15-819K: Logic Programming

Lecture 3

Induction

Frank Pfenning

September 5, 2006

One of the reasons we are interested in high-level programming languages is that, if properly designed, they allow rigorous reasoning about properties of programs. We can prove that our programs won't crash, or that they terminate, or that they satisfy given specifications. Logic programs are particularly amenable to formal reasoning.

In this lecture we explore induction, with an emphasis on *induction on the structure of deductions* sometimes called *rule induction* in order to prove properties of logic programs.

3.1 From Logical to Operational Meaning

A logic program has multiple interpretations. One is as a set of inference rules to deduce logical truths. Under this interpretation, the order in which the rules are written down, or the order in which the premisses to a rule are listed, are completely irrelevant: the true propositions and even the structure of the proofs remain the same. Another interpretation is as a program, where proof search follows a fixed strategy. As we have seen in prior lectures, both the order of the rules and the order of the premisses of the rules play a significant role and can make the difference between a terminating and a non-terminating computation and in the order in which answer substitutions are returned.

The different interpretations of logic programs are linked. The strength of that link depends on the presence or absence of purely operational constructs such as conditionals or cut, and on the details of the operational semantics that we have not yet discussed.

The most immediate property is *soundness* of the operational semantics: if a query A succeeds with a substitution θ, then the result of applying the substitution θ to A (written $A\theta$) is true under the logical semantics. In other words, $A\theta$ has a proof. This holds for pure logic programs but does not hold in the presence of logic variables together with negation-as-failure, as we have seen in the last lecture.

Another property is *completeness* of the operational semantics: if there is an instance of the query A that has a proof, then the query should succeed. This does not hold, since logic programs do not necessarily terminate even if there is a proof.

But there are some intermediate points. For example, the property of *non-deterministic completeness* says that if the interpreter were always allowed to choose which rule to use next rather than having to use the first applicable one, then the interpreter would be complete. Pure logic programs are complete in this sense. This is important because it allows us to interpret finite failure as falsehood: if the interpreter returns with the answer '`no`' it has explored all possible choices. Since none of them has led to a proof, and the interpreter is non-deterministically complete, we know that no proof can exist.

Later in the course we will more formally establish soundness and non-deterministic completeness for pure logic programs. It is relevant for this lecture, because when we want to reason about logic programs it is important to consider at which level of abstraction this reasoning takes place: Do we consider the logical meaning? Or the operational meaning including the backtracking behavior? Or perhaps the non-deterministic operational meaning? Making a mistake here could lead to a misinterpretation of the theorem we proved, or to a large amount of unnecessary work. We will point out such consequences as we go through various forms of reasoning.

3.2 Rule Induction

We begin by reasoning about the logical meaning of programs. As a simple example, we go back to the unary encoding of natural numbers from the first lecture. For reference we repeat the predicates for even and plus

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

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

Our aim is to prove that the sum of two even numbers is even. It is not immediately obvious how we can express this property on the relational specification. For example, we might say:

> *For any m, n, and p, if* $\mathsf{even}(m)$ *and* $\mathsf{even}(n)$ *and* $\mathsf{plus}(m, n, p)$ *then* $\mathsf{even}(p)$.

Or we could expressly require the existence of a sum p and the fact that it is even:

> *For any m, n, if* $\mathsf{even}(m)$ *and* $\mathsf{even}(n)$ *then there exists a p such that* $\mathsf{plus}(m, n, p)$ *and* $\mathsf{even}(p)$.

If we knew that plus is a total function in its first two arguments (that is, "*For any m and n there exists a unique p such that* $\mathsf{plus}(m, n, p)$."), then these two would be equivalent (see Exercise 3.2).

We will prove it in the second form. The first idea for this proof is usually to examine the definition of plus and see that it is defined structurally over its first argument m: the rule pz accounts for z and the rule ps reduces $\mathsf{s}(m)$ to m. This suggests an induction over m. However, in the predicate calculus (and therefore also in our logic programming language), m can be an arbitrary term and is therefore not a good candidate for induction.

Looking at the statement of the theorem, we see we are given the information that $\mathsf{even}(m)$. This means that we have a deduction of $\mathsf{even}(m)$ using only the two rules evz and evs, since we viewed these two rules as a complete definition of the predicate $\mathsf{even}(m)$. This licenses us to proceed by induction on the structure of the deduction of $\mathsf{even}(m)$. This is sometimes called *rule induction*. If we want to prove a property for all proofs of a judgment, we consider each rule in turn. We may assume the property for all premisses of the rule and have to show that it holds for the conclusion. If we can show this for all rules, we know the property must hold for all deductions.

In our proofs, we will need names for deductions. We use script letters $\mathcal{D}$, $\mathcal{E}$, and so on, to denote deduction and use the two-dimensional notation

$$\begin{array}{c}\mathcal{D}\\ J\end{array}$$

if $\mathcal{D}$ is a deduction of J.

Theorem 3.1 *For any m, n, if* $\mathsf{even}(m)$ *and* $\mathsf{even}(n)$ *then there exists a p such that* $\mathsf{plus}(m, n, p)$ *and* $\mathsf{even}(p)$.

Proof: By induction on the structure of the deduction $\mathcal{D}$ of $\mathsf{even}(m)$.

Case: $\mathcal{D} = \dfrac{}{\mathsf{even}(\mathsf{z})}\,\mathsf{evz}$ where $m = \mathsf{z}$.

$\mathsf{even}(n)$	Assumption
$\mathsf{plus}(\mathsf{z}, n, n)$	By rule pz
There exists p such that $\mathsf{plus}(\mathsf{z}, n, p)$ and $\mathsf{even}(p)$	Choosing $p = n$

Case: $\mathcal{D} = \dfrac{\begin{matrix}\mathcal{D}' \\ \mathsf{even}(m')\end{matrix}}{\mathsf{even}(\mathsf{s}(\mathsf{s}(m')))}\,\mathsf{evs}$ where $m = \mathsf{s}(\mathsf{s}(m'))$.

$\mathsf{even}(n)$	Assumption
$\mathsf{plus}(m', n, p')$ and $\mathsf{even}(p')$ for some p'	By ind. hyp. on $\mathcal{D}'$
$\mathsf{plus}(\mathsf{s}(m'), n, \mathsf{s}(p'))$	By rule ps
$\mathsf{plus}(\mathsf{s}(\mathsf{s}(m')), n, \mathsf{s}(\mathsf{s}(p')))$	By rule ps
$\mathsf{even}(\mathsf{s}(\mathsf{s}(p')))$	By rule evs
There exists p such that $\mathsf{plus}(\mathsf{s}(\mathsf{s}(m')), n, p)$ and $\mathsf{even}(p)$	
	Choosing $p = \mathsf{s}(\mathsf{s}(p'))$.

□

We have written here the proof in each case line-by-line, with a justification on the right-hand side. We will generally follow this style in this course, and you should arrange the answers to exercises in the same way because it makes proofs relatively easy to check.

3.3 Deductions and Proofs

One question that might come to mind is: Why did we have to carry out an inductive proof by hand in the first place? Isn't logic programming proof search according to a fixed strategy, so can't we get the operational semantics to do this proof for us?

Unfortunately, logic programming search has some severe restrictions so that it is usable as a programming language and has properties such as soundness and non-deterministic completeness. The restrictions are placed both on the forms of programs and the forms of queries. So far, in the logic that underlies Prolog, rules establish only atomic predicates. Furthermore, we can only form queries that are conjunctions of atomic propositions, possibly with some variables. This means that queries are purely *existential*: we asked whether there *exists* some instantiation of the variables so that there *exists* a proof for the resulting proposition as in the query `?- plus(s(z), s(s(z)), P)` where we simultaneously ask for a p and a proof of $\mathsf{plus}(\mathsf{s}(\mathsf{z}), \mathsf{s}(\mathsf{s}(\mathsf{z})), p)$.

On the other hand, our theorem above is primarily *universal* and only on the inside do we see an *existential* quantifier: *"For every m and n, and for every deduction $\mathcal{D}$ of $\mathsf{even}(m)$ and $\mathcal{E}$ of $\mathsf{even}(n)$ there exists a p and deductions $\mathcal{F}$ of $\mathsf{plus}(m, n, p)$ and $\mathcal{G}$ of $\mathsf{even}(p)$."*

This difference is also reflected in the structure of the proof. In response to a logic programming query we only use the inference rules defining the predicates directly. In the proof of the theorem about addition, we instead use induction in order to show that deductions of $\mathsf{plus}(m, n, p)$ and $\mathsf{even}(p)$ exist. If you carefully look at our proof, you will see that it contains a recipe for constructing these deductions from the given ones, but it does not construct them by backward search as in the operational semantics for logic programming. As we will see later in the course, it is in fact possible to represent the induction proof of our first theorem also in logic programming, although it cannot be found only by logic programming search.

We will make a strict separation between proofs using only the inference rules presented by the logic programmer and proofs *about* these rules. We will try to be consistent and write *deduction* for a proof constructed directly with the rules and *proof* for an argument about the logical or operational meaning of the rules. Similarly, we reserve the terms *proposition*, *goal*, and *query* for logic programs, and *theorem* for properties of logic programs.

3.4 Inversion

An important step in many induction proofs is *inversion*. The simplest form of inversion arises if have established that a certain proposition is true, and that the proposition matches the conclusion of only one rule. In that case we know that this rule must have been used, and that all premisses of the rule must also be true. More generally, if the proposition matches the conclusion of several rules, we can split the proof into cases, considering each one in turn.

However, great care must be taken with applying inversion. In my experience, the most frequent errors in proofs, both by students in courses such as this and in papers submitted to or even published in journals, are (a) missed cases that should have been considered, and (b) incorrect applications of inversion. We can apply inversion only if we already know that a judgment has a deduction, and then we have to take extreme care to make sure that we are indeed considering all cases.

As an example we prove that the list difference is uniquely determined, if it exists. As a reminder, the definition of append in rule form. We use the Prolog notation $[\,]$ for the empty list, and $[x|xs]$ for the list with head x and tails xs.

$$\frac{}{\mathsf{append}([\,], ys, ys)}\,\mathsf{apnil} \qquad \frac{\mathsf{append}(xs, ys, zs)}{\mathsf{append}([x|xs], ys, [x|zs])}\,\mathsf{apcons}$$

We express this in the following theorem.

Theorem 3.2 *For all xs and zs and for all ys and ys', if* $\mathsf{append}(xs, ys, zs)$ *and* $\mathsf{append}(xs, ys', zs)$ *then* $ys = ys'$.

Proof: By induction on the deduction $\mathcal{D}$ of $\mathsf{append}(xs, ys, zs)$. We use $\mathcal{E}$ to denote the given deduction $\mathsf{append}(xs, ys', zs)$.

Case: $\mathcal{D} = \dfrac{}{\mathsf{append}([\,], ys, ys)}$ where $xs = [\,]$ and $zs = ys$.

$\mathsf{append}([\,], ys', ys)$	Given deduction $\mathcal{E}$
$ys' = ys$	By inversion on $\mathcal{E}$ (rule apnil)

Case: $\mathcal{D} = \dfrac{\begin{array}{c}\mathcal{D}_1\\ \mathsf{append}(xs_1, ys, zs_1)\end{array}}{\mathsf{append}([x|xs_1], ys, [x|zs_1])}$ where $xs = [xs|xs_1]$, $zs = [xs|zs_1]$.

$\mathsf{append}([x|xs_1], ys', [x|zs_1])$	Given deduction $\mathcal{E}$
$\mathsf{append}(xs_1, ys', zs_1)$	By inversion on $\mathcal{E}$ (rule apcons)
$ys = ys'$	By ind. hyp. on $\mathcal{D}_1$

□

3.5 Operational Properties

We do not yet have formally described the operational semantics of logic programs. Therefore, we cannot prove operational properties completely rigorously, but we can come close by appealing to the intuitive semantics. Consider the following perhaps somewhat unfortunate specification of the predicate digit for decimal digits in unary notation, that is, natural numbers between 0 and 9.

$$\frac{}{\mathsf{digit}(\mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{z}))))))))))} \qquad \frac{\mathsf{digit}(\mathsf{s}(N))}{\mathsf{digit}(N)}$$

For example, we can deduce that z is a digit by using the second rule nine times (working bottom up) and then closing of the deduction with the first rule. In Prolog notation:

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

While logically correct, this does not work correctly as a decision procedure, because it will not terminate for any argument greater than 9.

Theorem 3.3 *Any query* `?- digit(`n`)` *for* $n > 9$ *will not terminate.*

Proof: By induction on the computation. If $n > 9$, then the first clause cannot apply. Therefore, the goal `digit(`n`)` must be reduced to the subgoal `digit(s(`n`))` by the second rule. But $\mathsf{s}(n) > 9$ if $n > 9$, so by induction hypothesis the subgoal will not terminate. Therefore the original goal also does not terminate. □

3.6 Aside: Forward Reasoning and Saturation

As mentioned in the first lecture, there is a completely different way to interpret inference rules as logic programs than the reading that underlies Prolog. This idea is to start with axioms (that is, inference rules with no premisses) as logical truths and apply all rules in the forward direction, adding more true propositions. We stop when any rule application that we could perform does not change the set of true propositions. In that case we say the database of true propositions is *saturated*. In order to answer a query we can now just look it up in the saturated database: if an instance of the query is in the database, we succeed, otherwise we fail.

In the example from above, we start with a database consisting only of digit(s(s(s(s(s(s(s(s(s(z)))))))))). We can apply the second rule with this as a premiss to conclude that digit(s(s(s(s(s(s(s(s(z))))))))). We can repeat this process a few more times until we finally conclude digit(z). At this point, any further rule applications would only add facts with already know: the set is saturated. We see that, consistent with the logical meaning, only the numbers 0 through 9 are digits, other numbers are not.

In this example, the saturation-based operational semantics via forward reasoning worked well for the given rules, while backward reasoning did not. There are classes of algorithms which appear to be easy to describe via saturation, that appear significantly more difficult with backward reasoning and vice versa. We will therefore return to forward reasoning and saturation later in the class, and also consider how it may be integrated with backward reasoning in a logical way.

3.7 Historical Notes

The idea to mix reasoning *about* rules with the usual logic programming search goes back to work by Eriksson and Hallnäs [2] which led to the GCLA logic programming language [1]. However, it stops short of supporting full induction. More recently, this line of development been revived by Tiu, Nadathur, and Miller [9]. Some of these ideas are embodied in the Bedwyr system currently under development.

Another approach is to keep the layers separate, but provide means to express proofs of properties of logic programs again as logic programs, as proposed by Schürmann [8]. These ideas are embodied in the Twelf system [6].

Saturating forward search has been the mainstay of the theorem proving community since the pioneering work on resolution by Robinson [7]. In logic programming, it has been called *bottom-up evaluation* and has historically been applied mostly in the context of databases [5] where saturation can often be guaranteed by language restrictions. Recently, it has been revisited as a tool for algorithm specification and complexity analysis by Ganzinger and McAllester [3, 4].

3.8 Exercises

The proofs requested below should be given in the style presented in these notes, with careful justification for each step of reasoning. If you need a lemma that has not yet been proven, carefully state and prove the lemma.

Exercise 3.1 *Prove that the sum of two even numbers is even in the first form given in these notes.*

Exercise 3.2 *Prove that* $\mathsf{plus}(m, n, p)$ *is a total function of its first two arguments and exploit this property to prove carefully that the two formulations of the property that the sum of two even numbers is even, are equivalent.*

Exercise 3.3 *Prove that* $\mathsf{times}(m, n, p)$ *is a total function of its first two arguments.*

Exercise 3.4 *Give a relational interpretation of the claim that "addition is commutative" and prove it.*

Exercise 3.5 *Prove that for any list xs,* $\mathsf{append}(xs, [\,], xs)$.

Exercise 3.6 *Give two alternative relational interpretations of the statement that "*append *is associative." Prove one of them.*

Exercise 3.7 *Write a logic program to reverse a given list and prove that when reversing the reversed list, we obtain the original list.*

Exercise 3.8 *Prove the correctness of quicksort from the previous lecture with respect to the specification from Exercise 2.2: If `quicksort(`xs`,` ys`)` is true then the second argument is an ordered permutation of the first. Your proof should be with respect to logic programs to check whether a list is ordered, and whether one list is a permutation of another.*

3.9 References

[1] M. Aronsson, L.-H. Eriksson, A. Gäredal, L. Hallnäs, and P. Olin. The programming language GCLA—a definitional approach to logic programming. *New Generation Computing*, 7(4):381–404, 1990.

[2] Lars-Henrik Eriksson and Lars Hallnäs. A programming calculus based on partial inductive definitions. SICS Research Report R88013, Swedish Institute of Computer Science, 1988.

[3] 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.

[4] 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.

[5] Jeff Naughton and Raghu Ramakrishnan. Bottom-up evaluation of logic programs. In J.-L. Lassez and G. Plotkin, editors, *Computational Logic. Essays in Honor of Alan Robinson*, pages 640–700. MIT Press, Cambridge, Massachusetts, 1991.

[6] 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.

[7] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, January 1965.

[8] Carsten Schürmann. *Automating the Meta Theory of Deductive Systems*. PhD thesis, Department of Computer Science, Carnegie Mellon University, August 2000. Available as Technical Report CMU-CS-00-146.

[9] Alwen Tiu, Gopalan Nadathur, and Dale Miller. Mixing finite success and finite failure in an automated prover. In C.Benzmüller, J.Harrison, and C.Schürmann, editors, *Proceedings of the Workshop on Empirically Successful Automated Reasnoing in Higher-Order Logics (ES-HOL'05)*, pages 79–98, Montego Bay, Jamaica, December 2005.