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*** $(\mathscr{G}_1 \mid \mathscr{C}_1) \Longrightarrow (\mathscr{G}_2 \mid \mathscr{C}_2)$, is defined as follows. Suppose $\mathscr{G}_1$ is of the form $L_1, \cdots, L_i, \cdots, L_n$ where L_i is ***selected***.

- L_i is a constraint:
  $\mathscr{G}_2$ is $L_2, \cdots, L_n$ and $\mathscr{C}_2$ is $\mathscr{C}_1 \wedge L_i$.
  If $solve(\mathscr{C}_2) \equiv false$, then $(\mathscr{G}_2 \mid \mathscr{C}_2)$ is a *false* state.
- L_i is an atom:
  $\mathscr{C}_2$ is $\mathscr{C}_1$, and $\mathscr{G}_2$ is a rewriting of $\mathscr{G}_1$ at L_i using some rule R.
  The variables in $\mathscr{G}_2$ are renamed away from $(\mathscr{G}_1 \mid \mathscr{C}_1)$. If there is no such R, then then $(\mathscr{G}_2 \mid \mathscr{C}_2)$ 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:

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

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

Examples of Descriptions

- ***boundedness***
  "the second argument is bound"
  - eg. bound to anything: $p(X, Y) \mid Y = [Head|Tail]$
  - 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 satsifies 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 satisfing the MoU:
  - `?- sumlist([1], S).`
  - `L = [1, 2], S > Z, sumlist(L, S)`
- Goals not satisfing the MoU:
  - `?- sumlist(L, 2).`
  - `S > 3, sumlist(L, S), L = [1, 2].`

Check:
$$\begin{cases} sumlist([1], S) \mid true \\ \Downarrow \\ sumlist(L_1, S_1) \mid [1] = [N_1|L_1], S = N_1 + S_1 \\ \Downarrow \\ \square \mid [1] = [N_1|L_1], S = N_1 + S_1, L_1 = [], S_1 = 0 \end{cases}$$

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.

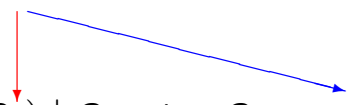
Example

```
(1)   sum(N, S + N) :- sum(N - 1, S).
(2)   sum(0, 0).
```

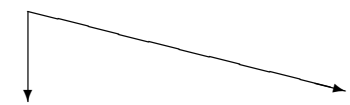
A classic example of wrong rule order:

$sum(1, S) \mid true$

$sum(0, S_1) \mid S = 1 + S_1 \qquad \square \mid 1 = 0 (false)$



$sum(-1, S_2) \mid S = 1 + S_2 \qquad \square \mid S = 1$



$sum(-2, S_3) \mid S = 1 + S_3 \qquad \square \mid -1 = 1 (false)$

...

Example - attempt 2

```
(3)   sum(0, 0).
(4)   sum(N, S + N) :- sum(N - 1, S).
```

We have reversed the rule order, but still:

$$sum(1,0) \mid true$$

$$\Downarrow$$

$$sum(0,-1) \mid true$$

$$\Downarrow$$

$$sum(-1,-1) \mid true$$

$$\Downarrow$$

$$sum(-2,0) \mid true$$

$$\Downarrow$$

$$sum(-3,2) \mid true$$

$$\Downarrow$$

$$\ldots$$

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

Example - attempt 3

```
(5)   sum(0, 0).
(6)   sum(N, S + N) :- sum(N - 1, S), N >= 1.
```

Note that the (new) constraint $N \geq 1$ is ***redundant***.

$$sum(1,0) \mid true$$

$$\Downarrow$$

$$sum(0,-1), 0 \geq 1 \mid true$$

$$\Downarrow$$

$$sum(-1,-1) - 1 \geq 1, 0 \geq 1 \mid true$$

$$\Downarrow$$

$$sum(-2,0) - 2 \geq 1, -1 \geq 1, 0 \geq 1 \mid true$$

$$\Downarrow$$

$$sum(-3,2) - 3 \geq 1, -2 \geq 1, -1 \geq 1, 0 \geq 1 \mid true$$

$$\Downarrow$$

$$\ldots$$

The problem is that the new constraint is reachable only after the recursive call because of 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 guidline:
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 $sum(\cdots)$ where the first argument is ground, the predicate `sum` is not deterministic in:

```
(5)   sum(0, 0).
(6)   sum(N, S + N) :- sum(N - 1, S), N >= 1.
```

but is deterministic in:

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

Deterministic Predicates

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

Note that the first literal in both rules are NOT deterministic.

Now swap literals so that deterministic ones come first:

```
gradfather(Z, X) :- father(Y, X), father(Z, Y).
gradfather(Z, X) :- mother(Y, X), father(Z, Y).
```

This is nore 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

$(G \longrightarrow G_t; G_e) \mid C$ executes as follows: if $(G \mid C)$

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

Example:

```
abs(X, Y) :- (X >= 0 -> Y = X ; Y = -X).
```

Once

$(once(G), \tilde{L}) \mid C$ executes as follows: if $(G \mid C)$

- succeeds with answer C_1, then we derive $(\tilde{L} \mid C_1)$
- 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)   sum(0, 0).
(2)   sum(N, S + N) :- N >= 1, sum(N - 1, S).
```

$$sum(N, 7) \mid true$$

$$\Downarrow$$

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

$$\Downarrow$$

$$sum(N_2, S_2) \mid N = N_2 + 2, S_2 = 4 - 2 * N_2, N_2 \geq 0$$

$$\Downarrow$$

$$sum(N_3, S_3) \mid N = N_3 + 3, S_3 = 1 - 3 * N_3, N_3 \geq 0$$

$$\Downarrow$$

$$sum(N_4, S_4) \mid N = N_4 + 4, S_4 = -3 - 4 * N_4, N_4 \geq 0$$

$$\Downarrow$$

$$\ldots$$

Problem: none of the constraints above are unsatisfiable.

Solution:

```
(3)   sum(0, 0).
(4)   sum(N, S + N) :- N >= 1, S >= 0, 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)   fact(0, 1).
(2)   fact(N, N*F) :- N >= 1, F >= 1, fact(N - 1, F).
(Note:  F >= 1 is answer-redundant)
```

The goal `fact(N, 7)` runs forever.

$$fact(N,7) \mid true$$

$$\Downarrow$$

$$fact(N-1,F_1) \mid F_1 \geq 1, N \geq 1, 7 = N * F_1$$

$$\Downarrow$$

$$fact(N-2,F_2) \mid F_2 \geq 1, N \geq 2, 7 = N * (N-1) * F_2$$

$$\Downarrow$$

$$fact(N-3,F_3) \mid F_3 \geq 1, N \geq 3, 7 = N * (N-1) * (N-2) * F_3$$

$$\Downarrow$$

$$fact(N-4,F_4) \mid F_4 \geq 1, N \geq 4, 7 = N * (N-1) * (N-2) * (N-3) * F_4$$

$$\Downarrow$$

$$\ldots$$

Solver Redundancy

In the previous state:

$$fact(N-4, F_4) \mid F_4 \geq 1, N \geq 4, 7 = N*(N-1)*(N-2)*(N-3)*F_4$$

in fact, the expression $N*(N-1)*(N-2)*(N-3)*F_4$ 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)   fact(0, 1).
(4)   fact(N, FN) :-
            FN = F * N, N >= 1, F >= 1, N <= FN,
            fact(N - 1, F).
```

The goal `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.