

Chapter 10

Co-Routines and Constraint Solving

Pure logic programs are independent of the selection rule. This means for example, that in the Prolog program below, we can chose either to select $p(X,Z)$ first or $q(Z,Y)$ first without affecting the logical meaning of the program.

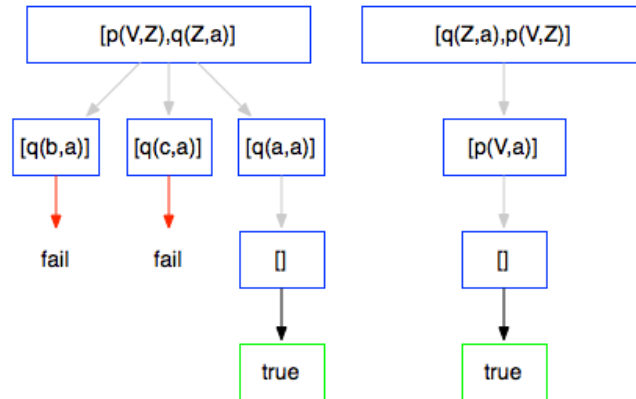
```
pq(X,Y) :- p(X,Z), q(Z,Y).  
p(a,b).  
p(b,c).  
p(c,a).  
  
q(a,a).  
q(b,d).
```

However, while the order of selecting literals does *not* affect the semantics, it does affect performance. Indeed, take the query $pq(a,V)$, leading to the subgoal $p(a,Z), p(Z,V)$. Here it is better to perform $p(a,Z)$ first. However, for the query $pq(V,a)$ it is better to perform the $q(Z,a)$ first (see Figure 10.1):

- when selecting $q(V,Z)$ first we have a choice point with three possibilities; i.e., we may evaluate p up to three times.
- when selecting $p(Z,a)$ first, we get exactly one solution and evaluate q only once.

The situation is of course much more dramatic when the relations p and q become large. In the extreme case, one order may find a solution quickly, while the other does not-terminate. For example, let us revisit `append`, and add a predicate for appending two lists:

```
dapp(X,Y,Z,R) :- app(X,Y,XY), app(XY,Z,R).  
app([],R,R).  
app([H|X],Y,[H|Z]) :- app(X,Y,Z).
```

Figure 10.1: Influence of the order of selection for $pq(V, a)$

If we take the query `dapp([1],[2],[3],R)` then the Prolog left-to-right selection rule is perfect. However, for backwards use such as `dapp(X,Y,Z,[1,2,3])` it is far from perfect. As another example, take the query `dapp(X,Y,[3],[2])`. It should be clear that there is no solution. However, the Prolog left-to-right selection rule leads to an infinite SLD-tree (see Figure 10.2).

One solution would be to write two version of `dapp`:

```

dapp(X,Y,Z,R) :- app(X,Y,XY), app(XY,Z,R).
dapp_back(X,Y,Z,R) :- app(X,Y,XY), app(XY,Z,R).

```

However, we may require more than one version (what if we know the middle argument first,...). Also, if our predicate uses other predicates, then we may have to write (and maintain) multiple versions of each of those.

The programming language Mercury gives the user the possibility to provide modes for predicates, and the compiler will automatically generate different versions with different orders as required.

```

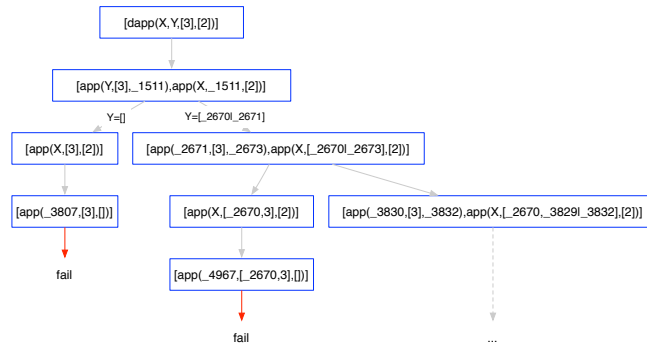
:- mode app(in, in, out).
:- mode app(out, out, in).
:- mode dapp(in,in,in,out).
:- mode dapp(out,out,out,in).

```

Modern Prolog systems provide a more dynamic way to solve this problem: block and when declarations.

10.1 When Declarations

Let us look at when declarations first:

Figure 10.2: Infinite SLD-tree for $dapp(X, Y, [3], [2])$

```

pq(X, Y) :-
    when((nonvar(X); nonvar(Z)), p(X, Z)),
    when((nonvar(Z); nonvar(Y)), q(Z, Y)).

app(X, Y, Z) :- when((nonvar(X); nonvar(Z)), app2(X, Y, Z)).
app2([], R, R).
app2([H|X], Y, [H|Z]) :- app(X, Y, Z).

| ?- dapp(X, Y, [3], [2]).
no

```

10.2 Block Declarations

Block declarations are another way to influence the selection rule. They are more limited in power, but are often more convenient to declare, because we declare them per predicate rather than at each call site. Also, at least in SICStus Prolog, block declarations are much more efficient.

For our first example, we would provide the following block declarations instead of the `when` annotations:

```

:- block p(-, -).
:- block q(-, -).

```

These indicate that `p` and `q` should be blocked, if both the first and second argument are not instantiated (i.e., are a variable). Thus, the calls `p(X, Y)` or `p(X, X)` will block until at least one of its arguments is no longer a variable. The calls `p(a, b)`, `p(X, b)`, or `p(a, X)` will not block and be executed straightaway.

For our append program, we could provide the following block declarations, in place of the `when` annotations:

```

:- block app(-, ?, -).

```

This declaration says that `app` should block when both the first and third argument are variables. The value of the second argument does not matter, as indicated by the question mark.

10.3 From Generate-Test to Test-Generate

Generate and test is an AI (artificial intelligence) search method to solve complicated problems. The idea is to have one generator which generates candidate solutions to our problem, and a tester which checks whether we have found a solution. In Prolog this can be encoded as two predicates:

```
search(Sol) :- generate(Sol), test(Sol).
```

It is often relatively easy to write a generate and test algorithm, especially in Prolog. The main problem is of course performance. Here, we will show how co-routines can dramatically improve the performance by moving the tests as early as possible.

```
sorted([]).
sorted([H|T]) :- sorted2(T,H).
:- block sorted2(-,?), sorted2(?,-).
sorted2([],_).
sorted2([H|T],Prev) :- sorted3(H,T,Prev).
:- block sorted3(-,?,?).
sorted3(H,T,Prev) :- Prev<H, sorted2(T,H).
```

```
permute([], []).
permute(T, [X|PermRest]) :- del(T,X,Rest),
permute(Rest,PermRest).
```

```
del([H|T],H,T).
del([H|T],X,[H|R]) :- del(T,X,R).
```

```
| ?- p1([55,33,2,1,7,8,9,32,101,22,1022],R).
```

```
calling: permsort1([55,33,2,1,7,8,9,32,101,22,1022],_638)
exit: permsort1([55,33,2,1,7,8,9,32,101,22,1022],[1,2,7,8,9,22,32,33,55,101,1022])
Runtime: 5690 ms
Total Runtime: 5690 ms
Walltime: 5700 ms
R = [1,2,7,8,9,22,32,33,55,101|...] ?
yes
| ?- p2.
```

```
calling: permsort2([55,33,2,1,7,8,9,32,101,22,1022],_616)
exit: permsort2([55,33,2,1,7,8,9,32,101,22,1022],[1,2,7,8,9,22,32,33,55,101,1022])Run
```

```
Total Runtime: 0 ms
Walltime: 0 ms
yes
```

10.4 Other Uses and Tricks

10.4.1 Debugging

Debugging programs with coroutines can be a major challenge. However, coroutines themselves can be handy when debugging:

```
:- block watch_variable(?,-).
watch_variable(Name,Value) :-
    print('Variable '), print(Name),
    print(' instantiated with '), print(Value),nl.
```

10.4.2 Forcing all solutions

The problem with the blocking versions of `pq` or `dapp` above is that we cannot use the program to force enumeration of solutions if all arguments are variables. A trick is to add another dummy variable, which acts as a waitflag: if it is instantiated then the predicate should no longer block:

```
pq(X,Y,WF) :- p(X,Z,WF), q(Z,Y,WF).

:- block p(-,-,-).
p(a,b,_).
p(b,c,_).
p(c,a,_).

:- block q(-,-,-).
q(a,a,_).
q(b,d,_).
```

We can now force enumeration with `pq(X,Y,WF), WF=1`.

```
| ?- pqc(X,Y,WF), WF=1.
X = a,
Y = d,
WF = 1 ? ;
X = c,
Y = a,
WF = 1 ? ;
no

| ?- pqc(X,a,WF),WF=1.
X = c,
```

```
WF = 1 ? ;
no
```

Exercise 10.4.1 What is the difference between calling `ppc(X, a, WF)`, `WF=1` and `ppc(X, a, 1)`? Which one is better? Below is the trace for both calls:

```
% trace
| ?- ppc(X,a,WF),WF=1.
    1      1 Call: ppc(_496,a,_530) ?
    -      - Block: p(_1276,_1282,_1288)
    2      2 Call: q(_1282,a,_1288) ?
    -      - Unblock: p(_1276,a,_1288)
    3      3 Call: p(_1276,a,_1288) ?
    3      3 Exit: p(c,a,_1288) ?
?      2      2 Exit: q(a,a,_1288) ?
?      1      1 Exit: ppc(c,a,_1288) ?
    4      1 Call: _1288=1 ?
    4      1 Exit: 1=1 ?

X = c,
WF = 1 ?
yes
% trace
| ?- ppc(X,a,1).
    1      1 Call: ppc(_496,a,1) ?
    2      2 Call: p(_496,_1154,1) ?
?      2      2 Exit: p(a,b,1) ?
    3      2 Call: q(b,a,1) ?
    3      2 Fail: q(b,a,1) ?
    2      2 Redo: p(a,b,1) ?
?      2      2 Exit: p(b,c,1) ?
    4      2 Call: q(c,a,1) ?
    4      2 Fail: q(c,a,1) ?
    2      2 Redo: p(b,c,1) ?
    2      2 Exit: p(c,a,1) ?
    5      2 Call: q(a,a,1) ?
    5      2 Exit: q(a,a,1) ?
    1      1 Exit: ppc(c,a,1) ?

X = c ?
yes
```

10.4.3 Threading Waitflags

It is quite common that we only want to perform certain computations if a given choice point has been resolved.

E.g., when we have a programming language with an if-statement, we may wish to examine the body only if it has been decided which way the branch goes. The trick is to thread the output of the one checking predicate as argument to the next predicate, and then block until this argument is instantiated.

```
:- block check_instruction(-).
check_instruction(if(Test,Then,Else)) :-
    check_if(Test,Result), check_body(Result,Then,Else).
```

```
:- block check_body(-,?,?).
check_body(true,Then,_) :- check_instruction(Then).
check_body(false,_,Else) :- check_instruction(Else).
```

10.5 Towards Constraint Logic Programming

We will now use co-routines to implement better support for arithmetic operators. First, let us write a simple predicate for addition, using the built-in `is/2` predicate:

```
plus1(X,Y,R) :- R is X+Y.
```

The `is/2` behaves very much like a construct in imperative language: the right-hand side must be known and if the left-hand side is

- a variable then this corresponds to an assignment in an imperative language,
- a value, then this corresponds to a test in an imperative language.

The following corresponds to $R1 = 1+2$; $R2 = R1+3$; $R = R2+4$; in an imperative language like Java:

```
| ?- plus1(1,2,R1), plus1(R1,3,R2), plus1(R2,4,R).
R = 10,
R1 = 3,
R2 = 6 ?
yes
```

We cannot change the order of the calls, `is/2` cannot be used when the right-hand side is not completely known:

```
| ?- plus1(R2,4,R), plus1(R1,3,R2), plus1(1,2,R1).
! Instantiation error in argument 2 of is/2
! goal: _101 is _104+4
```

10.5.1 Passive Constraints

We can remedy this shortcoming by using co-routines, to ensure that the `is/2` is only called when it is sufficiently instantiated:

```
:- block plus2(-,?,?), plus2(?,-,?).
plus2(X,Y,R) :- R is X+Y.
```

```
| ?- plus2(2,2,4).
yes
| ?- plus2(2,2,5).
no
```

```
| ?- plus2(R2,4,R), plus2(R1,3,R2), plus2(1,2,R1).
R = 10,
R1 = 3,
R2 = 6 ?
yes
```

The above implementation is not yet fully satisfactory. It will check an addition if all arguments are known:

```
| ?- plus2(1,R,R2), plus2(5,7,R2).
R2 = 12,
user:plus2(1,R,12) ?
yes
```

The only case it will actively instantiate (compute) an argument is if the first two arguments are known. This is better than a completely passive version of addition, which would not even do that:

```
:- block plus2(-,?,?), plus2(?,-,-), plus2(?,?,-).
plus2(X,Y,R) :- R is X+Y.
```

10.5.2 Active Constraints

However, we can make the addition much more active, by also instantiating the first two arguments, if enough information is present.

```
:- block plus3(-,?,-), plus3(?,-,-), plus3(-,-,?).
plus3(X,Y,R) :- ( var(X) -> X is R-Y
                  ; var(Y) -> Y is R-X
                  ; otherwise -> R is X+Y
                  ).
```

```
| ?- plus3(R2,4,R), plus3(R1,3,R2), plus3(1,2,R1).
R = 10,
R1 = 3,
R2 = 6 ?
yes
| ?- plus3(1,R,R2), plus3(5,7,R2).
R = 11,
R2 = 12 ?
yes
```

Note that we have used non-declarative features such as `var/1` and the if-then-else construct. Still, the overall predicate `plus3` can still be understood as a purely logical predicate encoding addition. In fact, to the outside world, it is more declarative than the original `plus1`!

10.5.3 Combining with Test-Generate

Let us try and solve a magic square using the above and using test and generate.

First, we write a generator for squares:

```
square(N,From,To,Square) :- length(Square,N), square2(Square,N,From,To).

square2([],_,_,_).
square2([Row|T],N,From,To) :- length(Row,N), square2(T,N,From,To),
    in_range(Row,From,To).

in_range([],_,_).
in_range([H|T],From,To) :- enumerate(H,From,To), in_range(T,From,To).

enumerate(H,From,To) :- ground(H), !, H>=From, H<=To.
enumerate(H,From,To) :- From =< To,
    (H=From ; F1 is From+1, enumerate(H,F1,To)).
```

The purpose of the first clause of `enumerate` may be more difficult to grasp; why should a variable in the square be instantiated? The answer is that we will use a test and generate approach using active constraints: hence, the tester may already determine the value of certain variables.

To write the tester, we will use an auxiliary predicate to sum up the numbers in a list using our `plus3` predicate:

```
:- block sum_list(-,?,?).
sum_list([],R,R).
sum_list([H|T],A,R) :- plus3(H,A,A2), sum_list(T,A2,R).
```

We also need an auxiliary predicate to check that a list of numbers contains only distinct elements:

```
:- block all_different(-).
all_different([]).
all_different([H|T]) :- dif_tail(T,H), all_different(T).

:- block dif_tail(-,?).
dif_tail([],_).
dif_tail([H|T],X) :- dif(H,X), dif_tail(T).
```

Note, we have made use of the built-in `dif/2` predicate: it will already block until it can determine the truth value.

[Insert some examples of `dif/2`.]

Let us now complete the simple tester for a 3 by 3 magic square of sum 15:

```
tester3([ R1, R2 , R3 ] ) :-
    sum_list(R1,0,15),
    sum_list(R2,0,15),
```

```

sum_list(R3,0,15),
R1 = [A1,A2,A3], R2 = [B1,B2,B3], R3 = [C1,C2,C3],
sum_list([A1,B1,C1],0,15),
sum_list([A2,B2,C2],0,15),
sum_list([A3,B3,C3],0,15),
sum_list([A1,B2,C3],0,15), % diagonal 1
sum_list([A3,B2,C1],0,15), % diagonal 2
all_different([A1,A2,A3,B1,B2,B3|R3]).
t3(Sol) :- tester3(Sol), square(3,1,9,Sol).

```

Note that the first solution for `t3` is found in less than 0.005 seconds, while `square(3,1,9,Sol)`, `tester3(Sol)` takes 37.450 seconds to find the first solution. (At least four orders of magnitude difference.)

Exercise 10.5.1 Compare the performance with a passive constraints approach, or even a naive generate and test approach.

Exercise 10.5.2 Write the tester so that it is generic in the number of rows and columns and in the magic number itself. Try using your program to actually infer the magic number.

10.6 Moving to full CLP(FD)

Our `plus3` predicate still has some limitations. Indeed, it can only deal with fully known values. We cannot give it information about possible ranges of values of its arguments. For example, above we knew that the numbers in the 3 by 3 magic square were between 1 and 9.

Take another extreme example. Suppose we know that the sum of two numbers is 1,000,000: `plus3(X,Y,1000000)`. Suppose we also know that the first number is between 0 and 1000000: `enumerate(X,1000000)` and the second number between -999995 and 5: `enumerate(Y,-999995,5)`. The call `plus3(X,Y,1000000), enumerate(X,0,1000000), enumerate(Y,-999995,5)` will not be very efficient, as it will try out 1,000,000 solutions for `X`, starting from 0. Luckily, our active constraints will immediately instantiate `Y` for us, and we will not try out 1,000,000 solutions for `Y` for each solution of `X`.

```

| ?- plus3(X,Y,1000000), enumerate(X,0,1000000), enumerate(Y,-999995,5).
X = 999995,
Y = 5 ? ;
X = 999996,
Y = 4 ? ;
X = 999997,
Y = 3 ? ;
X = 999998,
Y = 2 ? ;
X = 999999,

```

```

Y = 1 ? ;
X = 1000000,
Y = 0 ? ;
no

```

We can solve this problem by going from the logic programming paradigm to constraint logic programming (CLP). In a sense, full constraint logic programming provides passive and active constraints. In addition to our `plus3` predicate, it may also store additional partial information about variables, and propagate this information.

There are various CLP implementations and domains. SICStus Prolog supports

- CLP(R) : reals and CLP(Q):rationals
- CLP(B): booleans
- CLP(FD): finite domains

There exists also constraint solver for other programming languages, e.g., the IBM ILOG package.¹ However, the most natural integration of Constraint Programming (CP) is within logic programming. Indeed, classical logic programming can already be seen as a constraint solver (over rational trees).

Typically, the finite domain constraint solver is most useful for industrial applications. The main idea is that every CLP(FD) variable is an integer in some finite domain.² For example, to declare that `X` is a variable in the range 1 to 9, one can write: `X in 1..9`. CLP(FD) stores for every variable (it knows about), an interval of possible values. The interval gets smaller as computation progresses; if the interval becomes empty then backtracking is initiated. CLP(FD) provides equality, disequality and inequality constraints. In general, the Prolog counterpart is preceded by a hash (`#`). For example, to say that `X` plus `Y` is 10, we can write: `X+Y #= 10`. To say that `X` is greater than `Y`, we write `X #> Y`. These constraints are used to propagate the interval information from one variable to another, or to detect inconsistency.

Let us first recode the simple example above in CLP(FD). To use it, you need to execute `use_module(library(clpfd))`.

```

| ?- use_module(library(clpfd)).
...
| ?- X in 0..1000000, Y in -999995..5, X+Y #= 1000000.
X in 999995..1000000,
Y in 0..5 ? ;
no

```

You can see that CLP(FD) has not yet provided us with a solution. It only tells us that there maybe solutions, provided `X` is between 999995..1000000 and

¹See http://www-01.ibm.com/software/integration/optimization/cp1/about/?S_CMP=wspace

²There is some support for unbounded integers.

Y in 0..5.³ To be sure that there is a solution and to find one, we need to call the `labeling` predicate. The second argument to `labeling` is a list of variables which will be enumerated. The first argument is a list of options, which we will ignore for the moment.

```
| ?- X in 0..1000000, Y in -999995..5, X+Y #= 1000000, labeling([], [X,Y]).
X = 999995,
Y = 5 ? ;
X = 999996,
Y = 4 ? ;
X = 999997,
Y = 3 ? ;
X = 999998,
Y = 2 ? ;
X = 999999,
Y = 1 ? ;
X = 1000000,
Y = 0 ? ;
no
```

We will now recode the magic square in CLP(FD). There already exists an `all_different/1` predicate. We also have a list to sum up a list of values:

```
| ?- sum([X,Y], #=, 10), X in 0..5.
X in 0..5,
Y in 5..10 ?
yes
```

Here is a generic solution; we also use two predicates from the lists library:

```
:- use_module(library(lists), [transpose/2, append/2]).

tester(N, Square, MagicNr) :- % N=dimension of square
    sum_all_rows(Square, MagicNr, N, N),
    append(Square, AllNumbers),
    all_different(AllNumbers),
    transpose(Square, TS),
    sum_all_rows(TS, MagicNr, N, N),
    get_diagonal(Square, 1, 1, Diagonal1), sum_list(Diagonal1, 0, MagicNr),
    get_diagonal(Square, N, -1, Diagonal2), sum_list(Diagonal2, 0, MagicNr).

get_diagonal([], _, _, []).
get_diagonal([Row|T], N, Inc, [D|DT]) :- member_nr(D, Row, N), N1 is N + Inc,
    get_diagonal(T, N1, Inc, DT).
```

³The Prolog system only shows us the intervals for nonground variables; it does not show additional constraints/coroutines attached to them.

```

member_nr(X,[X|_],1) :- !.
member_nr(X,[_|T],N) :- TN is N-1, member_nr(X,T,TN).

sum_all_rows([],_,N,_) :- N<1.
sum_all_rows([Row|T],MN,N,GlobalN) :- N>0, N1 is N-1,
    length(Row,GlobalN), % will instantiate Row if GlobalN known
    clpfd_sum_list(Row,MN), sum_all_rows(T,MN,N1,GlobalN).

clpfd_sum_list(List,Sum) :- sum(List,'#=',Sum).

| ?- tester(4,Sol,MN), square(4,1,16,Sol).
MN = 34,
Sol = [[1,2,15,16],[12,14,3,5],[13,7,10,4],[8,11,6,9]] ?
yes

```

Exercise 10.6.1 Use CLP(FD) to perform an interval analysis for variables in Java Bytecode or Three-Adress Code by instrumenting your earlier interpreters or analysers.

10.7 CLP view of LP

```

| ?- X=Y,Y=Z,Z=1.
X = 1,
Y = 1,
Z = 1 ?
yes

| ?- X#=Y,Y#=Z,Z#=1.
X = 1,
Y = 1,
Z = 1 ?
yes

```

10.8 Other Applications

SAT-Solving (unit propagation)