Forward Logic Programming

Constructive Logic (15-317)
Instructor: Giselle Reis

Lecture 17

1 Introduction

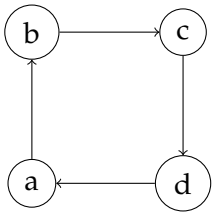
Logic programs can be executed in different ways. Backward chaining (or top-down evaluation) consists on matching the query with heads of clauses, generating new sub-goals, until all goals are solved. Forward chaining (or bottom-up evaluation) consists on generating new facts iteratively from the set of known facts until *saturation*: no new facts can be generated. These new facts are generated by matching the known facts with bodies of clauses and adding the (unified) head to the knowledge database (i.e., the list of known facts).

Take for example the following program:

```
path(X,Y) :- edge(X,Y).
path(X,Y) :- edge(X,Z), path(Z,Y).

cycle :- path(X,X).
```

The two first rules define when there is a path between two nodes in a graph. The last rule states that if there is a path from a node to itself, then the graph has a cycle. Let's see what happens if we add to it a description (on the right) of the graph (on the left):




```
edge(a,b).
edge(b,c).
edge(c,d).
edge(d,a).
```

A forward chaining engine will greedily start generating new facts. It will match each `edge` information with the body of the first clause, generating the set:

```
path(a,b), path(b,c), path(c,d), path(d,a)
```

Now that the database contains both `edge` and `path` facts, it can solve the body of the second clause, generating the new paths:

```
path(a,a), path(a,c), path(a,d), path(b,a), path(b,b), path(b,d),
path(c,a), path(c,b), path(c,c), path(d,b), path(d,c), path(d,d)
```

Since the database now contains paths from a vertex to itself, the last rule can be fired, generating the `cycle` token. At this point no new information can be obtained, and the forward computation stops. Two important things to keep in mind:

- Facts are always *ground* atoms, meaning that there are no free variables lingering around. For this reason, rules used in forward logic programming require that all variables occurring in the head must also occur in the body.
- The database is considered as a set, so the multiplicity of facts does not matter. Even if there are infinitely many ways to obtain a cycle or a path from a to c in the graph above, only one atom for each information is generated.

The programs used in this lecture were tested using dlv[1]. Assuming the rules and graph description are in a file called `graph.dlv`, the result of running `dlv graph.dlv` is the following:

```
DLV [build BEN/Dec 17 2012   gcc 4.6.1]

{edge(a,b), edge(b,c), edge(c,d), edge(d,a), path(a,a), path(a,b),
 path(a,c), path(a,d), path(b,a), path(b,b), path(b,c), path(b,d),
 path(c,a), path(c,b), path(c,c), path(c,d), path(d,a), path(d,b),
 path(d,c), path(d,d), cycle}
```

If the program generates an infinite database, then it will run forever. Although dlv has some checks in place to detect non-terminating rules, it is not difficult to implement something that runs forever. But then again, it is also not difficult to implement something that runs forever in Prolog (or in any other programming language) ;)

Remark 1. *When bottom-up logic programming was first introduced, there were methods to transform Prolog programs such that they would behave more nicely using this new evaluation order (e.g. [2]).*

[1] https://dlv.demacs.unical.it/

Querying a forward logic program can be accomplished by contradiction. We can write a rule specifying that if an atom is true, the program should fail (represented by an empty head). So if no output is generated, we know the query succeeded. Querying whether the graph has a cycle can done by adding the following clause to the program:

```
:- cycle.
```

In which case no answer is generated upon executing it.

Since we already know the proof theoretical interpretation of forward logic programming, we will see a nice application of this technique to parsing[2].

2 Parsing as a Logic Program

Parsing a sentence (i.e., a sequence of strings) consists of deciding whether this sentence belongs to a *grammar*. A grammar, on its turn, is a set of production rules that produce the sentences of a language. This being a course in computer science, we will concern ourselves with *context-free* grammars.

Definition 1. *A* context-free grammar *is a tuple $\langle V, \Sigma, \mathcal{P}, S\rangle$ where V is a set of non-terminal symbols (or variables), Σ is a set of terminal symbols, $\mathcal{P}$ is a set of production rules of the form $X \Rightarrow \gamma$ ($X \in V$ and γ is a string with other non-terminals or terminals), and $S \in V$ is the start symbol.*

This grammar is called context-free because the variable X may be replaced according to a production rule independently on the context where it occurs in the string.

As an example, take the following grammar for linear arithmetic.

$$
\begin{array}{lcll}
[\text{zero}] & T & \Rightarrow & \texttt{0} \\
[\text{one}] & T & \Rightarrow & \texttt{1} \\
[\text{var}] & T & \Rightarrow & \texttt{var} \\
[\text{plus}] & T & \Rightarrow & T + T \\
[\text{neg}] & T & \Rightarrow & -T \\
[\text{pars}] & T & \Rightarrow & (T)
\end{array}
$$

Each production rule is labelled for reference. By `var` we denote any identifier for an arithmetic variable, such as `x`, `y`, `z`, etc.

A derivation of a string w containing only terminals is a sequence of production rule applications $T \Rightarrow \gamma_1 ... \Rightarrow \gamma_n \Rightarrow w$. Each step consists of choosing a non-terminal X from γ_i and a production rule $X \Rightarrow \alpha$, and replacing that occurrence of X in γ_i by α thus

[2] Credits to André Platzer and Frank Pfenning. More examples can be found at http://www.cs.cmu.edu/~fp/courses/lp/lectures/20-bottomup.pdf.

obtaining γ_{i+1}. If w can be derived in a grammar G, then we say that w is in the language of G ($L(G)$).

The string `-(x+1)` is in the language of the grammar above:

$$\begin{array}{rcll} T & \Rightarrow & -T & [\text{neg}] \\ & \Rightarrow & -(T) & [\text{pars}] \\ & \Rightarrow & -(T+T) & [\text{plus}] \\ & \Rightarrow & -(\texttt{x}+T) & [\text{var}] \\ & \Rightarrow & -(\texttt{x}+\texttt{1}) & [\text{one}] \end{array}$$

All this talk about rules and rule applications makes us wonder if we cannot represent parsing as a deductive system (with inference rules and all). The first thing we need to decide is what will be the judgment used in this deductive system. What is it we are trying to show? Well, if a word is in a grammar's language. Or, in other words, if there is a derivation of w from S. But since S is transformed on the way, we need a more general statement: $\gamma \Rightarrow^* w$ denotes that the string w can be obtained from the string γ. The following rules define when this judgment holds:

$$\frac{\gamma_1 \Rightarrow^* w_1 \quad \gamma_2 \Rightarrow^* w_2}{\gamma_1\ \gamma_2 \Rightarrow^* w_1 w_2}\ conc \qquad \frac{}{\texttt{a} \Rightarrow^* \texttt{a}}\ id \qquad \frac{\gamma \Rightarrow^* w}{X \Rightarrow^* w}\ (r) X \Rightarrow \gamma$$

Rule $conc$ reduces the problem to simpler ones by splitting the string, rule id is an axiom (a terminal can be obtained from itself with 0 applications of production rules), and rule (r) is actually a set of rules, one for each production rule in the grammar. This last rule mimics the application of one production rule on the path to w. So if we want to know if w is a word in the language of the grammar, all we need to do is find a derivation of $S \Rightarrow *w$. In this system, the derivation of `-(x+1)` becomes:

$$\dfrac{\dfrac{}{\texttt{-} \Rightarrow^* \texttt{-}}\ id \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{\texttt{x} \Rightarrow^* \texttt{x}}\ id}{T \Rightarrow^* \texttt{x}}\ \text{var} \quad \dfrac{}{\texttt{+} \Rightarrow^* \texttt{+}}\ id \quad \dfrac{\dfrac{}{\texttt{1} \Rightarrow^* \texttt{1}}\ id}{T \Rightarrow^* \texttt{1}}\ \text{one}}{T\texttt{+}T \Rightarrow^* \texttt{x+1}}\ conc \times 2}{T \Rightarrow^* \texttt{x+1}}\ \text{plus} \qquad \dfrac{\dfrac{}{\texttt{(} \Rightarrow^* \texttt{(}}\ id \quad \dfrac{}{\texttt{)} \Rightarrow^* \texttt{)}}\ id}{\texttt{(}T\texttt{)} \Rightarrow^* \texttt{(x+1)}}\ conc \times 2}{T \Rightarrow^* \texttt{(x+1)}}\ \text{pars}}{\texttt{-}T \Rightarrow^* \texttt{-(x+1)}}\ conc}{T \Rightarrow^* \texttt{-(x+1)}}\ \text{neg}$$

It all seems right and well, except that if we follow the database saturation strategy we are lost: there are infinitely many words in this language... We need to be smarter about

this. If we have a string w_0 at hand and need to know if it is in the grammar, how can we use forward computation to decide this in a way that termination is guaranteed?

We need to first feed some facts to the program. In principle it contains assertions $\texttt{a} \Rightarrow^* \texttt{a}$ for every terminal in the language, but if we are only concerned with w_0 we can have such facts only for the terminals in w_0. Then we can restrict rule applications such that the conclusions, or new facts generated, $\gamma \Rightarrow^* w$ are such that (1) γ is a (sub)string in the grammar and (2) w is a (sub)string of w_0. The first condition avoids generating facts to which no rule can be applied. The second condition avoids generating facts that do not contribute to the derivation of w_0. Since there are only finitely many sub-strings in w_0, only a finite number of facts can be generated and we can go ahead and run the forward computation until database saturation. If the fact $S \Rightarrow^* w_0$ is in the final state, we know that w_0 belongs to the language, otherwise it does not.

Using this algorithm to parse `-(x+1)` we get a database with the following facts (among other redundant ones):

1	`-`	$\Rightarrow^*$	`-`	
2	`(`	$\Rightarrow^*$	`(`	
3	`x`	$\Rightarrow^*$	`x`	
4	`+`	$\Rightarrow^*$	`+`	
5	`1`	$\Rightarrow^*$	`1`	
6	`)`	$\Rightarrow^*$	`)`	
7	T	$\Rightarrow^*$	`1`	$\mathsf{one}(5)$
8	T	$\Rightarrow^*$	`x`	$\mathsf{var}(3)$
9	T`+`T	$\Rightarrow^*$	`x+1`	$conc \times 2(8, 4, 7)$
10	T	$\Rightarrow^*$	`x+1`	$\mathsf{plus}(9)$
11	`(`T`)`	$\Rightarrow^*$	`(x+1)`	$conc \times 2(2, 9, 6)$
12	T	$\Rightarrow^*$	`(x+1)`	$\mathsf{pars}(11)$
13	`-`T	$\Rightarrow^*$	`-(x+1)`	$conc(1, 12)$
14	T	$\Rightarrow^*$	`-(x+1)`	$\mathsf{neg}(13)$

This looks like a lot of work and a lot of check... Why would we want to do that when we have a perfectly fine system to find a derivation of w using backward chaining? Complexity, of course!

2.1 Complexity

What is the complexity of finding a derivation for $S \Rightarrow^* w$, such that $|w| = n$, using backward chaining? The most problematic rule during proof search is $conc$, given that we can split this string in many different ways. Worst case the word is not in the language and we have to exhaust all possible splits of this string. What we are constructing is a full binary tree with n leaves, so the question becomes: how many full binary trees with

n leaves are there? The answer is the n^{th} Catalan number[3]:

$$C_n = \frac{(2n)!}{(n+1)!n!}$$

Complexity-wise that does not look so good...

What is the complexity of saturating a database using the forward chaining method described previously for a string w such that $|w| = n$? This complexity is the sum of the size of the saturated database plus the number of ways we can *try* to apply a rule [1]: the algorithm searches through the know facts to see if they match the premises of a rule, before checking if the conclusion can or cannot be added to the database.

The saturated database can only contain facts $\gamma \Rightarrow^* w'$ such that w' is a substring of w, and γ is a substring of the right-side of grammar rules. Let k be the length of the longest right-side of all grammar rules (in our case, $k = 3$). A string of size n has $\binom{n+1}{2}$ non-empty substrings, therefore the saturated database contains at most $\binom{k+1}{2}\binom{n+1}{2} \in O(k^2n^2)$ facts.

For each of those facts, we can apply one of $conc$ or (r) rules to generate new facts (id does not have premises to match existing facts). The number of ways we can apply a rule depends on the rule. The rules (r), having one premise, will have at most $O(k^2n^2)$ matches. For $conc$, we have $O(k^2n^2)$ options for the first premise. Once the prefix of the substrings are decided, we have only a linear number of suffixes, so $O(k^3n^3)$. Thus the final complexity is $O(k^3n^3) = O(n^3)$, since k is a constant.

That is much better than what we had before!

The forward chaining method described is a well-know parsing algorithm for context-free grammars called **CYK algorithm** (Cocke-Younger-Kasami algorithm). It is one of the most efficient parsing algorithms in terms of worst-case asymptotic complexity. Interestingly, it uses dynamic programming to obtain the efficiency, which was also the case for the linear version of Fibonacci.

2.2 CYK parsing in Chomsky normal form

Although we are happy with the CYK parsing algorithm, it still technically depends on the length of the longest grammar rule. If this is too long, it may complicate the search quite a bit. Instead, if our grammar is in Chomsky[4] normal form, we restrict to $k = 2$, which behaves much better, without loosing any expressive power. A grammar is said to be in Chomsky normal form if all production rules are of the shape:

$$X \Rightarrow YZ \qquad \text{or} \qquad X \Rightarrow a$$

Where X, Y, Z are variables and a is a terminal symbol.

[3] Credits to Professor Christos Kapoutsis for figuring this out! :)

[4] If you do not know Noam Chomsky, go google him. He gives great interviews about politics nowadays.

These rules are represented as the facts $\mathsf{rule}(x, \mathsf{cat}(y, z))$ and $\mathsf{rule}(x, \mathsf{char}(a))$ respectively (where variables are now lowercase because we do not want them to be actual variables in the logic program).

The input string (which we want to parse) $w = a_1...a_n$ is represented as facts $\mathsf{string}(i, a_i)$, where i is a natural number using the successor representation.

We will define rules that compute $\mathsf{parse}(x, i, j)$, meaning that the substring $a_i...a_j$ can be generated (in one or more steps) from the variable x. These rules are (where uppercase letters *do* mean variables in the logic program):

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

Notice how they naturally incorporate the restrictions of the generated facts, so no ad-hoc verification needs to be performed. After saturating the database, the word will be successfully parsed if the fact $\mathsf{parse}(ss, s(0), n)$, where ss is the start symbol, is derived.

Assuming that the grammar has g production rules, let's see what is the complexity of this algorithm. The saturated database has g facts rule, n facts string and at most $O(g * n^2)$ facts parse (at most g different variables for n values of i and n values of j).

Now we count how many ways each rule can be fired. The first premise of the first rule can be fired g times, which fixes A and leaves n different ways to fire string. That is $O(g * n)$. For the second rule we can again match the first premise with at most g different facts. This fixes Y, leaving n^2 different possibilities for the second parse premise. For the third premise, only K is still free, and it can have n different values. Combining all of them, we have $O(g * n^3)$ different firings for the second rule.

Therefore, the worst-case complexity of the whole algorithm is $O(g * n^3)$, which is precisely the complexity of CYK parsing.

References

[1] David McAllester. On the Complexity Analysis of Static Analyses. *J. ACM*, 49(4):512–537, 2002.

[2] Raghu Ramakrishnan, Divesh Srivastava, and S. Sudarshan. *Efficient Bottom-Up Evaluation of Logic Programs*, pages 287–324. Springer US, 1992.