CLP($\mathcal{R}$) – Constraint Logic Programming

Constraints

What is a constraint?

A relation that should be satisfied.

Examples:

- "Two columns of table must be equal width"
- "Window X must be above window Y"
- A resister in circuit simulation should obey Ohm's law.
- "$X + 4Y < 10$"

Advantages: *declarative*, high-level.

Disadvantages: easier to state than satisfy, debugging, program state interaction.

Constraints (cont)

(a more formal definition)

Form and meaning of constraint is specified by its *domain*:

- syntax.
- valid variable values.
- meanings of symbols in constraint.

Consider constraint $Y^2 < 1$
Meaning is different if for integers, reals, or complex numbers.

A *primitive constraint* from domain $\mathcal{D}$ is a relational symbol with arguments.
Each argument is a made from variables, constants, and functions from $\mathcal{D}$.

Example: For $\mathcal{R}$: `X + 5 < Y`

Domains & Constraints

For the domain of real numbers, $\mathcal{R}$:

variable values: Any real number.
constant values: Any real number.
relations: $<$, $>$, $\leq$, $\geq$, and $=$.
functions: $+$, $-$, $\times$, and $/$.

A *constraint* is a conjunction of primitive constraints:

$$c_1 \wedge c_2 \wedge \ldots \wedge c_n$$

where c_i are primitive.

Solver Properties

Solver input: a constraint.

Solver output: *true*, *false*, *unknown*.
(typically a *true* output is accompanied by proof)

A solver is *complete* if it never answers *unknown*.

A solver is *sound* if it never answers incorrectly. (we will require sound solvers)

We prefer complete solvers. Sometimes completeness is impractical or impossible.

Valuations and Satisfiability

A *valuation* is an assignment to variables.

A constraint is *satisfiable* if there exists a valuation for its variables that makes it true (otherwise *unsatisfiable*).

If constraint is satisfiable, sound complete solver should output *true* with the valuation as proof.

Exercise:

1. Is a solver that always outputs *unknown* sound? complete?
2. Is a solver that always outputs *true* sound? complete?

CLP($\mathcal{D}$)

For domain $\mathcal{D}$, CLP($\mathcal{D}$) is a language framework.

Example CLP languages:

- Prolog
- CLP(Σ^*) – domain of regular sets.
- CLP($\mathcal{R}$) – domain of real numbers (and tree types)

CLP($\mathcal{R}$):

- A superset of Prolog.
- Solves arbitrary sets of linear (in)equalities.
- Can solve other constraints if it can use one-step deductions.

CLP($\mathcal{R}$) Examples

Sample *goals*:

```
1 ?- X + 3*Y > 4, Y = 10.

Y = 10
-26 < X

*** Yes

2 ?- 2*X + 3*Y = 4, 6*X - Y = 10.

Y = 0.2
X = 1.7

*** Yes

3 ?- X*X = -1.

-1 = X * X

*** Maybe
```

CLP($\mathcal{R}$) Programs

CLP($\mathcal{R}$) programs are collections of *facts* and *rules*:

```
/* centigrade-fahrenheit relation */
cf(C,F) :- F = 1.8*C + 32.

/* max relation: max(Z,X,Y) -- "Z is max of X and Y" */
max(X,X,Y) :- X > Y.
max(Y,X,Y) :- X <= Y.

/* append relation: append(Zs,Xs,Ys) -- like Zs = Xs ++ Ys */
append(Ys,[],Ys).
append([X|Zs],[X|Xs],Ys) :- append(Zs,Xs,Ys).
```


cf(C,F)

CLP($\mathcal{R}$):

```
cf(C,F) :- F-32.0 = 1.8*C.
```

The same program in Scheme:

```
(define (c-to-f c)
  (+ (* 1.8 c) 32.0))

(define (f-to-c f)
  (/ (- f 32.0) 1.8))
```

What about goal: `cf(X,X).`

max(Z,X,Y)

CLP($\mathcal{R}$):

```
max(X,Y,X) :- X>=Y.
max(X,Y,Y) :- X<Y.
```

The same program in Scheme:

```
(define (max x y)
  (if (>= x y) x y))
```

In Miranda:

```
max x y = x, if x>=y
        = y, if x<y
```

What if we try goal `max(10,5,Y).` or `max(10,10,Y).`

append(Zs,Xs,Ys)

CLP($\mathcal{R}$):

```
append(Ys,[],Ys).
append([X|Zs],[X|Xs],Ys) :- append(Zs,Xs,Ys).
```

Scheme and Miranda versions of append:

```
(define (append x y)
  (if (null? x)
      y
      (cons (car x) (append (cdr x) y))))

append [] y = y
append (x:xs) y = x : append xs y
```

What about:

1. `append(Zs,[1,2,3],[10]).`
2. `append([1,2,3,4],[1,2],Ys).`
3. `append([1,2,3,4],Xs,Ys).`

CLP($\mathcal{R}$) – Definitions

A *user defined constraint* is of form $p(t_1, \ldots, t_n)$ where p is an n-ary predicate and t_i are expressions from constraint domain.

A *literal* is a primitive or user defined constraint.

A *goal* is a sequence of literals: $L_1, \ldots, L_m$.

The empty goal, $\square$, is trivially true.

A *rule* is of the form `A :- B.` where `A` is a user defined constraint and `B` is a goal. Meaning: "`B` implies `A`"

A *fact* is a rule with empty goal. Written: `A.`

Evaluation in CLP($\mathcal{R}$)

At the `clpr` prompt, you type a goal and `clpr` tries to find a solution.

E.g. `cf(X,Y), X > 100, Y < 200.`

CLP($\mathcal{R}$) Evaluation:

1. Start with empty *constraint store*
2. Consider first constraint of goal:
   (a) If primitive, push onto constraint store.
   (b) If user defined, *rewrite* using *rule*/*fact*.
   (if multiple rules, use first and save rest for backtracking)
3. Reduce constraint store.
4. If constraint store is *false* then backtrack.
5. If goal is empty then output *true* and contents of constraint store.
6. Continue with Step 2.

Rewriting Goals

Let the state of the solver be $\langle G \mid C \rangle$.

- G is goal: $L_1, \ldots, L_m$
- C is constraint store: $c_1, \ldots, c_n$

If L_1 is $p(t_1, \ldots, t_k)$ and we have rule: $p(s_1, \ldots, s_k) \vdash B$.

Let ρ be a renaming of variables (to be unique).

Rewrite state as: $\langle t_1 = \rho(s_1), \ldots, t_k = \rho(s_k), \rho(B), L_2, \ldots, L_m \mid C \rangle$

`max(Z,5,10)`

Recall:

```
max(X,X,Y) :- X > Y.
max(Y,X,Y) :- X <= Y.
```

Initial State: $\langle cf(X,Y), X > 100, Y < 200 \mid \square \rangle$

Derivation Tree

The solver is doing a depth first traversal of the derivation tree

- Root is $\langle G \mid \square \rangle$
- Success state is $\langle \square \mid C \rangle$ where C is satisfiable.
- Failure state is $\langle \square \mid C \rangle$ where C is unsatisfiable.
- Backtrack in derivation tree when we hit a failure state.
- On success, output C and ask to *retry*.

- Note: if C is ever unsatisfiable, then set $G = \square$

Choices:

- When more than one literal in goal, CLP($\mathcal{R}$) chooses first one.
- When more than one rule or fact for predicate, CLP($\mathcal{R}$) chooses first one first (others will be reached when back-tracking).

Combining Predicates

```
length([],0).
length([_|Xs],N) :- N>0, length(Xs,N-1).
```

The `_` is a place holder.

```
ones([]).
ones([1|Xs]) :- ones(Xs).
```

What is the difference between:

- `length(List,3), ones(List).`
- `ones(List), length(List,3).`

Moral: order of literals is goal matters!

What would happen if we fliped the order that ones was defined?
Moral: order of rules for predicate definition matters!

More Examples

To combine existing predicates, introduce new variables!

```
same_length(Xs,Ys) :- A=B, length(Xs,A), length(Ys,B).
```

Consider `member` predicate:

```
member(X,[X|Xs]).
member(X,[Y|Ys]) :- member(X,Ys).
```

In Miranda:

```
member x [] = False
member x (x:xs) = True
member x (y:xs) = member x xs
```

Note: In CLP($\mathcal{R}$) we do not worry about *false* case.