A PROLOG DEMAND DRIVEN COMPUTATION INTERPRETER

Luís Moniz Pereira

Departamento de Informática
Universidade Nova de Lisboa
Quinta da Torre
2825 Monte da Caparica
PORTUGAL

This interpreter realizes the demand driven computation process described in [1]

Notes:

Predicate "stream" accepts a functional predicate goal R *and delivers a stream* X *of results, where difference lists are used to represent streams. After each value in the stream is produced it pauses, and displays the stream. If a* <CR> *is given it continues, if a* <space> *is given it shows the calls waiting to be demand driven and continues. Example call:* stream(p=:X).

Predicate "up_to" produces up to N *values of a stream for a given call. Example call:* up_to(3,conc([1,2],[3,4])=:X).

Predicate "#" evaluates any predicate call. If the predicate is functionally defined, it evaluates it recursively until a list is produced, where the head of the list, if any, contains the first result of evaluating the call, and the tail a call to a functional predicate for producing the next result. The call [] *evaluates to the empty list.*

To do so, it uses predicate "@", which picks up a clause for a functionally defined predicate and evaluates its body if there is one. However, if any argument is demand driven and is not yet evaluated, no clause can be picked up and "@" will evaluate the demand driven arguments, and return to "#" the call with its arguments evaluated.

```
?- op(230,xfx, =:).      /*  functional relations   */
?- op(240,fx , @ ).      /*  access to program clauses   */
?- op(240,fx , #).       /*  evaluation   */
?- op(254,xfx,<-).       /*  functional relations' conditions   */

/*   USER INTERFACE    */

stream(R=:X) :- s(R,X-X).

s(R,X-Z) :- # R=:A , !, ( (A=[] ; A=[_|T], list(T) ), Z=A          ;
                           A=[V|T], Z=[V|Y], show(T,X), s(T,X-Y) ).

show(T,X) :- write(X), get0(C), ( C=32, write(T), skip(10), nl ; C=10 ), nl.

up_to(N,R=:[V|Y]) :- N>0, # R=:[V|S] , !, M is N-1, up_to(M,S=:Y).
up_to(_,_=:[]).

/*   INTERPRETER    */

# R=:S  :-  ( list(R), R=S ; @ R=:A , # A=:S ) .
# (A,B)  :-  # A , # B .
# G      :-  G .

list([]).
list([_|_]).

/*   access to non-unit and unit functional predicate clauses, regular
     Prolog clauses, and system predicates                          */

@ G :- (G<-C), # C.              /*   non-unit clauses   */
@ G :- G.                        /* unit clauses, Prolog, and system */

/*  user specified info about demand driven arguments  */

@ conc(A,X) =: conc(EA,X)        :-  # A=:EA.
@ select(N,A) =: select(N,EA)    :-  # A=:EA.
@ sift(A) =: sift(EA)            :-  # A=:EA.
```

```
@ filter(X,A) =: filter(X,EA)          :- # A=:EA.
@ merge(A,B) =: merge(EA,EB)           :- # A=:EA, # B=:EB.
@ mul(A,B) =: mul(A,EB)                :- # B=:EB.

/*  PROGRAMS  */

/*  conc  */

conc([],X)   =: X.
conc([X|Y],U) =: [X|conc(Y,U)].

/*  bounded buffer  */

bounded_buffer(WS,RS) =: AS <- bmerge(WS,RS,0,S1), buffer(S1,U-U)=:AS.

bmerge([write(X)|WS],RS,I,[write(X)|AS])  :- I<5, K is I+1, bmerge(WS,RS,K,AS).
bmerge(WS,[read|RS],I,[read|AS])          :- I>0, K is I-1, bmerge(WS,RS,K,AS).
bmerge(_,[],_,[]).

buffer([write(X)|S],V-[X|W])  =: buffer(S,V-W).
buffer([read|S],[X|V]-W)      =: [X|buffer(S,V-W)].
buffer([],_-[])               =: [].

/*  infinite list of integers  */

intfrom2 =: inc(2).

inc(X) =: [X|inc(K)] <- K is X+1.

n_integers(N) =: Y <- intfrom2=:X, select(N,X)=:Y.

select(0,_)      =: [].
select(N,[X|Y]) =: [X|select(K,Y)] <- N>0, K is N-1.

/*  primes  */

primes =: sift(intfrom2).

sift([X|Y]) =: [X|sift(filter(X,Y))].

filter(X,[Y|Z]) =: [Y|filter(X,Z)] <- Y mod X=\= 0.
filter(X,[Y|Z]) =: [Y|filter(X,Z)] <- Y mod X=:= 0.

/*  quicksort  */

qs([])     =: [].
qs([X|Y]) =: conc(qs(Y1),[X|qs(Y2)]) <- part(X,Y,Y1,Y2).

part(X,[H|T],[H|S],R) :- H=<X, part(X,T,S,R).
part(X,[H|T],S,[H|R]) :- H>=X, part(X,T,S,R).
part(_,[],[],[]).

/*  cyclic network of agents  */

p=:Y <- merge(mul(2,[1|Y]), merge(mul(3,[1|Y]), mul(5,[1|Y]))) =: Y.

merge([X|Y],[U|V]) =: [X|merge(Y,[U|V])] <- X<U.
merge([X|Y],[U|V]) =: [U|merge([X|Y],V)] <- X>U.
merge([X|Y],[U|V]) =: [X|merge(Y,V)]     <- X=U.

mul(X,[Y|Z])  =: [W|mul(X,Z)] <- W is X*Y.
```

REFERENCE

[1] HANSSON, A.; HARIDI, S.; TÄRNLUND, S.-Å: "Properties of a Logic Programming Language" in "Logic Programming" (K. Clark and S.-Å. Tärnlund eds.), Academic Press 1982, and also report 8/81, Computing Science Dept., Uppsala University, Sweden.