Contents

Reification – introductory example

- Consider variables `U in 0..9` and `V in 0..9`
- Try encoding the constraint `ex1geq5(U,V)`: exactly one of `U` and `V` is ≥ 5.
- A possible helper: `'x>=5<->b'(X, B)`: The boolean (i.e. 0 or 1 valued) variable `B` takes the value 1 iff `X #>= 5` holds
- Try implementing this helper constraint using an arithmetic constraint

```
'x>=5<->b'(X, B) :-  B #= X/5.
```

- Using the helper it is easy to implement `ex1geq5(U,V)`:

```
ex1geq5(U, V) :- 'x>=5<->b'(U, B1), 'x>=5<->b'(V, B2),
                 B1 + B2 #= 1.
```

- The `'x>=5<->b'(X, B)` helper constraint reflects (or reifies) the truth value of `X #>= 5` in the boolean variable `B`
- `library(clpfd)` supports reified constraints in general:

```
'x>=5<->b'(X, B) :-
    X #>= 5 #<=> B.
```

  This works without any limitation on the domain of `X`.

Reification – what is it?

- Reification = reflecting the truth value of a constraint into a 0/1-variable
- Form: `C #<=> B`, where `C` is a constraint and `B` is a 0/1-variable
- Example: `(X #>= 5) #<=> B` (*)
- Meaning: `C` holds if and only if `B=1`
- 4 implications:
  - If `C` holds, then `B` must be 1
  - If ¬`C` holds, then `B` must be 0
  - If `B=1`, then `C` must hold
  - If `B=0`, then ¬`C` must hold
- Not every constraint can be reified
  - Arithmetic formula constraints (`#=`, `#=<`, etc.) can be reified
  - The `X in` *`ConstRange`* membership constraint can be reified, e.g. rewrite (*) to a membership constraint: `(X in 5..sup) #<=> B`
  - Global constraints (e.g. `all_distinct/1, sum/3`) cannot be reified

Reification – what is it good for?

1. Use the 0/1-variables – that reflect the truth value of reified constraints – in propositional (logical) constraints
2. Use the 0/1-variables – that reflect the truth value of reified constraints – in arithmetic constraints
3. Combine multiple constraints with the help of propositional (logical) operators

1. Propositional constraints

- Propositional connectives allowed by SICStus Prolog CLPFD:

`#\ Q`	negation	`op(710,  fy, #\  ).`
`P #/\ Q`	conjunction	`op(720, yfx, #/\ ).`
`P #\ Q`	exclusive or	`op(730, yfx, #\  ).`
`P #\/ Q`	disjunction	`op(740, yfx, #\/ ).`
`P #=> Q`	implication	`op(750, xfy, #=> ).`
`Q #<= P`	implication	`op(750, yfx, #<= ).`
`P #<=> Q`	equivalence	`op(760, yfx, #<=>).`

- The operand of a propositional constraint can be
  - a variable `B`, whose domain automatically becomes `0..1`; or
  - an integer (`0` or `1`); or
  - a reifiable constraint; or
  - recursively, a propositional constraint.
- The propositional constraints are built from variables, integers and reifiable constraints using the above operators
- Example: `(X#>5) #<=> B1, (Y#>7) #<=> B2, B1 #\/ B2`

2. Using 0/1-variables in arithmetic constraints

- 0/1-variables can be used just like any other FD-variable, e.g., in arithmetic calculations
- Typical usage: counting the number of times a given constraint holds
- Example:

```
% pcount(L, N): list L has N positive elements.
pcount([X|Xs], N) :-
    (X #> 0) #<=> B,
    N1 #= N-B,
    pcount(Xs, N1).
pcount([], 0).
```

3. Combining constraints by means of propositional operators

- It is possible to combine multiple constraints with the help of propositional (logical) operators
  - Example:
    `(X#>5) #\/ (Y#>7)`
  - Handled by transforming it to a set of reifications and arithmetic constraints:
    `(X#>5) #<=> B1, (Y#>7) #<=> B2, B1+B2#>0`
  - Not possible with non-reifiable constraints
    - Example: `(X#>5) #\/ all_different([X,Y])` will lead to an error

Executing reified constraints

- Posting the constraint C `#<=> B` immediately implies `B in 0..1`
- The execution of C `#<=> B` requires three daemons:
  - When `B` is instantiated:
    - if `B=1`, post C; if `B=0`, post $\neg C$
  - When C is entailed (i.e. the store implies C), set `B` to `1`
  - When C is disentailed (i.e. $\neg C$ is entailed), set `B` to `0`

Entailment levels

Detecting entailment can be done with different levels of precision:

- A reified membership constraint *C* detects domain-entailment, i.e. `B` is set as soon as *C* is a consequence of the store
- A linear arithmetic constraint *C* is guaranteed to detect bound-entailment, i.e. `B` is set as soon as *C* is a consequence of the interval closure of the store
  - The interval closure is obtained by removing the 'holes' in the domains
  - Example:
    - Store: `X in {1,3}, Y in {2,4}, Z in {2,4}`
    - Interval closure: `X in {1,2,3}, Y in {2,3,4}, Z in {2,3,4}`
    - Constraint: `(X+Y#\=Z) #<=> B`
    - The store actually implies $X+Y \neq Z$ (odd+even≠even), but its interval closure does not
      $\Longrightarrow$ Result will be `B in 0..1` instead of `B=1`
- At the latest when a constraint becomes ground, its (dis)entailment is detected

Conversion from pure Prolog

Scheme for converting pure Prolog conditionals to CLPFD code using reification:

```
foo(...) :- NonrecTest.
foo(...) :-
    (   Cond -> Then
    ;   Else
    ),
    foo(...).
```

```
foo(...) :- NonrecTest#.
foo(...) :-
    Cond# #<=> B,
    B #=> Then#,
    #\ B #=> Else#,
    foo(...).
```

`Cond`, `Then`, `Else` and `NonrecTest`

- should contain solely arithmetic tests and operations
- are transformed to their constraint counterparts (mostly by simply inserting `#`-s): `Cond#`, `Then#`, `Else#` and `NonrecTest#`

Conversion from pure Prolog – example

```
% pcount(L, N): L has N positive elements.
```

```
pcount_pure(L,N) :-
    pcount_pure1(L,0,N).

pcount_pure1([],N,N).
pcount_pure1([X|Xs],N0,N) :-
    (    X>0 -> N1 is N0+1
    ;    N1=N0
    ),
    pcount_pure1(Xs,N1,N).
```

```
pcount_clpfd(L,N) :-
    pcount_clpfd1(L,0,N).

pcount_clpfd1([],N,N).
pcount_clpfd1([X|Xs],N0,N) :-
    X#>0 #<=> B,
    B #=> N1#=N0+1,
    #\ B #=> N1#=N0,
    pcount_clpfd1(Xs,N1,N).
```

Contents

Handling disjunctions

- Example: intervals $[x, x+5)$ and $[y, y+5)$ are disjoint: $(x+5 \leq y) \vee (y+5 \leq x)$
- Reification-based solution

```
| ?- domain([X,Y], 0, 6), X+5 #=< Y #\/ Y+5 #=< X.
        ⇒ X in 0..6, Y in 0..6
```

- Speculative solution

```
| ?- domain([X,Y], 0, 6), (X+5 #=< Y ; Y+5 #=< X).
        ⇒  X in 0..1, Y in 5..6 ? ;
        ⇒  X in 5..6, Y in 0..1 ? ; no
```

- Solution (hack?) with a clever use of arithmetics:

```
| ?- domain([X,Y], 0, 6),
     scalar_product([1,-1],[X,Y],#=,D,[consistency(domain)]),
     abs(D) #>= 5.
            ⇒ X in (0..1)\/(5..6), Y in (0..1)\/(5..6) ?
```

Constructive disjunction (CD)

- Assume a disjunction $C_1 \vee \ldots \vee C_n$
- Let $D(X, S)$ = the domain of X in store S
- The idea of constructive disjunction:
  - For each i, let S_i be the store obtained by adding C_i to S
  - Proceed with store S_U, the union of S_i, i.e. for all X, $D(X, S_U) = \cup_i D(X, S_i)$
- Algorithmically:
  - For each i:
    - post C_i
    - save the new domains of the variables
    - undo C_i
  - Narrow the domain of each variable to the union of its saved domains

Manipulating the domains – reflection predicates

- The representation of a constraint variable contains
  - the size of the domain
  - the lower bound of the domain
  - the upper bound of the domain
  - the domain as an FD-set (internal representation format)
- The above pieces of information can be obtained (in constant time) using
  - `fd_size(X, Size)`: `Size` is the size (number of elements) of the domain of `X` (integer or `sup`).
  - `fd_min(X, Min)`: `Min` is the lower bound of `X`'s domain; `Min` can be an integer or the atom `inf`
  - `fd_max(X, Max)`: `Max` is the upper bound of `X`'s domain (integer or `sup`).
  - `fd_set(X, Set)`: `Set` is the domain of `X` in FD-set format
- Further reflection predicates
  - `fd_dom(X, Range)`: `Range` is the domain of `X` in *ConstRange* format
  - `fd_degree(X, D)`: `D` is the number of constraints attached to `X`

FD-set vs. `ConstRange` format

```
| ?- X in 1..9, X#\=5, fd_set(X,S), fd_dom(X,R).
```

⇒ `S = [[1|4],[6|9]], R = (1..4)\/(6..9)`

FD-set is an internal format; user code should not make any assumptions about its representation

Manipulating the domains – FD-set operations

Some of the many useful operations:

- `is_fdset(Set)`: `Set` is a proper FD-set.
- `empty_fdset(Set)`: `Set` is an empty FD-set.
- `fdset_parts(Set, Min, Max, Rest)`: `Set` consists of an initial interval `Min..Max` and a remaining FD-set `Rest`. Can be used both for splitting and composing.
- `fdset_interval(Set, Min, Max)`: `Set` represents the interval `Min..Max`.
- `fdset_union(Set1, Set2, Union)`: The union of `Set1` and `Set2` is `Union`.
- `fdset_union(Sets, Union)`: The union of the list of FD-sets `Sets` is `Union`.
- `fdset_instersection/[2,3]`: analogous to `fdset_union/[2,3]`
- `fdset_complement(Set1, Set2)`: `Set2` is the complement of `Set1`.
- `list_to_fdset(List, Set)`, `fdset_to_list(Set, List)`: conversions between FD-sets and lists
- `X in_set Set`: Similar to `X in Range` but for FD-sets

Implementing constructive disjunction

- Computing the CD of a list of constraints `Cs` wrt. a *single* variable `Var`:

```
cdisj(Cs, Var) :-
    findall(S, (member(C,Cs),C,fd_set(Var,S)), Doms),
    fdset_union(Doms,Set),
    Var in_set Set.
```

- Usage:

```
| ?- domain([X,Y],0,6), cdisj([X+5#=<Y,Y+5#=<X], X).
        ⇒  X in(0..1)\/(5..6), Y in 0..6 ?
```

- CD is not a constraint, but a one-off pruning technique.
- As it interacts with other constraints, may improve on domain consistency:

```
| ?- domain([X,Y], 0, 20), X+Y #= 20, cdisj([X#=<5,Y#=<5],X).
        ⇒ X in(0..5)\/(15..20), ...
```

Shaving – a special case of constructive disjunction

- Basic idea: “What if” $X = v$? (…and hope for failure.) If this fails without labeling $\Longrightarrow X \neq v$, otherwise do nothing.
- Shaving an integer `V` off the domain of `X` is like a constr. disjunction $(X = v) \vee (X \neq v)$ w.r.t. X (but only the $X = v$ case is checked)

```
shave_value(V, X) :- \+ X = V, !, X in \{V}.
shave_value(_, _).
```

- Shaving all values in X's domain $\{v_1, \ldots, v_n\}$ is the same as performing a constructive disjunction for $(X = v_1) \vee \ldots \vee (X = v_n)$ w.r.t. X

```
shave_all0(X) :-     fd_set(X, FD), fdset_to_list(FD, L),
                     shvals(L, X).

shvals([], _).
shvals([V|Vs], X) :- shave_value(V, X), shvals(Vs, X).
```

- A variant using `findall`:

```
shave_all(X) :-      fd_set(X, FD), fdset_to_list(FD, L),
                     findall(X, member(X,L), Vs),
                     list_to_fdset(Vs, FD1), X in_set FD1.
```

An example for shaving, from a kakuro puzzle

- Kakuro puzzle: like a crossword, but with distinct digits 1–9 instead of letters; sums of digits are given as clues.

```
% L is a list of N distinct digits 1..9 with sum Sum.
kakuro(N, L, Sum) :-
     length(L, N), domain(L, 1, 9), all_distinct(L), sum(L,#=,Sum).
```

- Example: a 4 letter “word” `[A,B,C,D]`, the sum is 23, domains:

```
sample_domains(L) :- L = [A,_,C,D], A in {5,9}, C in {6,8,9}, D=4.
```

```
| ?- L=[A,B,C,D], kakuro(4, L, 23), sample_domains(L).
     ⇒  A in {5}\/{9}, B in (1..3)\/(5..8), C in {6}\/(8..9) ?
```

- Only `B` gets pruned:
  - 4 pruned because of `all_distinct`
  - 9 pruned because of `sum`

An example for shaving, from a kakuro puzzle

- Shaving 9 off `C` shows the value 9 for `C` is infeasible:

```
| ?- L=[A,B,C,D], kakuro(4, L, 23), sample_domains(L),
     shave_value(9,C). ⇒  ..., C in {6}\/{8} ?
```

- Shaving off the whole domain of `B` leaves just three values:

```
| ?- L=[A,B,C,D], kakuro(4, L, 23), sample_domains(L), shave_all(B).
     ⇒ ..., B in {2}\/{6}\/{8}, ... ?
```

- These two shaving operations happen to achieve domain consistency:

```
| ?- kakuro(4, L, 23), sample_domains(L), labeling([], L).
     ⇒ L = [5,6,8,4] ? ; L = [5,8,6,4] ? ; L = [9,2,8,4] ? ; no
```

When to perform shaving?

- Shaving may be applied repeatedly, until a fixpoint (may not pay off)
- Shaving is normally done during labeling. To reduce its costs, one may:
  - limit it to variables with small enough domain (e.g. of size 2)
  - perform it only after every n^{th} labeling step (requires global variables)
- Example:

```
% Label the variables in Vars; after every Nth value assignment,
% shave the domain of variable X
labeling_with_shaving(X,N,Vars) :-
    bb_put(i,0),
    labeling([value(shave_during_labeling(X,N))],Vars).

% Auxiliary predicate, called by labeling in every iteration.
% X and N are given in the call to labeling, V is the next variable
shave_during_labeling(X,N,V,_Rest,_BB0,_BB) :-
    labeling([],[V]),
    bb_get(i,I),
    (    I<N -> I1 is I+1, bb_put(i,I1)
    ;    shave_all(X), bb_put(i,0)
    ).
```