Constraints and Logic Programming

- All problems involving finite domains (including booleans, and sets) may be solved, in principle, with logic programming (LP) alone.

  (after all, finite domains are a subset of the Herbrand Universe: constants).

- Why should one move towards Constraint Programming (CP), rather than staying within LP?

  - Greater Efficiency:
    - Most combinatorial problems are more easily solved by constraint propagation than simple generate and test implemented in LP.
  - Greater Expressiveness:
    - Specific constraints are harder to express in LP (e.g. conditional constraints, global constraints).

Constraints and Logic Programming

- If propagation of constraints is so important for solving constraint (satisfaction / optimisation) problems why not to abandon LP altogether and move towards Constraint Logic Programming (CLP).

  - Modelling - Declarativeness
    - LP features such as unification (allowing flexible input/output parameters) and backtracking provide a declarative style of programming where constraints can be very easily expressed.

  - Natural extension:
    - The semantics of LP already assume a special type of constraint processing – equality of Herbrand terms.
    - All that CLP requires is an extension of this constraint solving capabilities to other useful domains.

Operational Semantics of CLP

- The operational semantics of CLP (LP is a special case) can be described as a transition system on states.
- The state of a constraint solving system can be abstracted by a tuple

  <G, C, S >

  where
  - **G** is a set of constraints (goals) to solve
  - **C** is the set of active constraints
  - **S** is a set of passive constraints
- Sets **C**, **S** are considered the **Constraint Store**.
- A derivation is just a sequence of transitions.
- A state that cannot be rewritten is a final state.
- An executor of CLP (LP) aims at finding successful final states of derivations starting with a query.

Operational Semantics of CLP

- A derivation is **failed** if it is finite and its final state is fail.
- A derivation is **successful** if it is finite and processes all the constraints, i.e. it ends in a state with form

$$\langle \emptyset, C, S \rangle$$

- The state transitions are the following

  S(elect): Select a goal (constraint)

$$\frac{\textbf{g requires solving constraint c and other goals G'}}{\langle G \cup \{g\}, C, S \rangle \rightarrow_s \langle G \cup G', C, S \cup \{c\} \rangle}$$

- It is assumed that
  - A computation rule selects some goal g, among those still to be considered.
  - This goal requires some constraint c to be solved and/or further goals, G' , to be considered

Operational Semantics of CLP

- The other transition rules require the existence of two functions, **`infer(C,S)`** and **`consistent(C)`**, adequate to the domains that are considered.

- **I(nfer):** Process a constraint, inferring its consequences

$$\frac{\texttt{(C', S')} = \texttt{infer(C,S)}}{\texttt{< G, C, S >} \rightarrow_i \texttt{< G, C', S' >}}$$

- **C(heck):** Check the consistency of the Store

$$\frac{\texttt{consistent (C)}}{\texttt{< G, C, S >} \rightarrow_c \texttt{< G, C, S >}} \qquad \frac{\neg\,\texttt{consistent (C)}}{\texttt{< G, C, S >} \rightarrow_c \texttt{fail}}$$

Operational Semantics of CLP

In Logic Programming

- Handling a goal *g* is simply checking whether there is a clause $h \leftarrow B$ whose head *h* matches the goal. In this case, the equality constraint between Herbrand terms *g*=*h* is added to the passive store.

$$\frac{\mathtt{g\ is\ a\ goal} \ \wedge \ \exists\ \mathtt{h} \leftarrow \mathtt{B}}{\mathtt{<\ G} \cup \mathtt{\{g\},\ C,\ S\ >} \rightarrow_{\mathtt{s}} \mathtt{<\ G} \cup \mathtt{B,\ C,\ S} \cup \mathtt{\{h = g\}\ >}}$$

- Function infer(C,S) performs unification of the terms included in *g* and *h*, after applying all substitutions included in C

```
function infer(C, g=h)
     if unify(g↑C,h↑C, C′) then infer=(success, C′)
     else infer = (fail,_);
end function
```

- Checking consistency is simply checking whether the previous unification has returned success or failure (of course, in practice the two functions are merged).

Operational Semantics of CLP

- For example, assuming that
  - **`C = {X / f(Z), Y / g(Z,W)}`** `% a solved form`
  - **`G`** includes goal **`p(Z,W)`**
  - there is a clause **`p(a,b) :- B.`**

then the following transitions take place, when goal **`p(Z,W)`** is "called"

`< G , {X/f(Z), Y/g(Z,W)}, {} >`

$\rightarrow_s$ % goal selection

`< G\{g}∪B, {X/f(Z), Y/g(Z,W)}, {p(Z,W) = p(a,b)}} >`

$\rightarrow_{i,c}$ % unification

`< G\{g}∪B, {X/f(a), Y/g(a,b), Z/a, W/b}, {} >`

Operational Semantics of CLP

In Constraint Logic Programming

- Handling a goal *g* is as before. Handling a constraint in a certain domain, usually built-in, simply places the constraint in set S.

.

$$\frac{\texttt{g is a constraint}}{\texttt{< G} \cup \texttt{\{g\}, C, S >} \rightarrow_s \texttt{< G, C, S} \cup \texttt{\{g\} >}}$$

- Function infer(C,S) propagates the constraint to the active store, by methods that depend on the constraint system used. Tipically,
  - It attempts to reduce the values in the domains of the variables; or
  - obtain a new solved form (like in unification)

- Checking consistency also depends on the constraint system.
  - It checks whether a domain has become empty; or
  - It was not possible to obtain a solved form.

Operational Semantics of CLP

A positive example

- Assuming that variables A and B can take values in 1..3, then the constraint store may be represented as

```
C,S = {A in 1..3, B in 1..3}, { }
```

- Rule S: If the constraint selected by rule S is A > B, then the store becomes

```
C,S = {A in 1..3, B in 1..3}, { A > B}
```

- Rule I: The Infer rule propagates this constraint to the active store, which becomes

```
C,S = {A in 2..3, B in 1..2,  A > B}, {  }
```

- Rule C: The system does not find any inconsistency in the active store, so the system does not change.

Operational Semantics of CLP

A negative example

- Assuming that variables A and B can take, respectively values in the sets {1,3,5} and {2,4,6}. The constraint store may be represented as

```
C,S = { A in {1,3,5}, B in {2,4,6} }, { }
```

- Rule S: If the constraint selected by rule S is A = B, the store becomes

```
C,S = { A in {1,3,5}, B in {2,4,6} }, { A = B }
```

- Rule I: The rule propagates this constraint to the active store. Since no values are common to A and B, their domains become empty

```
C,S = {A in {}, B in {} }, {  }
```

- Rule C: This rule finds the empty domains and makes the transition to fail (signalling the system to backtrack in order to find alternative successful derivations).

(C)LP solutions

- In LP and CLP, solutions are obtained by inspecting the constraint store of the final state of a successful derivation.

- In systems that maintain a solved form (like LP, but also constraint systems CLP(B) for booleans, and CLP(Q), for rationals), solutions are obtained by projection of the store to the relevant variables.

  For example if the initial goal was p(X,Y) and the final store is

  `{X/f(Z,a), Y/g(a,b), W/c}`

  then the solution is the projection to variables X and Y

  `{X/f(Z,a), Y/g(a,b)}`

- In systems where a solved form is not maintained, solutions require some form of enumeration. For example, from

  `C = {X in 2..3, Y in 1..2,  X > Y},`

  the solutions **<X=2;Y=1>**, **<X=3;Y=1>** and **<X=3;Y=1>**, are obtained by enumeration.

Propagação e Pesquisa

- Consistência de:
  - Nó
  - Arco
  - Intervalo
  - Arco generalizada (restrições globais)
- Heurísticas:
  - Escolha de variável (e.g. first-fail)
  - Escolha de valor

CLP(FD) no SICStus

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

- Domínio (inteiros):
  - `domain(+Vars, +Min,+Max)`
  - `?Var in +Range`
- Enumeração: `indomain/1, labeling/2`
- Otimização: `minimize/maximize(Goal,?X)`

Restrições

- Aritméticas:
  #=, #\=, #>, #<, #>=, #=<
- Proposicionais:
  #\/, #/\, #=>, #<=>, e #\ (/1:not, /2: xor)
- Reificação:
  #<=>

Restrições Globais

- `sum(+Xs, +RelOp, ?Value)`
- `minimum(?Value,+Xs)` e `maximum(?Value,+Xs)`
- `global_cardinality(+Xs,+Vals)`
- `all_different/1,2` (acúcar sintático)
- `all_distinct/1,2`
- `element(?Index, +List, ?Value)`
- `assignment(+Xs, +Ys)`
- `circuit(?Succ)`
- `disjoint2(+Rectangles)`
- `cumulative/1,2` e `cumulatives/2,3`