another issue is exemplified by the equation $N = \mathsf{s}(\mathsf{s}(N))$ which does not have a solution: the right-hand side will have have two more successors than the left-hand side so the two terms can never be equal. Unfortunately, Prolog does not properly account for this and treats such equations incorrectly by building a circular term (which is definitely not a part of the underlying logical foundation). This could come up if we pose the query $\mathsf{plus}(0, N, \mathsf{s}(N))$: *"Is there an n such that $0 + n = n + 1$."*

We discuss the reasons for Prolog's behavior later in this course (which is related to efficiency), although we do not subscribe to it because it subverts the logical meaning of programs.

We will come back to a full discussion of unification at a later lecture. For the moment, this intuitive account of unification will suffice for our purposes.

9 Beyond Prolog

Since logic programming rests on an operational interpretation of logic, we can study various logics as well as properties of proof search in these logics in order to understand logic programming. In this way we can push the paradigm to its limits without departing too far from what makes it beautiful: its elegant logical foundation.

Ironically, even though logic programming derives from logic, the language we have considered so far (which is the basis of Prolog) does not require any logical connectives at all, just the mechanisms of judgments and inference rules. Extensions of it do lead to logical connectives, though.

10 Historical Notes

Logic programming and the Prolog language are credited to Alain Colmerauer and Robert Kowalski in the early 1970s. Colmerauer had been working on a specialized theorem prover for natural language processing, which eventually evolved to a general purpose language called Prolog (for *Programmation en Logique*) that embodies the operational reading of clauses formulated by Kowalski. Interesting accounts of the birth of logic programming can be found in papers by the Colmerauer and Roussel [1] and Kowalski [2].

We like Sterling and Shapiro's *The Art of Prolog* [4] as a good introductory textbook for those who already know how to program and we recommends O'Keefe's *The Craft of Prolog* as a second book for those aspiring to become real Prolog hackers. Both of these are somewhat dated and do not cover many modern developments, which are the focus of this course. We therefore do not use them as textbooks here.

References

[1] Alain Colmerauer and Philippe Roussel. The birth of Prolog. In *Conference on the History of Programming Languages (HOPL-II), Preprints*, pages 37–52, Cambridge, Massachusetts, April 1993.

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

[3] Per Martin-Löf. On the meanings of the logical constants and the justifications of the logical laws. *Nordic Journal of Philosophical Logic*, 1(1):11–60, 1996.

[4] Leon Sterling and Ehud Shapiro. *The Art of Prolog*. The MIT Press, Cambridge, Massachusetts, 2nd edition edition, 1994.