Lecture Notes on Forward Logic Programming

15-317: Constructive Logic
Frank Pfenning *

Lecture 21
November 17, 2015

1 Introduction

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. Contrast this with the operational semantics of backwards or goal-directed logic programming, which went backwards from the goal by successively reducing it by an applicable rule in a deterministic order. And recall how resolution proofs had (at least) two ways of being read and constructed: forward from the facts or backwards from the goal.

Every forward 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 this so-called bottom-up logic programming (alias forward logic programming) with several programs, including graph reachability, context-free grammar parsing, and liveness analysis. It turns out that forward logic programming provides an exceedingly elegant way of reading off complexity results for free from a direct analysis of each rule.

*With edits by André Platzer

2 Bottom-Up Inference

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

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

This works very well on queries such as even(s(s(s(s(0))))) (which succeeds) and even(s(s(s(0)))) (which fails). In fact, the operational reading of this program under goal-directed search constitutes a decision procedure for ground queries 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 even(0), then obtain even(s(s(0))), 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 even(n) must be true. If not (and our rules are complete), then even(n) must be false. We write 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 odd(n)), and how we interpret the saturated database (we check if odd(0) was deduced which would be a contradiction). In a later lecture we will see that it is possible to combine forward and backward reasoning to make 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 can also try to stick with the original predicate for even, but add another argument which is a bound to guarantee saturation. The idea is that, during saturation in search for an input $\mathsf{s}^n(0)$—by which we mean the function s nested n times—we are actually only interested in all facts about numbers of at most that magnitude. We count the bound down, two at a time, in the second argument while the first argument computes even numbers.

$$\frac{\mathsf{even}(N, \mathsf{s}(\mathsf{s}(B)))}{\mathsf{even}(\mathsf{s}(\mathsf{s}(N)), B)}\ \mathsf{evss}$$

When trying to check if a number n is even, we seed the database with

$$\mathsf{even}(\mathsf{0}, n)$$

which expresses that 0 is considered even no matter what the bound (n in this case). Operationally, n is the bound, while 0 is the first even number.

After saturation, which is guaranteed to happen after $\lfloor n/2 \rfloor$ steps, we check if some fact of the form

$$\mathsf{even}(n, \_)$$

is in the database. Such bounds or sometimes other simple structural restrictions to make sure that the facts generated during saturation are on the critical path toward the actual goal can be quite helpful to reduce the time till saturation.

By an invariant of the evss algorithm, all deduced facts $\mathsf{even}(b, n)$ have the same sum $b + n$, since the first argument is increased by two s applications while the second argument is decreased correspondingly. So we actually know that if n is even we must actually have $\mathsf{even}(n, \mathsf{0})$ at the end, and $\mathsf{even}(n-1, \mathsf{s}(\mathsf{0}))$ if n is odd, leading to a decision procedure. This can be used to eliminate the *first* argument, arriving at more or less the code for odd shown above, but with a different justification.

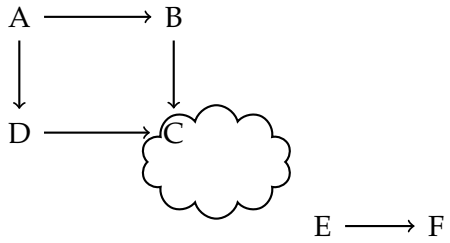
3 Graph Reachability

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

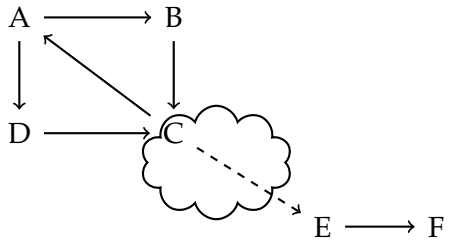
$$\frac{\mathsf{edge}(X, Y)}{\mathsf{path}(X, Y)} \qquad \frac{\mathsf{edge}(X, Y) \quad \mathsf{path}(Y, Z)}{\mathsf{path}(X, Z)}$$

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 this graph, in which the cloud is supposed to illustrate some big subgraph with many connections, the edge $A \to B$ from A to B and then to C will be tried first (the precise order depends on clause order), with a long failed path search from C to E, which cannot work, because C and E are not connected. Afterwards, Prolog's backtracking will consider the edge $A \to D$ and then $D \to C$, which will repeatedly set out on the long but pointless search for a path toward E through the cloud starting at C in vain.



In the following cyclic graph, the search will not even terminate, even if there is a path from C to E, since the cyclic edge from C to A will be considered first, recursively leading to $A \to B \to C \to A$ infinitely often.



In the forward direction the rules will always saturate. Once forward reasoning discovered $path(A, C)$ by one path $A \to B \to C$, it will not have a second need to explore the consequences of the same fact again, just because it found a second path $A \to D \to C$. In a sense, advantages of forward logic programming over backward logic programming are that it comes with built-in memoization and ways of determining when the computation is done with producing new facts. We can also give, just from the rules, a complexity analysis of the saturation algorithm.

But first, let's examine how forward logic programming proof search produces facts in the second graph by exhaustively applying all rules to facts:

```
% first rule applied to edge
1  path(A,B)
2  path(B,C)
3  path(C,A)
4  path(A,D)
5  path(D,C)
6  path(C,E)
7  path(E,F)
% second rule applied to edge
8  path(C,B)   % from 1
9  path(A,C)   % from 2
...
15  path(B,E)   % from 6
16  path(C,F)   % from 7
...
20  path(A,E)   % from 15
...
```

At this point, the goal `path(A,E)` has been put into the database of facts, so we can return the answer `yes`. Note that, quite unlike goal-oriented backward logic programming, forward logic programming generates such a database of facts. In particular, if a fact such as `path(A,C)` is generated twice (once from $A \to B \to C$ and once from $A \to D \to C$), we still only add it once since its consequences in terms of the applicable rules are going to be the same no matter how often fact has been derived. That gives forward logic programming the advantages of built-in memoization to avoid repetitive exploration of search spaces.

4 Complexity Analysis

McAllester [McA02] 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 by a simple analysis of its structure. 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.

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	$\mathsf{inst}(l, \mathsf{halt})$

This language is stripped-down but computationally unversal and still a close-enough approximation of the ideas behind intermediate languages in compilers. Here is an example of a program along with all live variables at the respective lines:

		Instructions	Live variables	
1	:	if y goto 8	y, x	
2	:	$q = x/y$	x, y	
3	:	$t = q * y$	x, y, q	
4	:	$r = x - t$	x, y, t	
5	:	$x = y$	y, r	
6	:	$y = r$	r	x
7	:	goto 1		x, y
8	:	$x = x + 0$	x	
9	:	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, which we cannot generally know anyhow.

$$\frac{\begin{array}{l} L : \mathsf{goto}\ K \\ \mathsf{live}(W, K) \end{array}}{\mathsf{live}(W, L)} \qquad \frac{\begin{array}{l} L : \mathsf{if}\ X\ \mathsf{goto}\ K \\ \mathsf{live}(W, K) \end{array}}{\mathsf{live}(W, L)} \qquad \frac{\begin{array}{l} L : \mathsf{if}\ X\ \mathsf{goto}\ K \\ \mathsf{live}(W, \mathsf{s}(L)) \end{array}}{\mathsf{live}(W, L)}$$

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 since there are n lines in the program and then at most v ways to match the second premiss (if any) since the respective code line has already been matched in the first premiss. Hence the overall complexity is bounded by $O(v \cdot n)$.

6 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, because, inductively, each rule will only have its premisses applied to ground facts and (by variable inclusion) its conclusion is ground since it will only involve variables that are already unified with ground terms. 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, but we will not discuss this further in this course.

7 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 [NR91].

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. The unification rules we saw in an earlier lecture are an example of a decision procedure for unifiability. Liveness analysis is an example of program analysis first formulated in this fashion by McAllester [McA02], 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 [GM01, GM02] by allowing deletions in the database. We treat this in a later lecture where we generalize bottom-up inference to linear logic.

8 Exercises

Exercise 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 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 3 *The set of prefix firings depends on the order of the premisses. Give an example to demonstrate this.*

References

[GM01] 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.

[GM02] 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.

[McA02] David A. McAllester. On the complexity analysis of static analyses. *Journal of the ACM*, 49(4):512–537, 2002.

[NR91] 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.