Deductive Databases and Logic Programming

(Winter 2007/2008)

Chapter 9: Constraint Logic Programming

- Introduction, Examples
- Basic Query Evaluation
- Finite Domain Constraint Solver

Introduction (1)

- Constraint logic programming (CLP) extends standard logic programming by constraints, which can in principle be any kind of logical formulae.
- A concrete system permits only a subset of formulae, because it needs a constraint solver that can
  - ◇ check a set of constraints for consistency,

    Sometimes incomplete constraint solvers are used, which are not able to detect all inconsistencies.
  - ◇ simplify a set of constraints, so that it can be presented to the user as an answer.

    Optimal would be `Variable = Value`, but this is not always possible.

Introduction (2)

- Normally, the constraints are simply literals with special predicates.
- The semantics of these predicates is not defined by rules, but by a logical "constraint theory".
- The semantics is implemented or approximated by the constraint solver in the system.

  "Constraint Handling Rules" (CHR) is an approach to define constraint solvers by special rules. However, these look quite different from the usual logic programming rules.

Introduction (3)

- The function symbols that can be used for term construction in constraint literals is limited: It must be evaluable functions which the constraint solver knows (e.g., `+`, `-`, `*`, `/`).
- Variables in constraints have a specific domain (e.g., integers). They cannot be bound to arbitrary terms.
- Note the difference between
  - ◇ The value of variable `X` is the term `Y+3`.
  - ◇ The value of variable `X` is an integer, at the moment it is known that `X = Y+3`.

Introduction (4)

- For instance, the following logic program would work in the Ciao System with the package CLP(R) for constraint programming over reals:

```
:- use_package(clpr).
p(X,Y) :- X .=. Y * 3, q(X,Y).
q(X,Y) :- X - 2 .=. Y.
```

  Ciao can be downloaded from [http://www.ciaohome.org/].

- The operator `.=.` means simply $=$, but that symbol is used for unification already.

- For the query `p(X,Y)`, one gets the answer `Y.=.1.0`, `X.=.3.0` (system of linear equations was solved).

Introduction (5)

- In SWI-Prolog, the same example looks as follows:

```
:- use_module(library(clpr)).
p(X,Y) :- {X = Y * 3}, q(X,Y).
q(X,Y) :- {X - 2 = Y}.
```

In the Debian/Ubuntu Linux distribution, this constraint solver is missing (probably a bug). The Windows distribution contains it. Under Linux, one can put the files to `/usr/lib/swi-prolog/library/clp`.

- Here, constraints are marked with `{...}`.

This shows that working with constraints is not very portable. In ECLiPSe, e.g. the operator $= is used and type declarations for the variables are important.

Introduction (6)

- For comparison, consider the logic program

```
p(X,Y) :- X = Y * 3, q(X,Y).
q(X,Y) :- X - 2 = Y.
```

- Here, the answer is "no": The call to `q` is `q(Y*3,Y)`. Then it is required to unify `(Y*3)-2` and `Y`.

  This only fails if the system uses an occur check. If the second equation were written `X = Y+2`, it would fail in any Prolog system.

- If one uses "`is`" instead of "`=`", one would get an instantiation fault.

  "`is`" requires that all variables occurring on the right hand side are bound to an arithmetic expression.

Basic Query Evaluation (1)

- Query evaluation works as SLD-resolution, however besides the current goal, there is also a current set of constraints.
- If the first/selected literal in the current goal is a constraint,
  - ◇ it is moved to the current set of constraints.
  - ◇ The current set of constaints is then checked for consistency (with the constraint solver). If it is inconsistent, the current derivation fails.

    Since one typically searches the derivation tree depth first, backtracking to the last untried choice point will then occur.

Basic Query Evaluation (2)

- If the first/selected literal is a normal literal (not a constraint), a usual SLD-step is done, however
  - Instead of unifying the head $p(t_1, \ldots, t_n)$ of a program clause (with fresh variables) with the selected literal $p(u_1, \ldots, u_n)$, one adds the constraints $u_1 = t_1, \ldots, u_n = t_n$ to the constraint store.

    Furthermore, as usual, the body of the clause is inserted into the goal (replacing the selected/calling literal).
  - In this way, unification can be seen as a very specific constraint solver.

Basic Query Evaluation (3)

- The constraint solver may also transform the current set of constraints into an equivalent one (simplify them, e.g. solve equations).

  This will typically occur when a constraint is added.

- A derivation is successful when it ends in an empty goal and a satisfiable constraint store.
- Instead of an answer substitution, the contents of the constraint store will then be printed.

  Possibly restricted to the variables that occurred in the query.

Delaying Conditions (1)

- One specific feature of CLP is that constraints are usually not immediately evaluated, but delayed until more information becomes available about the variables occurring in them.
- For instance, consider the program:

```
p(X) :- X .<. 5, q(X).
q(X) :- r(X,Y).
r(7,Y) :- very_complex_calculation(Y).
```

- The condition `X .<. 5` cannot be evaluated at the point where it is called.

Delaying Conditions (2)

- The constraint `X .<. 5` waits in the background and as soon as `X` is bound to `7`, it causes the failure.
- In Prolog (with `<` instead of `.<.`) one would get an instantiation fault.
- One would have to move `X < 5` after the call `q(X)`, but then the `very_complex_calculation` is done, and the failure is detected only afterwards.
- This gives logic programming a kind of coroutining.

Unification vs. Equations (1)

- The above example also explains why unification is conceptually replaced by solving special equations:
  - ◇ The variable `X` gets the value `7` by a unification step, and the constraint solver must react.
- This does not necessarily mean that when a CLP system executes a standard logic program, the implementation is different.

  Actually, most modern Prolog systems have CLP libraries.

Unification vs. Equations (2)

- However, the following example does not work:

```
p(X,Y) :- X .=. Y * 3, q(X,Y).
q(Z+2,Z).
```

- Formally, one gets the system of equations

$$X = Y * 3,\ X = Z + 2,\ Y = Z.$$

  This could be solved with solution X=3, Y=1, Z=1.

- But in SWI-Prolog and Ciao, one gets a kind of instantiation fault.

  Unification introduces equations with "=", while the constraint solver mainly works with ".=.". A problem might also be to determine whether Z is a standard Prolog variable or a numerical one.

Finite Domain Constraints (1)

- A finite domain solver keeps for each variable a finite domain of values, usually an interval of integers (or union of such intervals).
- For the CLP-FD Solver in SWI-Prolog, one declares a domain for a variable e.g. with

```
X in 0..9
```

- It is also possible to do this for a list of variables, e.g.

```
[X, Y, Z] ins 0..100
```

Finite Domain Constraints (2)

- Equality for the CLP-FD solver is written `#=`.

  Note that the constraints are not written in `{..}`.

  Besides `#=`, the solver knows e.g. about `#<`, `#<=`, `#>=`, `#>`, `#\=`. Boolean values are treated as 0 and 1, and then boolean connections `#\` (not), `#\/` (or), `#/\` (and), `#<==>` (iff), `#==>` (then), `#<==` (if) can be used. The expressions can contain `+`, `-`, `*`, `/`, `mod`, `rem`, `abs`, `min`, `max`, `^`.

- One can ask the solver to try all possible values for a variable (or list of variables) one by one with

  `label(X)`

  When this variable has a concrete value, often the constraints entail values for other variables.

Finite Domain Constraints (3)

- A well-known puzzle is

$$
\begin{array}{ccccc}
 & S & E & N & D \\
+ & M & O & R & E \\
\hline
M & O & N & E & Y
\end{array}
$$

- The task is to find digits for the letters (all different, S and M not zero) such that this addition gives the correct result.
- A solution can easily be found with a finite domain solver. One only needs a formal specification of the problem as shown on the next slide.

Finite Domain Constraints (4)

- Puzzle in SWI-Prolog:

```
:- use_module(library(clpfd)).

puzzle([S,E,N,D],[M,O,R,E],[M,O,N,E,Y]) :-
    Vars = [S,E,N,D,M,O,R,Y],
    Vars ins 0..9,
    all_different(Vars),
                S * 1000 + E * 100 + N * 10 + D +
                M * 1000 + O * 100 + R * 10 + E #=
    M * 10000 + O * 1000 + N * 100 + E * 10 + Y,
    M #\= 0, S #\= 0.
```

Example from [http://www.swi-prolog.org/man/clpfd.html] (with one small modification in the head).

Finite Domain Constraints (5)

- To solve the puzzle, call

```
puzzle(A, B, C), label(A).
```

Of course, one could call `label` also for the other variable lists, but that is not necessary. The constraint solver can uniquely determine their values once it has assigned concrete values to the variables in `A`.

- CLP can also be used to make functions with arithmetic invertible:

```
:- use_module(library(clpfd)).

fac(0, 1).
fac(N, F)  :-  N #> 0, N1 #= N-1, F #= N*F1,
               fac(N1, F1).
```