All Candidates LP

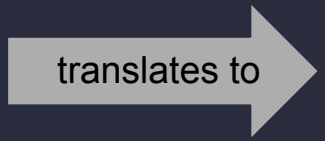
Overall strategy

-Generate all possible terms (eliminate those that don't respect constraints)

-Generate all possible functions from those terms (eliminate those that don't respect constraints)

Example for function of v3 ("lp_models/corrupted/3/3-corrupted-f.lp"):

```
%Regulatory function of v3
function(v3, 3).
term(v3, 1, v4).
term(v3, 2, v2).
term(v3, 3, v1).
```



Fv3 = v4 or v2 or v1

Variables of v3: v4, v2 and v1
Possible terms (notation : [‘v4’,’v1’] means v4 AND v1):

Level 1 (one variable max per term)
1:['v1'] | 2:['v2'] | 3:['v4']

Level 2
1:['v4', 'v1'] | 2:['v4', 'v2'] | 3:['v2', 'v1']

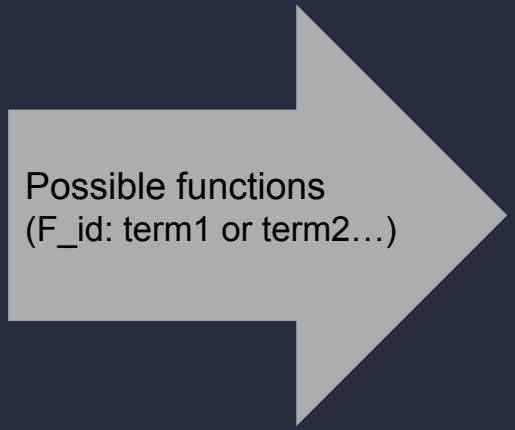
Level 3 (all vars in the same term)
1:['v4', 'v2', 'v1']

- Possible terms:

Level 1
1:['v1'] | 2:['v2'] | 3:['v4']

Level 2
1:['v4', 'v1'] | 2:['v4', 'v2'] | 3:['v2', 'v1']

Level 3
1:['v4', 'v2', 'v1']



Clause number: 1
4: ['v1']
16: ['v2']
11: ['v4']
17: ['v4', 'v1']
10: ['v4', 'v2']
6: ['v2', 'v1']
9: ['v4', 'v2', 'v1']
Clause number: 2
2: ['v1'] or ['v2']
7: ['v1'] or ['v4']
15: ['v1'] or ['v4', 'v2']
8: ['v2'] or ['v4', 'v1']
12: ['v2'] or ['v4']
5: ['v4'] or ['v2', 'v1']
1: ['v4', 'v1'] or ['v2', 'v1']
13: ['v4', 'v1'] or ['v4', 'v2']
3: ['v4', 'v2'] or ['v2', 'v1']
Clause number: 3
14: ['v1'] or ['v2'] or ['v4']
18: ['v4', 'v1'] or ['v4', 'v2'] or ['v2', 'v1']

Implementation

> get a function's variables and generate levels

```
%Variables of inconsistent function F
variable(F,N) :- term(F,_,N), inconsistent(E,F,_).

%Total number of variables of function F
total_variables(F,T) :- T = #count{N : variable(F,N)}, inconsistent(E,F,_).


%Levels represent the number of terms held by each clause, e.g. in level 1 each
clause will only have 1 term,
%in level L there will be a single clause for all L terms
level(F,1..T) :- total_variables(F,T).
```

Implementation

> determine exact number of clauses per level (given by $_vC_L$)

e.g. if we have 3 variables (a,b,c) , then level 2 will have $_3C_2$ = 3 possible clauses: (a,b), (a,c) and (b,c)

(from here on out the name term is used to refer to a single variable inside of a clause)

```
%Calculate maximum factorial required for our calculations
get_factorial(T) :- total_variables(F,T).

%Calculate how many clauses exactly each level should have
level_clause_total(F,L,FACT1/FACT2/FACT3) :- level(F,L), total_variables(F,T),
    factorial(T, FACT1), factorial(T-L, FACT2), factorial(L, FACT3).
```

Implementation

> generate exact number of terms for every clause of every level

```
%Generate terms for every clause of every level (all possible function terms are generated
here)
%e.g for {a,b,c}:
%level 1 (3 clauses, 1 term each):   a; b; c
%level 2 (3 clauses, 2 terms each):  a & b; a & c ; b & c
%level 3 (1 clause, 3 terms in it): a & b & c
%F - function that could be defined by these variable combinations (the inconsistent
function)
%L - the level of the generated term
%N - the variable being placed in clause C of level L
%C - the clause of level L that is being considered
{generated_term(F,L,N,1..C) : variable(F,N)} = C*L :-  level(F,L), level_clause_total(F,L,C).
```

Implementation

> get occupied clauses and total number of terms in each clause (used in integrity constraints displayed next)

```
%Clauses occupied by generated terms
clause(F,L,C) :- generated_term(F,L,N,C).

%Total number T of terms on clause C of level L .
terms_per_clause(F,L,C,T) :- T = #count{N : generated_term(F,L,N,C)},
function(F,_), clause(F,L,C).
```

Implementation

> integrity constraints for generated clauses

```
%Fill in clauses in order
:- clause(F,L,C+1), not clause(F,L,C), C > 0.

%Number of terms per clause must to be equal to the level
:- terms_per_clause(_,L,_,T), T != L.

%Every clause must be unique
:- clause(F,L,C1), clause(F,L,C2), C1 != C2,
    REPETITIONS = #count{N : generated_term(F,L,N,C1), generated_term(F,L,N,C2)}, REPETITIONS = L.
```

Implementation

> generated clauses will be assigned to candidate functions
(ideally the number of max candidates is not fixed and we would use as many IDs as required, still needs some work)

```
#const max_candidates = 18.
%IDs of each candidate
id(F,1..max_candidates) :- inconsistent(_,F,_).
```

Implementation

> predicates used in integrity constraints and to ensure we are covering all possible function candidates, respectively

```
%Clause C1 of level L1 is not contained in clause C2 of level L2 (necessary because
%we're only interested in generating function candidates in the BCF form, which is the disjunction
%of prime implicants. If a clause C1 is contained inside of another clause C2, then C2 is not a prime
implicant,
%and therefore cannot be in the same function as C1.)
not_contained_in(F,L1,C1,L2,C2) :- clause(F,L1,C1), clause(F,L2,C2), generated_term(F,L1,N,C1), not
generated_term(F,L2,N,C2), L2 > L1.

%How many clauses there will be in a given candidate (used to ensure we are generating all candidates with
%1,2,...,MAX clauses possible)
function_clauses(F,1..MAX_CLAUSES) :- total_variables(F,T), factorial(T,FACT_T), factorial(T/2,
FACT_TDIV1), factorial(T-T/2, FACT_TDIV2), MAX_CLAUSES = FACT_T/FACT_TDIV1/FACT_TDIV2.
```

Implementation

> generation of candidate functions

```
%function_candidate(F,ID,CLAUSE_NUMBER,L,C)
%F - function that needs to be changed (inconsistent function)
%ID - unique ID of this function candidate
%CLAUSE_NUMBER - number of total clauses in this candidate
%L - level of the generated clause that is a part of this candidate
%C - number of the generated clause that is a part of this candidate
{function_candidate(F,ID,CLAUSE_NUMBER,L,C) : function_clauses(F, CLAUSE_NUMBER),
clause(F,L,C), id(F,ID)}.
```

Implementation

> predicate used to ensure we are generating unique candidates (respective integrity constraint displayed next)

```
%If two distinct function candidates with the same number of clauses have at least one clause
that is present in
%one candidate but not in the other, then they are distinct
distinct_candidates(F,ID1,ID2) :- function_candidate(F,ID1,CLAUSE_NUMBER,L,C),
function_candidate(F,ID2,CLAUSE_NUMBER,_,_),
not function_candidate(F,ID2,CLAUSE_NUMBER,L,C), ID1 < ID2.
```

Implementation

> integrity constraints for function candidates

```
%Functions must contain as many clauses as their CLAUSE_NUMBER
:- function_candidate(F,ID,CLAUSE_NUMBER,_,_), #count{1,L,C : function_candidate(F,ID,CLAUSE_NUMBER,L,C)}
!= CLAUSE_NUMBER.

%Clauses in a function cannot contain any other clause in that function (BCF)
:- function_candidate(F,ID,CLAUSE_NUMBER,L1,C1), function_candidate(F,ID,CLAUSE_NUMBER,L2,C2), not
not_contained_in(F,L1,C1,L2,C2), L2 > L1.

%Function candidates must be distinct
:- function_candidate(F,ID1,CLAUSE_NUMBER,_,_), function_candidate(F,ID2,CLAUSE_NUMBER,_,_), ID1 < ID2,
    not distinct_candidates(F,ID1,ID2).

%IDs must be unique
:- function_candidate(F,ID,CLAUSE_NUMBER1,_,_), function_candidate(F,ID,CLAUSE_NUMBER2,_,_), CLAUSE_NUMBER1
!= CLAUSE_NUMBER2.
```

Implementation

> optimize for the maximum number of candidates possible

```
%Optimize
#maximize{1,F,ID :
function_candidate(F,ID,CLAUSE_NUMBER,L,C)}.
```