ALL SOLUTIONS

Luís Moniz Pereira
António Porto

Departamento de Informática
Universidade Nova de Lisboa
1899 Lisboa, Portugal

Any Prolog programmer sooner or later feels the need for a predicate capable of producing the set of all solutions to a given problem.

Those not fortunate enough to have a Prolog system offering such a predicate as a built-in feature usually resort to ad-hoc techniques for achieving its effect in a particular setting. We show a compact, reasonably efficient and sound implementation of such a predicate, that anybody can use since it is written in Prolog itself.

The predicate is

```
all ( T , G , L )
```

and it reads " all instances of the term T *for which the goal* G *is satisfied are the members of list* L *".* L *is required to be non-empty, so* 'all' *fails if* G *has no solution.*

The term T *is just a template for building* L*, so free variables within will not be bound upon execution of* 'all'.

G *can be any valid goal expression in Prolog, including* 'cut's *(which only affect the evaluation of* G *within the evaluation of* 'all'*) and* 'all's *(whose nesting is very useful for structuring sets of solutions). Furthermore,* G *can be of the special form*

```
G1 same T1
```

where G1 *is any goal expression and* T1 *is any term. This variant allows the distinction between two roles of the free variables appearing in* G *but not in* T*:*

If G *is not of the* 'same' *type, the different solutions of* G *for which instances of* T *are put in* L *can correspond to different instantiations of any free variable in* G*, and* 'all' *acts as a deterministic predicate.*

If, however, G *is of the form* 'G1 same T1'*, the different solutions of* G1 *for which instances of* T *are put in* L *must correspond to the same instance of* T1*, which remains enforced within the execution of* 'all'*; backtracking into* 'all' *will produce another instance of* T1 *and another corresponding list* L*, until no more solutions to* G1 *exist and 'all' finally fails.*

All this is best seen with an example. Suppose we have the following micro data base:

```
drinks(john,tea,hot).
drinks(john,milk,hot).
drinks(john,milk,cold).
drinks(john,milk,warm).
drinks(john,beer,cold).
drinks(john,wine,cold).

drinks(bill,milk,cold).
drinks(bill,beer,cold).
drinks(bill,beer,warm).

drinks(joe,beer,cold).
drinks(joe,wine,cold).
drinks(joe,wine,warm).
drinks(joe,tea,hot).
drinks(joe,tea,warm).
drinks(joe,tea,cold).
```

Some natural language questions follow, along with the corresponding formulation in terms of the 'all' *predicate, and its solution(s).*

Who drinks?

```
all( P, drinks(P,_,_), X).

X = [ john, bill, joe].
```

Who drinks the same drink?

```
all( P, drinks(P,D,_) same D, X).

X = [ john, joe] , D = tea ;
X = [ john, bill] , D = milk ;
...
```

Who drinks each drink?

```
all( D-Ps, all( P, drinks(P,D,_) same D, Ps), X).

X = [ tea-[john,joe], milk-[john,bill], ... ].
```

Who drinks (and at which temperatures) each drink?

```
all( D-PT, all( P:Ts, all( T, drinks(P,D,T) same (D,P), Ts) same D, PT), X).

X = [ tea-[ john:[hot], joe:[hot,warm,cold]], milk-[ john:[hot,...],...],...].
```

The Prolog definition of 'all' *now follows:*

```
?- op(50,xfx,same).

all(T,G same X,S) :- !, all(T same X,G,Sx), produce(Sx,S,X).

all(T,G,S) :- asserta(one(end)), solve(G), asserta(one(T)), fail.

all(T,G,S) :- set(S).

solve(G) :- G.

set(S) :- build(S,[]), ( S=[], !, fail ;
                          asserta(set(S)), fail ).
```

```
set(S) :- retract(set(S)).

build(NS,S) :- retract(one(X)), ( nonvar(X), X=end, NS=S ;
                                  join(S,X,XS), build(NS,XS) ), !.

join(S,X,S) :- in(S,X).

join(S,X,[X|S]).

in([X|_],X).

in([_|S],X) :- in(S,X).

produce([T1 same X1|Tn],S,X) :- split(Tn,T1,X1,S1,S2),
                                 ( S=[T1|S1], X=X1 ;
                                   !, produce(S2,S,X) ).

split([],_,_,[],[]).

split([T same X|Tn],T,X,S1,S2) :- split(Tn,T,X,S1,S2).

split([T1 same X|Tn],T,X,[T1|S1],S2) :- split(Tn,T,X,S1,S2).

split([T1|Tn],T,X,S1,[T1|S2]) :- split(Tn,T,X,S1,S2).
```

Some remarks should be made:

1) The non-logical predicates 'asserta' *and* 'retract' *are called from* 'all' *and* 'build' *just to implement a stack where solutions are kept during backtracking within* G.

2) The predicate 'get' *is defined so as to recover the space used by the recursive execution of* 'build', *instead of calling* 'build' *directly from* 'all'.

3) 'solve' *is necessary, so that any* 'cut's *within* G *do not affect the clauses for* 'all'.

4) There is some time lost in keeping L *free of repeated elements. For applications where this feature is not necessary one can define a faster* 'all' *by changing the clauses for* 'build', 'produce' *and* 'split' *as follows:*

```
build(NS,S) :- retract(one(X), ( nonvar(X), X=end, NS=S ;
                                 build(NS,[X|S]) ), !.

produce([T1 same X1|Tn],S,X) :- split(Tn,X1,S1,S2),
                                 ( S=[T1|S1], X=X1 ;
                                   !, produce(S2,S,X) ).

split([],_,[],[]).

split([T1 same X|Tn],X,[T1|S1],S2) :- split(Tn,X,S1,S2).

split([T1|Tn],X,S1,[T1|S2]) :- split(Tn,X,S1,S2).
```

5) Where, using DECsystem-10 Prolog's predicate 'setof', *one would write*

setof(X, p(X,Y), S) *and* setof(X, Y^p(X,Y), S) ,

we would write, respectively,

all(X, p(X,Y) same Y, S) *and* all(X, p(X,Y), S) ,

with the difference that we do not sort S.

For natural language processing we prefer our version, since hidden variables do not have to be existentially quantified explicitly.

6) This 'all' *has been tested and used extensively.*