CS5126: Logic Programming and Constraints

Joxan Jaffar

March 3 - April 7, 2008
Searching a Derivation Tree

**Rule Order**

Does not affect the answers, only in the sequence they are discovered. However, it can

- affect how *quickly* an answer is found,
- determine if an answer is *ever* found

**Literal Order**

A *selection derivation step* \((G_1 \mid C_1) \rightarrow (G_2 \mid C_2)\), is defined as follows. Suppose \(G_1\) is of the form \(L_1, \cdots, L_i, \cdots, L_n\) where \(L_i\) is *selected*.

- \(L_i\) is a constraint:
  \(G_2\) is \(L_2, \cdots, L_n\) and \(C_2\) is \(C_1 \land L_i\).
  If \(solve(C_2) \equiv false\), then \((G_2 \mid C_2)\) is a *false* state.

- \(L_i\) is an atom:
  \(C_2\) is \(C_1\), and \(G_2\) is a rewriting of \(G_1\) at \(L_i\) using some rule \(R\).
  The variables in \(G_2\) are renamed away from \((G_1 \mid C_1)\). If there is no such \(R\), then then \((G_2 \mid C_2)\) is a *false* state.

If the solver were *complete*, then computing answers is is *independent* of literal order. Otherwise, we can get infinite derivations when in fact the constraints are unsatisfiable.

Any answer constraint is *always correct* (it never describes an error state) regardless of the solver.
Modes of Usage

A *mode of usage* for a predicate $p$ is a description of the arguments of $p$ encountered at runtime.

A goal $G$ **satisfies** a mode of usage if for every state in the derivation tree for $G$ of the form:

$$p(s_1, \ldots, s_n), L_1, \ldots, L_m \mid C$$

the effect of the constraint store $C$ on the arguments $s_1, \ldots, s_n$ of $p$ is correctly described by the mode of usage.

**Examples of Descriptions**

- **boundedness**
  “the second argument is bound”
  - eg. bound to anything: $p(X, Y) \mid Y = [\text{Head}|\text{Tail}]$
  - eg. bound to a fixed length list: $p(X, Y) \mid Y = [Z_1, Z_2, Z_3]$

- **groundness**
  “the second argument is ground”
  - eg. equal to anything: $p(X, Y) \mid Y = 3$
  - eg. equal to some specific value: $p(X, Y) \mid Y = 3$

- **constrained**
  “the second argument satisfies a certain constraint: $p(X, Y) \mid 1 \leq Y \leq 9$. “
Example

sumlist([], 0).
sumlist([N | L], N + S) :- sumlist(L, S).

Mode of Usage: first argument is grounded to a list of numbers

- Goals satisfying the MoU:
  - ?- sumlist([1], S).
  - L = [1, 2], S > Z, sumlist(L, S)

- Goals not satisfying the MoU:
  - ?- sumlist(L, 2).
  - S > 3, sumlist(L, S), L = [1, 2].

Check:

\[
\begin{align*}
\text{sumlist}([1], S) & \mid \text{true} \\
\Downarrow & \\
\text{sumlist}(L_1, S_1) & \mid [1] = [N_1|L_1], S = N_1 + S_1 \\
\Downarrow & \\
\Box & \mid [1] = [N_1|L_1], S = N_1 + S_1, L_1 = [], S_1 = 0
\end{align*}
\]

In this mode of usage, the derivation tree is **linear** in the size of the input list.

**Note:** When considering tree, an important factor is whether one (or a few) solutions are sought, or if all solutions are sought.
Example

(1) \text{sum}(N, S + N) :- \text{sum}(N - 1, S).
(2) \text{sum}(0, 0).

A classic example of wrong rule order:

\text{sum}(1, S) \mid true

\downarrow

\text{sum}(0, S_1) \mid S = 1 + S_1 \quad \square \mid 1 = 0 (false)

\downarrow

\text{sum}(-1, S_2) \mid S = 1 + S_2 \quad \square \mid S = 1

\downarrow

\text{sum}(-2, S_3) \mid S = 1 + S_3 \quad \square \mid -1 = 1 (false)

\ldots
Example - attempt 2

(3) \( \text{sum}(0, 0) \).
(4) \( \text{sum}(N, S + N) :- \text{sum}(N - 1, S) \).

We have reversed the rule order, but still:

\[
\begin{align*}
\text{sum}(1, 0) & \mid \text{true} \\
\Downarrow & \\
\text{sum}(0, -1) & \mid \text{true} \\
\Downarrow & \\
\text{sum}(-1, -1) & \mid \text{true} \\
\Downarrow & \\
\text{sum}(-2, 0) & \mid \text{true} \\
\Downarrow & \\
\text{sum}(-3, 2) & \mid \text{true} \\
\Downarrow & \\
\ldots
\end{align*}
\]

Clearly the intended mode of usage is that the first argument is non-negative.
Example - attempt 3

(5) \text{sum}(0, 0).
(6) \text{sum}(N, S + N) :- \text{sum}(N - 1, S), N \geq 1.

Note that the (new) constraint $N \geq 1$ is \textit{redundant}.

\[
\begin{align*}
\text{sum}(1,0) & | \text{true} \\
\downarrow & \\
\text{sum}(0,-1), 0 \geq 1 & | \text{true} \\
\downarrow & \\
\text{sum}(-1,-1) - 1 \geq 1, 0 \geq 1 & | \text{true} \\
\downarrow & \\
\text{sum}(-2,0) - 2 \geq 1, -1 \geq 1, 0 \geq 1 & | \text{true} \\
\downarrow & \\
\text{sum}(-3,2) - 3 \geq 1, -2 \geq 1, -1 \geq 1, 0 \geq 1 & | \text{true} \\
\downarrow & \\
\vdots & \\
\end{align*}
\]

The problem is that the new constraint is reachable only after the recursive call because of \text{left-to-right selection}. 
Example - final attempt 4

(7)   sum(0, 0).
(8)   sum(N, S + N) :- N >= 1, sum(N - 1, S).

▶ ?- sum(0, 1) is finitely failed
▶ ?- sum(1, S) returns S = 1
Literal Ordering

A general guideline:
ensure failure occurs as soon as possible,
and delay choices to as late as possible.

We have seen examples of early failure.

**Example of Late Choice:** run goals with ONE answer first.

A tree is **deterministic** if it is finite and each node has at most one descendant which is not failed. A predicate is deterministic (for a mode of usage) if for any goal \(p(\cdots)\) (satisfying the mode), the tree is deterministic.

For the mode \(\text{sum}(\cdots)\) where the first argument is ground, the predicate \(\text{sum}\) is not deterministic in:

\begin{align*}
(5) & \quad \text{sum}(0, 0).
(6) & \quad \text{sum}(N, S + N) : - \text{sum}(N - 1, S), N \geq 1.
\end{align*}

but is deterministic in:

\begin{align*}
(7) & \quad \text{sum}(0, 0).
(8) & \quad \text{sum}(N, S + N) : - N \geq 1, \text{sum}(N - 1, S).
\end{align*}
father(a, b).
...
mother(b, c).
...
grandfather(Z, X) :- father(Z, Y), father(Y, X).
grandfather(Z, X) :- father(Z, Y), mother(Y, X).

Consider the mode of \texttt{grandfather(Z, X)} where \texttt{X} is ground (who is the grandfather of \texttt{X}?).

Note that the \textit{first literal} in both rules are NOT deterministic.

Now swap literals so that deterministic ones come first:

\begin{verbatim}
gradfather(Z, X) :- father(Y, X), father(Z, Y).
gradfather(Z, X) :- mother(Y, X), father(Z, Y).
\end{verbatim}

This is no more efficient. (Why?)
Deterministic Predicates

As a natural extension to determinism is the guideline: run predicates with fewer answers first.

parent(Y, X) :- father(Y, X).
parent(Y, X) :- mother(Y, X).
grandfather(Z, X) :- father(Z, Y), parent(Y, X).

Consider the mode of grandfather(Z, X) where X is ground (who is the grandfather of X?).

The above is not efficient. Much better is:

grandfather(Z, X) :- parent(Y, X), father(Z, Y).

(Why?)
If-Then-Else and Once

If-Then-Else

\[(G \rightarrow G_t; G_e) \mid C\]

- succeeds with answer \(C_1\), then we derive \((G_t \mid C_1)\)
- finitely fails, then we derive \((G_e \mid C)\)

Example:

\[
\text{abs}(X, Y) :- (X \geq 0 \rightarrow Y = X ; Y = -X).
\]

Once

\[(\text{once}(G), \overset{\sim}{L}) \mid C\]

- succeeds with answer \(C_1\), then we derive \((\overset{\sim}{L} \mid C_1)\)
- finitely fails, then we obtain finite failure.
Adding Redundant Constraints

Two kinds of redundancy in adding a constraint to a rule/goal:

- **Answer redundancy**
  This is when we add a constraint that is redundant because it does not change the *answers* of the program

- **Solver redundancy**
  This is when we add a constraint that is redundant because it does not change the *answers* of the constraint solver
Answer Redundancy

(1) \( \text{sum}(0, 0) \).

(2) \( \text{sum}(N, S + N) :- N \geq 1, \text{sum}(N - 1, S) \).

\[
\text{sum}(N, 7) \mid \text{true}
\]

\[
\downarrow
\]

\( \text{sum}(N_1, S_1) \mid N = N_1 + 1, S_1 = 6 - N_1, N_1 \geq 0 \)

\[
\downarrow
\]

\( \text{sum}(N_2, S_2) \mid N = N_2 + 2, S_2 = 4 - 2 \times N_2, N_2 \geq 0 \)

\[
\downarrow
\]

\( \text{sum}(N_3, S_3) \mid N = N_3 + 3, S_3 = 1 - 3 \times N_3, N_3 \geq 0 \)

\[
\downarrow
\]

\( \text{sum}(N_4, S_4) \mid N = N_4 + 4, S_4 = -3 - 4 \times N_4, N_4 \geq 0 \)

\[
\downarrow
\]

\[\cdots\]

**Problem:** none of the constraints above are unsatisfiable.

**Solution:**

(3) \( \text{sum}(0, 0) \).

(4) \( \text{sum}(N, S + N) :- N \geq 1, S \geq 0, \text{sum}(N - 1, S) \).

Note that this change does not change the *answers*. 
Solver Redundancy

A constraint is **solver redundant** if it is entailed by the constraint store.

Adding (solver) redundant constraints can be useful when it makes explicit information which an **incomplete solver** is incapable of determining.

1. \( \text{fact}(0, 1) \).
2. \( \text{fact}(N, N\times F) :- N \geq 1, F \geq 1, \text{fact}(N - 1, F) \).
(Note: \( F \geq 1 \) is answer-redundant)

The goal \( \text{fact}(N, 7) \) runs forever.

\[
\text{fact}(N, 7) \mid \text{true}
\]

\[
\downarrow
\]

\[
\text{fact}(N-1, F_1) \mid F_1 \geq 1, N \geq 1, 7 = N \times F_1
\]

\[
\downarrow
\]

\[
\text{fact}(N-2, F_2) \mid F_2 \geq 1, N \geq 2, 7 = N \times (N-1) \times F_2
\]

\[
\downarrow
\]

\[
\text{fact}(N-3, F_3) \mid F_3 \geq 1, N \geq 3, 7 = N \times (N-1) \times (N-2) \times F_3
\]

\[
\downarrow
\]

\[
\text{fact}(N-4, F_4) \mid F_4 \geq 1, N \geq 4, 7 = N \times (N-1) \times (N-2) \times (N-3) \times F_4
\]

\[
\downarrow
\]

\[
\ldots
\]
Solver Redundancy

In the previous state:

\[ \text{fact}(N - 4, F_4) \mid F_4 \geq 1, N \geq 4, 7 = N \times (N - 1) \times (N - 2) \times (N - 3) \times F_4 \]

in fact, the expression \( N \times (N - 1) \times (N - 2) \times (N - 3) \times F_4 \) must be greater than 24. However, many constraint solvers may not be be able to determine this.

Now add the fact that the factorial of \( N \) is always larger than \( N \):

(3) \quad \text{fact}(0, 1).

(4) \quad \text{fact}(N, FN) :-
FN = F \times N, N \geq 1, F \geq 1, N \leq FN,
\text{fact}(N - 1, F).

The goal \text{fact}(N, 7) now will in fact terminate (finitely fail).
Optimization

Running a goal derives one more answers. Optimization involves deriving the best answer.

Recall the “Options Trading” Example and the butterfly combination bets that a stock price remains in a certain range and bounds the loss.

call_option(B, S, C, E, P) :- 0 ≤ S, S ≤ E/100, P = -C*B.
call_option(B, S, C, E, P) :- S ≥ E/100, P = (100*S - E - C) * B.

put_option(B, S, C, E, P) :- 0 ≤ S, S ≤ E/100, P = (E-100*S-C) * B.
put_option(B, S, C, E, P) :- S ≥ E/100, P = -C * B.

butterfly(S, P1 + 2*P2 + P3) :-
    Buy = 1, Sell = -1,
    call_option(Buy, S, 100, 500, P1),
    call_option(Sell, S, 200, 300, P2),
    call_option(Buy, S, 400, 100, P3).

Optimization would be to discover the maximum P for ?- butterfly(S, P).
(S = 3, P = 100).
Simple Optimization

solve(X, C): find one solution X with cost C
try(soln1, soln2): given soln1, find a better soln2.

try(soln(X0, C0), soln(X, C)) :-
    C1 < C0,
    solve(X1, C1),
    try(soln(X1, C1), soln(X, C)).
try(soln(X, C), soln(X, C)).

- Needs an initial call to solve to obtain a first value of C
- The search process implements a basic branch-and-bound strategy

In what follows, we study more advanced techniques of search, for both feasible solutions as well as optimal solutions.