In the Lecture Series Logic Programming and Constraints

Constraints: a Preview

Presented by Stéphane Bressan

Prolog II *geler/2* (Freeze)

```
:- writeln(X), writeln(2), X = 1.
```


Prolog II *geler/2* (Freeze)

```
:- freeze(X, writeln(X)), writeln(2), X = 1.
```

The goal writeln(X) is frozen until X is instantiated

The output is 21

But this does not work in ECLiPSe

Delay Clause

```
delay mywriteln(X) if var(X).
mywriteln(X) :- writeln(X).
```

A delay clause is very similar to a normal Prolog clause. It has the form:

delay <Head> if <Body>.

A predicate may have one or more delay clauses. They have to be textually before and consecutive with the normal clauses of the predicate they belong to.

When a procedure with delay clauses is called, then the delay clauses are executed before executing the procedure itself. If one of the delay clauses succeeds, the call is suspended, otherwise they are all tried in sequence and, if all delay clauses fail, the procedure is executed as usual.

Delay clauses are used with pattern matching

For instance with:

delay p(a, X) if var(X).

the variables in the call cannot be bound by the matching, e.g. the head of the delay clause does not match the goal p(A, b) but it matches the goal p(a, b).

Freeze with Delay Clauses

```
/* freeze.pl */

delay freeze(X, Goal) if var(X).
freeze(_, Goal) :- call (Goal).
```


Suspensions

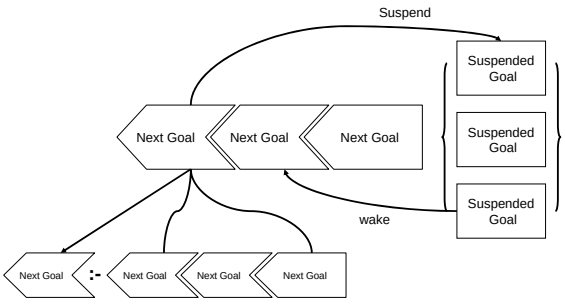
- **suspend(+Goal, +Priority, +CondList)**
- Priority is an integer between 1 and 12 that determines the priority with which the Goal will be scheduled when woken (1 being the most urgent and 12 the least urgent), or it is 0, in which case the priority defaults to the priority setting of the predicate which is called in Goal.
- CondList is one term or a list of terms of the form:

  Vars->Cond.

The condition Cond is the name of a predefined suspension list (can also be library or user defined triggers).

- inst: wake when a variable gets instantiated
- bound: wake when a variable gets instantiated or bound to another variable
- constrained: wake when a variable gets instantiated or bound to another variable or becomes otherwise constrained

ECLiPSe Resolvant



Be careful if you use cuts together with suspensions!

Freeze with Suspend

```
:- suspend(writeln(X), 1, X->inst), write(2),
   X=1.-
```


Freeze with Suspend

```
/* freeze.pl */

freeze(X, Goal):-
   suspend(Goal, 1, X-> inst).
```


Tracing Variables Life

```
report(X) :-
   suspend(report1(X), 1, X->constrained).

report1(X) :-
   ( nonground(X) ->
   (writeln(constrained(X)),
   term_variables(X, L),
   suspend(report1(L), 1, L->constrained) )
   ;
   writeln(instantiated(X)) ).
```


Following a variable's Life

```
:- report([X]), X = f(Y, Z), Z=g(H), Y = 1.
```


Attributed Variables

Attributed variables or metaterms are variables that have attributes, e.g:

X{attribute:value}

It is necessary to redefined unification for theses variables by defining a handler (the handler is called when the variable is unified)

Attribute

- **meta_attribute(+Name, +Handlers)**
  - Declares the variable attribute Name with the corresponding handlers and the procedures that implement them (unify, print, etc.)

Example

/* Attributed Variables */

```
:- meta_attribute(enum,
        [unify:unify_enum/2,
        print:print_enum/2]).
```


Enumeration Domain

As an example, let us implement variables of enumerable types using attributes.

We choose to represent these variable as attributed variables whose attribute is a list of possible values

Enumeration Domain

```
:- module(enum).
:- meta_attribute(enum, [unify:unify_enum/2, print:print_enum/2]).
:- import setarg/3 from sepia_kernel.

:-export enum/2.

enum(X, Y):- add_attribute(X, enum(Y)).

unify_enum(_, Attr) :- var(Attr).
unify_enum(Term, Attr) :- compound(Attr), unify_term_enum(Term, Attr).

unify_term_enum(Value, enum(ListY)) :-nonvar(Value), memberchk(Value, ListY).
unify_term_enum(Y{AttrY}, AttrX) :-  -?-> unify_enum_enum(Y, AttrX, AttrY).

unify_enum_enum(_, AttrX, AttrY) :-
  var(AttrY), % no attribute for this extension
  AttrX = AttrY. % share the attribute
unify_enum_enum(Y, enum(ListX), AttrY) :-
    nonvar(AttrY), AttrY = enum(ListY), intersection(ListX, ListY, ListXY),
    ( ListXY = [Val] -> Y = Val ; ListXY \= [], setarg(1, AttrY, ListXY)).

print_enum(_{enum(V)}, A) :- -?-> A = V.
```


Finite Domains Library

```
:- lib(fd).
```


Finite Domains

- **?Vars :: ?Domain**
  - Terms in Vars have the domain Domain.

Example:

```
:- X :: 23..54.
:- X :: 23..54, Y :: 36..100.
:- X :: 23..54, Y :: 36..100, X = Y.
```

Finite Domains

- **?X #= ?Y**
  - X is equal to Y
  - This constraints states that the two linear terms are equal.
  - It is activated whenever the maximum or minimum of a domain variable is updated that might require updating other domains. When propagating domain updates, the system takes into account only maximum and minimum values of the whole domain and makes sure that these values are consistent with those of other domain variables.
  - If one of the arguments is a non-linear polynomial, this predicate delays until it becomes linear.

Global Constraints

- **alldifferent(?List)**
  - The elements of the list List are pairwise different.

Finite Domains

```
/* */
/* domain */
d(0). d(1). d(2). d(3). d(4). d(5). d(6). d(7). d(8). d(9).
d(X, Y, Z):- d(X),  X >= Y,  X =< Z.
different([X|L]):- different(X, L), different(L).
different([]).
different(X, L):- member(X, L), !, fail.
different(_, _).

/* send */
send(L):-
      /* Variables */
      L=[S,E,N,D,M,O,R,Y],

      /* Generate */
      d(E, 0, 9) , d(N, 0, 9) , d(D, 0, 9) , d(O, 0, 9) , d(R, 0, 9) , d(Y, 0, 9),
      d(S, 1, 9), d(M, 1, 9),

      /* Test */
      different(L),
      1000*S + 100*E + 10*N + D +
    1000*M + 100*O + 10*R + E =:=
    10000*M + 1000*O + 100*N + 10*E + Y.
```


Finite Domains

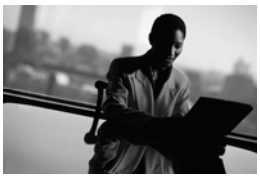
```
/* */
:- lib(fd). % finite domains library
/* domain */
d(X, Y, Z):- D :: X..Y.

/* send */
send(L):-
    /* Variables */
    L=[S,E,N,D,M,O,R,Y],

    /* Constraint */
    d(E, 0, 9) , d(N, 0, 9) , d(D, 0, 9) , d(O, 0, 9) , d(R, 0, 9) , d(Y, 0, 9),
    d(S, 1, 9), d(M, 1, 9),
    alldifferent(L),
    1000*S + 100*E + 10*N + D +
    1000*M + 100*O + 10*R + E #=
    10000*M + 1000*O + 100*N + 10*E + Y.

    /* Enumerate */
    labeling(L).
```


Finite Domains

```
/* */
:- lib(fd). % finite domains library
/* domain */
d(X, Y, Z):- D :: X..Y.

/* send */
send(L):-
      /* Variables */
      L=[S,E,N,D,M,O,R,Y],

      /* Constraints */
      d(E, 0, 9) , d(N, 0, 9) , d(D, 0, 9) , d(O, 0, 9) , d(R, 0, 9) , d(Y, 0, 9),
      d(S, 1, 9), d(M, 1, 9),
      alldifferent(L),
      d(R1, 0, 1), d(R2, 0, 1), d(R3, 0, 1),
      D+E #= Y + 10*R1,
      R1+N+R #= E + 10*R2,
      R2+E+O #= N + 10*R3,
      R3+S+M #= O +10*M,

/* Enumerate */
      labeling(L).
```


Credits

Clipart and media are licensed from Microsoft Office Online Clipart and Media



Copyright © 2008 by Stéphane Bressan