15-819K: Logic Programming

Lecture 20

Bottom-Up Logic Programming

Frank Pfenning

November 7, 2006

In this lecture we return to the view that a logic program is defined by a collection of inference rules for atomic propositions. But we now base the operational semantics on reasoning forward from facts, which are initially given as rules with no premisses. Every rule application potentially adds new facts. Whenever no more new facts can be generated we say forward reasoning *saturates* and we can answer questions about truth by examining the saturated database of facts. We illustrate bottom-up logic programming with several programs, including graph reachability, CKY parsing, and liveness analysis.

20.1 Bottom-Up Inference

We now return the very origins of logic programming as an operational interpretation of inference rules defining atomic predicates. As a reminder, consider the definition of even.

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

This works very well on queries such as $\mathsf{even}(\mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{z}))))))$ (which succeeds) and $\mathsf{even}(\mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{z})))))$ (which fails). In fact, the operational reading of this program under goal-directed search constitutes a decision procedure for ground queries $\mathsf{even}(n)$.

This specification makes little sense under an alternative interpretation where we eagerly apply the inference rules in the forward direction, from the premisses to the conclusion, until no new facts can be deduced. The

problem is that we start with $\mathsf{even}(\mathsf{z})$, then obtain $\mathsf{even}(\mathsf{s}(\mathsf{s}(\mathsf{z})))$, and so on, but we never terminate.

It would be too early to give up on forward reasoning at this point. As we have seen many times, even in backward reasoning a natural specification of a predicate does not necessarily lead to a reasonable implementation. We can implement a test whether a number is even via reasoning by contradication. We seed our database with the claim that n is not even and derive consequences from that assumption. If we derive a contradictory fact we know thate $\mathsf{even}(n)$ must be true. If not (and our rules are complete), then $\mathsf{even}(n)$ must be false. We write $\mathsf{odd}(n)$ for the proposition that n is not even. Then we obtain the following specification

$$\frac{\mathsf{odd}(\mathsf{s}(\mathsf{s}(N)))}{\mathsf{odd}(N)}$$

to be used for forward reasoning. This single rule obviously saturates because the argument to odd becomes smaller in every rule application.

What is not formally represented in this program is how we initialize our database (we assume $\mathsf{odd}(n)$), and how we interpret the saturated database (we check if $\mathsf{odd}(\mathsf{z})$ was deduced). In a later lecture we will see that it is possible to combine forward and backward reasoning to makes those aspects of an algorithm also part of its implementation.

The strategy of this example, proof by contradiction, does not always work, but there are many cases where it does. One should check if the predicate is decidable as a first test. We will see further examples later, specifically the treatment of unification in the next lecture.

20.2 Graph Reachability

Assuming we have a specification of $\mathsf{edge}(x, y)$ whenever there is an edge from node x to node y, we can specify reachability $\mathsf{path}(x, y)$ with the rules

$$\frac{\mathsf{edge}(X, Y)}{\mathsf{path}(X, Y)} \qquad \frac{\mathsf{edge}(X, Y) \quad \mathsf{path}(Y, Z)}{\mathsf{path}(X, Z)}$$

During bottom-up inference these rules will saturate when they have constructed the transitive closure of the edge relation. During backward reasoning these rules may not terminate (if there are cycles), or be very inefficient (if there are many paths compared to the number of nodes).

In the forward direction the rules will always saturate. We can also give, just from the rules, a complexity analysis of the saturation algorithm.

20.3 Complexity Analysis

McAllester [3] proved a so-called meta-complexity result which allows us to analyze the structure of a bottom-up logic program and obtain a bound for its asymptotic complexity. We do not review the result or its proof in full detail here, but we sketch it so it can be applied to several of the programs we consider here. Briefly, the result states that the complexity of a bottom-up logic program is $O(|R(D)| + |P_R(R(D))|)$, where $R(D)$ is the saturated database (writing here D for the initial database) and $P_R(R(D))$ is the set of prefix firings of rules R in the saturated database.

The number *prefix firings* for a given rule is computed by analyzing the premisses of the rule from left to right, counting in how many ways it could match facts in the saturated database. Matching an earlier premiss will fix its variables, which restricts the number of possible matches for later premisses.

For example, in the case of the transitive closure program, assume we have e edges and n vertices. Then in the completed database there can be at most n^2 facts $\mathsf{path}(x, y)$, while there are always exactly e facts $\mathsf{edge}(x, y)$. The first rule

$$\frac{\mathsf{edge}(X, Y)}{\mathsf{path}(X, Y)}$$

can therefore always match in e ways in the completed database. We analyze the premisses of the second rule

$$\frac{\mathsf{edge}(X, Y) \quad \mathsf{path}(Y, Z)}{\mathsf{path}(X, Z)}$$

from left to right. First, $\mathsf{edge}(X, Y)$ can match the database in $O(e)$ ways, as before. This match fixes Y, so there are now $O(n)$ ways that the second premiss could match a fact in the saturated database (each vertex is a candidate for Z). This yields $O(e \cdot n)$ possible prefix firings.

The size of the saturated database is $O(e+n^2)$, and the number of prefix firings of the two rules is $O(e + e \cdot n)$. Therefore the overall complexity is $O(e \cdot n + n^2)$. Since there are up to n^2 edges in the graph, we get a less informative bound of $O(n^3)$ expressed entirely in the number of vertices n.

20.4 CKY Parsing

Another excellent example for bottom-up logic programming and complexity analysis is a CKY parsing algorithm. This algorithm assumes that

the grammar is in Chomsky-normal form, where productions all have the form

$$\begin{array}{lcl} x & \Rightarrow & y\,z \\ x & \Rightarrow & a \end{array}$$

where x, y, and z stand for non-terminals and a for terminal symbols. The idea of the algorithm is to use the grammar production rules from right to left to compute which sections of the input string can be parsed as which non-terminals.

We initialize the database with facts $\mathsf{rule}(x, \mathsf{char}(a))$ for every grammar production $x \Rightarrow a$ and $\mathsf{rule}(x, \mathsf{jux}(y, z))$ for every production $x \Rightarrow y\,z$. We further represent the input string $a_1 \ldots a_n$ by assumptions $\mathsf{string}(i, a_i)$. For simplicity, we represent numbers in unary form.

Our rules will infer propositions $\mathsf{parse}(x, i, j)$ which we will deduce if the substring $a_i \ldots a_j$ can be parsed as an x. Then the program is represented by the following two rules, to be read in the forward direction:

$$\frac{\mathsf{rule}(X, \mathsf{char}(A)) \quad \mathsf{string}(I, A)}{\mathsf{parse}(X, I, I)} \qquad \frac{\mathsf{rule}(X, \mathsf{jux}(Y, Z)) \quad \mathsf{parse}(Y, I, J) \quad \mathsf{parse}(Z, s(J), K)}{\mathsf{parse}(X, I, K)}$$

After saturating the database with these rules we can see if the whole string is in the language generated by the start symbol s by checking if the fact $\mathsf{parse}(s, \mathsf{s}(\mathsf{z}), n)$ is in the database.

Let g be the number of grammar productions and n the length of the input string. In the completed database we have g grammar rules, n facts $\mathsf{string}(i, a)$, and at most $O(g \cdot n^2)$ facts $\mathsf{parse}(x, i, j)$.

Moving on to the rules, in the first rule there are $O(g)$ ways to match the grammar rule (which fixes A) and then n ways to match $\mathsf{string}(I, A)$, so we have $O(g \cdot n)$. The second rule, again we have $O(g)$ ways to match the grammar rule (which fixes X, Y, and Z) and then $O(n^2)$ ways to match $\mathsf{parse}(Y, I, J)$. In the third premiss now only K is unknown, giving us $O(n)$ way to match it, which means $O(g \cdot n^3)$ prefix firings for the second rule.

These considerations give us an overall complexity of $O(g \cdot n^3)$, which is also the traditional complexity bound for CKY parsing.

20.5 Liveness Analysis

We consider an application of bottom-up logic programming in program analysis. In this example we analyze code in a compiler's intermediate

language to find out which variables are live or dead at various points in the program. We say a variable is *live* at a given program point l if its value will be read before it is written when computation reaches l. This information can be used for optimization and register allocation.

Every command in the language is labeled by an address, which we assume to be a natural number. We use l and k for labels and w, x, y, and z for variables, and op for binary operators. In this stripped-down language we have the following kind of instructions. A representation of the instruction as a logical term is given on the right, although we will continue to use the concrete syntax to make the rules easier to read.

l	:	$x = op(y, z)$	$\mathsf{inst}(l, \mathsf{assign}(x, op, y, z))$
l	:	if x goto k	$\mathsf{inst}(l, \mathsf{if}(x, k))$
l	:	goto k	$\mathsf{inst}(l, \mathsf{goto}(k))$
l	:	halt	$\mathsf{inst}(l, \mathsf{halt})$

We use the proposition $x \neq y$ to check if two variables are distinct and write $\mathsf{s}(l)$ for the successor location to l which contains the next instruction to be executed unless the usual control flow is interrupted.

We write $\mathsf{live}(w, l)$ if we have inferred that variable w is live at l. This is an over-approximation in the sense that $\mathsf{live}(w, l)$ indicates that the variable *may* be live at l, although it is not guaranteed to be read before it is written. This means that any variable that is not live at a given program point is definitely *dead*, which is the information we want to exploit for optimization and register allocation.

We begin with the rules for assignment $x = op(y, z)$. The first two rules just note the use of variables as arguments to an operator. The third one propagates liveness information backwards through the assignment operator. This is sound for any variable, but we would like to achieve that x is not seen as live before the instruction $x = op(y, z)$, so we verify that $W \neq X$.

$$\frac{L : X = Op(Y, Z)}{\mathsf{live}(Y, L)} \qquad \frac{L : X = Op(Y, Z)}{\mathsf{live}(Z, L)} \qquad \frac{\begin{array}{l} L : X = Op(Y, Z) \\ \mathsf{live}(W, \mathsf{s}(L)) \\ W \neq X \end{array}}{\mathsf{live}(W, L)}$$

The rules for jumps propagate liveness information backwards. For unconditional jumps we look at the target; for conditional jumps we look both at the target and the next statement, since we don't analyze whether the

condition may be true or false.

$$\frac{L : \mathsf{goto}\ K \quad \mathsf{live}(W, K)}{\mathsf{live}(W, L)} \qquad \frac{L : \mathsf{if}\ X\ \mathsf{goto}\ K \quad \mathsf{live}(W, K)}{\mathsf{live}(W, L)} \qquad \frac{L : \mathsf{if}\ X\ \mathsf{goto}\ K \quad \mathsf{live}(W, \mathsf{s}(L))}{\mathsf{live}(W, L)}$$

Finally, the variable tested in a conditional is live.

$$\frac{L : \mathsf{if}\ X\ \mathsf{goto}\ K}{\mathsf{live}(X, L)}$$

For the complexity analysis, let n be the number of instructions in the program and v be the number of variables. The size of the saturated database is $O(v \cdot n)$, since all derived facts have the form $\mathsf{live}(X, L)$ where X is a variable and L is the label of an instruction. The prefix firings of all 7 rules are similarly bounded by $O(v \cdot n)$: there are n ways to match the first instruction and then at most v ways to match the second premiss (if any). Hence the overall complexity is bounded by $O(v \cdot n)$.

20.6 Functional Evaluation

As a last example in this lecture we present an algorithm for functional call-by-value evaluation. Our language is defined by

$$e ::= x \mid \lambda x.\, e \mid e_1\, e_2$$

We assume that substitution on terms is a primitive so we can avoid implementing it explicitly (see Exercise 20.4). Such an assumption is not unreasonable. For example, in the LolliMon language which provides both top-down and bottom-up logic programming, substitution is indeed built-in. Since we are only interested in evaluating closed terms, all values here have the form $\lambda x.\, e$.

We use three predicates:

$\mathsf{eval}(e)$	evaluate e
$e \rightarrow^* e'$	e reduces to e'
$e \hookrightarrow v$	e evaluates to v.

We seed the database with $\mathsf{eval}(e)$, saturate it, and then read off the value as $e \hookrightarrow v$. Of course, since this is the untyped λ-calculus, saturation is not guaranteed.

The first rules propagate the information about which terms are to be evaluated.

$$\frac{\mathsf{eval}(\lambda x.\, e)}{\lambda x.\, e \hookrightarrow \lambda x.\, e} \qquad \frac{\mathsf{eval}(e_1\, e_2)}{\mathsf{eval}(e_1)} \quad \frac{\mathsf{eval}(e_1\, e_2)}{\mathsf{eval}(e_2)}$$

In case we had an application we have to gather the results, substitute the argument into the body of the function, and recursively evaluate the result. This generates a reduction from which we need to initiate evaluation. Finally, we need to compose reductions to obtain the final value.

$$\frac{\begin{array}{l}\mathsf{eval}(e_1\, e_2)\\ e_1 \hookrightarrow \lambda x.\, e_1'\\ e_2 \hookrightarrow v_2\end{array}}{e_1\, e_2 \longrightarrow^* e_1'(v_2/x)} \qquad \frac{e \longrightarrow^* e'}{\mathsf{eval}(e')} \qquad \frac{e \longrightarrow^* e' \quad e' \hookrightarrow v}{e \hookrightarrow v}$$

As an example, consider the following database saturation process.

$$\begin{array}{l}\mathsf{eval}((\lambda x.\, x)\, (\lambda y.\, y))\\ \mathsf{eval}(\lambda x.\, x)\\ \mathsf{eval}(\lambda y.\, y)\\ \lambda x.\, x \hookrightarrow \lambda x.\, x\\ \lambda y.\, y \hookrightarrow \lambda y.\, y\\ (\lambda x.\, x)\, (\lambda y.\, y) \longrightarrow^* \lambda y.\, y\\ (\lambda x.\, x)\, (\lambda y.\, y) \hookrightarrow \lambda y.\, y\end{array}$$

This form of evaluation may seem a bit odd, compared to the usual top-down formulation (again, assuming substitution as a primitive)

$$\frac{}{\lambda x.\, e \hookrightarrow \lambda x.\, e} \qquad \frac{e_1 \hookrightarrow \lambda x.\, e_1' \quad e_2 \hookrightarrow v_2 \quad e_1'(v_2/x) \hookrightarrow v}{e_1\, e_2 \hookrightarrow v}$$

However, it does have some advantages. If we proved its completeness (see Exercise 20.5), we would get some theorems for free. For example, it is easy to see that $\mathsf{eval}((\lambda x.\, x\, x)\, (\lambda x.\, x\, x))$ saturates without producing a value for the application:

$$\begin{array}{l}\mathsf{eval}((\lambda x.\, x\, x)\, (\lambda x.\, x\, x))\\ \mathsf{eval}(\lambda x.\, x\, x)\\ (\lambda x.\, x\, x) \hookrightarrow (\lambda x.\, x\, x)\\ (\lambda x.\, x\, x)\, (\lambda x.\, x\, x) \longrightarrow^* (\lambda x.\, x\, x)\, (\lambda x.\, x\, x)\end{array}$$

At this point the database is saturated. This proves that the evaluation of $(\lambda x.\, x\, x)\, (\lambda x.\, x\, x)$ fails. In the top-down semantics (that is, with backward chaining as in Prolog), such a query would fail to terminate instead unless we added some kind of loop detection.

Note that the bottom-up program for evaluation, which consists of six rules, cannot be analyzed with McAllester's technique, because in the conclusion of the rule for reduction a new term $e_1'(v_2/x)$ is created. We can therefore not bound the size of the completed database. And, in fact, the saturation may fail to terminate (see Exercise 20.7).

20.7 Variable Restrictions

Bottom-up logic programming, as considered by McAllester, requires that every variable in the conclusion of a rule also appears in a premiss. This means that every generated fact will be ground. This is important for saturation and complexity analysis because a fact with a free variable could stand for infinitely many instances.

Nonetheless, bottom-up logic programming can be generalized in the presence of free variables and we will do this in a later lecture.

20.8 Historical Notes

The bottom-up interpretation of logic programs goes back to the early days of logic programming. See, for example, the paper by Naughton and Ramakrishnan [4].

There are at least three areas were logic programming specification with a bottom-up semantics has found significant applications: deductive databases, decision procedures, and program analysis. Unification, as present in the next lecture, is an example of a decision procedure for unifiability. Liveness analysis is an example of program analysis due to McAllester [3], who was particularly interested in describing program analysis algorithms at a high level of abstraction so their complexity would be self-evident. This was later refined by Ganzinger and McAllester [1, 2] by allowing deletions in the database. We treat this in a later lecture where we generalize bottom-up inference to linear logic.

20.9 Exercises

Exercise 20.1 *Write a bottom-up logic program for addition (`plus/3`) on numbers in unary form and then extend it to multiplication (`times/3`).*

Exercise 20.2 *Consider the following variant of graph reachability.*

$$\frac{\mathsf{edge}(X,Y)}{\mathsf{path}(X,Y)} \qquad \frac{\mathsf{path}(X,Y) \quad \mathsf{path}(Y,Z)}{\mathsf{path}(X,Z)}$$

Perform a McAllester-style complexity analysis and compare the infered complexity with the one given in lecture.

Exercise 20.3 *The set of prefix firings depends on the order of the premisses. Give an example to demonstrate this.*

Exercise 20.4 *Extend the bottom-up evaluation semantics for λ-terms by adding rules to compute the substitutions $e(v/x)$. You may assume that v is closed, and that the necessary tests on variable names can be performed.*

Exercise 20.5 *Relate the bottom-up and top-down version of evaluation of λ-terms to each other by an appropriate pair of theorems.*

Exercise 20.6 *Add pairs to the evaluation semantics, together with first and second projections. A pair should only be a value if both components are values, that is, pairs are eagerly evaluated.*

Exercise 20.7 *Give an example which shows that saturation of evaluation for λ-terms may fail to terminate.*

20.10 References

[1] 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.

[2] 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.

[3] Dave McAllester. On the complexity analysis of static analyses. *Journal of the ACM*, 49(4):512–537, 2002.

[4] 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.