Państwowa Wyższa Szkoła Zawodowa w Tarnowie
Wydział Politechniczny
Języki i systemy sztucznej inteligencji

Prezentacja biblioteki języka prolog CLPFD

Autorzy:
Patryk Zych
Dawid Pleśniarski
Michał Boruch

Praca wykonana pod przewodnictwem:

prof. dr hab. inż Antoni Ligęza

2 grudnia 2019

Spis treści

1 Wprowadzenie

Czym jest Constraint Logic Programming?

Constraint Logic Programming został wprowadzony w roku 1987 przez J.Jaffara oraz J.L Lassez'a. Stosując logiczne podejście programistyczne, definiujemy klasę języków programowania, języków CLP, z których wszystkie mają te same podstawowe właściwości semantyczne. Z koncepcyjnego punktu widzenia programy CLP są wysoce deklaratywne i mają solidne podstawy w jednolitych ramach formalnej semantyki. Ramy te nie tylko pokrywają się z programowaniem logicznym, ale bardziej naturalnie spełniają podstawowe właściwości programów logicznych. Z punktu widzenia użytkownika programy CLP mają wielką moc ekspresyjną ze względu na ograniczenia, którymi naturalnie manipulują. Wreszcie, z punktu widzenia implementatora, systemy CLP mogą być wydajne ze względu na wykorzystanie technik rozwiązywania ograniczeń w określonych domenach.

Clp(fd) to biblioteka zawarta w standardowej dystrybucji SWI-Prolog. Rozwiązuje problemy dotyczące zestawów zmiennych, w których relacje między zmiennymi wymagają spełnienia. Clp (fd) jest także przydatny do zastępowania wielu funkcji wartowniczych udostępnianych przez gettery i settery w językach zorientowanych obiektowo. Na przykład, jeśli nie znamy rzeczywistej wysokości osoby, możemy nadal twierdzić, że ma ona więcej niż 21 i mniej niż 108 cali (najmniejszy i najwyższy znany na świecie człowiek). Kiedy w końcu znajdziemy rozwiązanie, nieuzasadnione wartości zostaną odrzucone. Clp (fd) jest przydatny do optymalizacji problemów z wyszukiwaniem poprzez zmniejszenie przestrzeni wyszukiwania. Ograniczenia mogą przycinać całe poddrzewa przestrzeni wyszukiwania.

Programowanie z ograniczeniami pozwala użytkownikowi określić warunki, które musi spełnić rozwiązanie. Na podstawie danych ograniczeń, rozwiązanie jest wyszukiwane. W ostatnich latach programowanie z ograniczeniami zyskuje coraz większą popularność. Jest to spowodowane tym, że tego typu programowanie pozwala na modelowanie i rozwiązywanie problemów z wielu dziedzin, np:

- sztucznej inteligencji
- problemów kombinatorycznych
- problemów harmonogramowania- np: kiedy i co powinniśmy wykonać w fabryce aby wytworzyć produkt?
- problemów optymalizacji- np: jaki zestaw produktów powinniśmy wyprodukować, aby zmaksymalizować zyski?
- problemów kryteriów- np: znalezienie układu pokoi w szpitalu, który spełnia kryteria, takie jak umieszczenie sali operacyjnej w pobliżu pokoi pooperacyjnych lub znalezienie zestawu planów urlopowych, na które cała rodzina może się zgodzić.
- problemów sekwencji- np: znalezienie trasy, która poprowadzi nas do celu.
- ograniczeń geometrycznych grafiki- np: system CAD, w którym dwie linie muszą być prostopadłe, kolejne dwie równoległe, inne w odległości 30 cali i tak dalej.

- przetwarzania języków naturalnych
- systemów baz danych
- inżynierii elektronicznej
- znalezienie akceptowalnych rozwiązań dla połączonych zestawów zmiennych np: ustalenie, kto pójdzie na imprezę wśród grupy znajomych, gdy A pójdzie tylko, jeśli B pójdzie, ale B nie pójdzie, jeśli C pójdzie, i tak dalej.

Główną zaletą programowania z ograniczeniami jest jego deklaratywność. Oznacza to, że samo sformułowanie zadania jest jednocześnie programem rozwiązującym to zadanie. Programowanie to bazuje na modelowaniu zadania jako problemu spełnienia ograniczeń. Pierwszą dziedziną (domeną) zmiennych był skończonym zbiorem liczb naturalnych. Innymi dziedzinami są skończone zbiory, drzewa, rekordy, przedziały rzeczywiste. Najistotniejszą zaletą programowania z ograniczeniami jest ich propagacja, czyli filtrowanie niespójnych elementów ze zbioru rozwiązań.

2 Wprowadzenia do biblioteki CLP(FD)

Biblioteka Constraint Logic Programming over Finite Domains jest instancją głównego schematu Constraint Logic Programming, która rozszerza programowanie logiczne z ograniczeniami. Dodatkowo biblioteka clpfd pozwala nam deklarować liczby całkowite zgodnie z naturą języka Prolog.

Występują dwa główne przypadki użycia biblioteki CLP(fd)

- deklaratywna arytmetyka liczb całkowitych
- rozwiązywanie problemów z kombinatoryki takich jak planowanie czy chociażby zadania alokacyjne

Bibliotekę można podzielić na:

- ograniczenia arytmetyczne
- ograniczenia relacyjne
- predykaty enumeryczne
- ograniczenia kombinatoryczne

Najbardziej podstawowymi oraz najczęściej używanymi elementami biblioteki clpfd jest programowanie z ograniczeniami arytmetycznymi.

Głównym sposobem zaimportowania biblioteki clpfd jest po prostu dodanie takiej linii do naszego pliku programu.

```
:- use_module(library(clpfd)).
```

Pozwoli nam to na korzystanie z całej biblioteki CLP (FD)

3 Ograniczenia

Ograniczenia arytmetyczne są najczęściej wykorzystywanymi elementami z biblioteki clpfd. W Prologu ograniczenia arytmetyczne często zastępują predykaty niskiego poziomu. Główną zaletą ograniczeń arytmetycznych jest to, że są one prawdziwymi relacjami

Zasadniczo ograniczenia są to relacje między zmiennymi lub wyrażeniami. Poniżej rozważamy ograniczenia CLP (FD), które umożliwiają rozumowanie liczb całkowitych w deklaratywny sposób.

Najważniejszymi ograniczeniami arytmetycznymi CLP (FD) są:

- (#=)/2
- (#<)/2
- (#>)/2
- (#\=)/2

Jednak wszystkie główne ograniczenia arytmetyczne są przedstawione poniżej:

Expr1 #= Expr2	Expr1 jest równe Expr2
Expr1 #Expr2	Expr1 nie jest równe Expr2
Expr1 #>= Expr2	Expr1 jest większe lub równe Expr2
Expr1 #=< Expr2	Expr1 jest mniejsze lub równe Expr2
Expr1 #> Expr2	Expr1 jest większe od Expr2
Expr1 #< Expr2	Expr1 jest mniejsze od Expr2

Gdzie Expr może być

integer	Podana wartość
variable	Nieznana wartość
?(variable)	Nieznana wartość
Expr1 + Expr2	Dodawanie
Expr1 * Expr2	Mnożene
Expr1 - Expr2	Odejmowanie
Expr1 Êxpr2	Potęgowanie
min(Expr1, Expr2)	Minimum z dwóch wyrażeń
max(Expr1, Expr2)	Maksimum z dwóch wyrażeń
Expr1 mod Expr2	Reszta z dzielenia
Expr1 rem Expr2	Reszta z dzielenia z określonego przedziału
abs(Expr)	moduł
Expr1 div Expr2	dzielenie całkowite

Dodatkowo wspierane są także operacje bitowe takie jak:

- (\)/1
- (/\)/2
- (\/)/2
- (»)/2

- («)/2
- lsb/1
- msb/1
- popcount/1
- (xor)/2

Dlaczego warto korzystać z ograniczeń arytmetycznych?
Dużą korzyścią korzystania z ograniczeń arytmetycznych zamiast prymitywnych niższego poziomu (is)/2, (=:=), jest to, że predykaty niższego poziomu są zmodyfikowane. Oznacza to, że można ich używać tylko w kilku przykładach, a nie nadają się do bardziej ogólnych relacji. Dodatkowo te predykaty mieszają rozumowanie o liczbach całkowitych z liczbami zmiennoprzecinkowymi, a nawet z liczbami wymiernymi. Gdy czytelnik kodu zobaczy taki predykat, może się zastanawiać, który typ zmiennych wykorzystujemy. Gdy używamy ograniczeń arytmetycznych (#=)/2 oraz inne ograniczenia CLP(FD) jasno stwierdzamy, że mamy do czynienia z liczbami całkowitymi.

Praktycznie wszystkie programy Prolog rozumują liczby całkowite. Dlatego zaleca się umieszczenie biblioteki CLP (FD) w naszych programach.

4 Przykłady podstawowych ograniczeń

Ograniczenia arytmetyczne

Ograniczenia arytmetyczne są najbardziej podstawowym zastosowaniem CLP (FD). Wykorzystywanie podanych ograniczeń może zwiększyć czytelność programów.

Najbardziej podstawowym zastosowaniem ograniczeń CLP (FD) jest obliczanie wyrażeń arytmetycznych obejmujących liczby całkowite. Na przykład:

?- X #= 1+2.
X = 3.

Osiągnąć to można również za pomocą predykatu niższego poziomu (is) / 2. Jednak ważną zaletą ograniczeń arytmetycznych jest ich czysto relacyjny charakter: Ograniczenia mogą być stosowane w dowolnej formie, nawet jeśli nasza szukana liczba znajduje się po drugiej stronie znaku równości:

?- 3 #= Y+2.
Y = 1.

Ta relacyjność, że ograniczenia CLP (FD) są łatwe do wyjaśnienia i użycia, i dobrze nadają się zarówno dla początkujących, jak i doświadczonych programistów Prolog. Używając biblioteki clp(fd), jeśli chcielibyśmy wykorzystać predykaty niższego poziomu (is, =:=) otrzymamy taki oto rezultat:

?- 3 is Y+2.
ERROR: is/2: Arguments are not sufficiently instantiated

?- 3 =:= Y+2.
ERROR: =:=/2: Arguments are not sufficiently instantiated

W przypadku obsługiwanych wyrażeń ograniczenia CLP (FD) są zamiennikami predykatów arytmetycznych niskiego poziomu.
Wszystkie ograniczenia arytmetyczne:

- **?X #= ?Y** X jest równe Y. Jest to najważniejsze ograniczenie arytmetyczne.
- **?X #/= ?Y** X jest różne od Y.
- **?X #>= ?Y** X jest większe lub równe Y.
- **?X #=< ?Y** X jest mniejsze lub równe Y.
- **?X #> >Y** X jest większe od Y.
- **?X #< ?Y** X jest mniejsze od Y.

Biblioteka clp(fd) używa (goal_expansion/2) do automatycznego przerobienia ograniczeń w czasie kompilacji. Pozwala to na używanie predykatów arytmetycznych niskiego poziomu automatycznie, gdy tylko jest to możliwe. Na przykład predykat:
positive_integer(N) :- N #>= 1.
Jest wykonywany tak, jakby był zapisany w następujący sposób

```
positive_integer(N) :-
( integer(N)
-> N >= 1
; N #>= 1
).
```

Positive integer jest zapisany w dwóch formach, klasycznie za pomocą ograniczenia >= oraz z wykorzystaniem biblioteki CLP(FD) jako #>=
To pokazuje, dlaczego wydajność ograniczeń CLP (FD) jest prawie zawsze całkowicie wystarczająca, gdy są one używane w trybach mogących być obsługiwanymi przez arytmetykę niskiego poziomu. Aby wyłączyć automatyczne przepisywanie, ustaw flagę clpfd_goal_expansion jako false.
Zilustrujemy teraz korzyści wynikające z użycia #=/ 2 dla większej ogólności za pomocą prostego przykładu.
Rozważamy definicję n_factorial / 2, odnoszącą każdą liczbę naturalną N do jej silni F:

```
n_factorial(0, 1).
n_factorial(N, F) :-
    N #> 0,
    N1 #= N - 1,
    n_factorial(N1, F1),
    F #= N * F1.
```

Z wykorzystaniem ograniczeń CLP(FD) sprawdzamy kolejno:

- czy nasza liczba jest większa od 0
- do N1 przypisujemy nową wartość (N-1)
- dokonujemy rekurencyjnego wywołania funkcji F przypisujemy wynik wykonanej rekurencji

Ten program wykorzystuje ograniczenia CLP (FD) zamiast arytmetyki niskiego poziomu, a wszystko, co działałoby z arytmetyką niskiego poziomu, działa również z ograniczeniami CLP (FD), zachowując w przybliżeniu tę samą wydajność. Na przykład:

?- n_factorial(47, F).

F = 258623241511168180642964355153611979969197632389120000000000;

false.

Ze względu na większą elastyczność i ogólność ograniczeń CLP (FD), możemy dowolnie zmieniać kolejność celów w następujący sposób:

```
n_factorial(0, 1).
n_factorial(N, F) :-
    N #> 0,
    N1 #= N - 1,
    F #= N * F1.
    n_factorial(N1, F1),
```

W tym konkretnym przypadku poprawione są właściwości zakończenia predykatu. Na przykład następujące zapytania kończą się teraz:

?-n_factorial(N, 1).

N = 0 ;

N = 1 ;

false

?-n_factorial(N, 3).

false

Aby predykat zakończył działanie, jeśli wystąpi instancja dowolnego argumentu, dodaj (domniemane) ograniczenie F # = 0 przed wywołaniem rekurencyjnym. W przeciwnym razie zapytanie n_factorial (N, 0) jest jedynym niekończącym się przypadkiem tego rodzaju.

Podstawową zaletą ograniczeń CLP (FD) jest to, że pozwalają wypróbować różne polecenia oraz zastosować deklaratywne techniki debugowania.

Zmiana kolejności celów (i klauzul) może znacząco wpłynąć na wydajność programów Prologu i możesz wypróbować różne warianty, jeśli zastosujesz podejście deklaratywne. Co więcej, ponieważ wszystkie ograniczenia CLP(FD) zawsze wygasają, umiejscowienie ich na początku może poprawić ale nigdy nie pogorszy jakości, oraz nie wpłynie na sposób zakończenia twojego kodu.

Wyobraźmy sobie jednak, że biblioteka CLP (FD) nie jest w tym momencie dostępna. Jak moglibyśmy sformułować n_factorial wykorzystując do tego prymitywne ograniczenia?

Na samym początku po prostu spróbujmy podmienić predykaty CLP (FD) na predykaty niskiego poziomu:

```
n_factorial(0, 1).
n_factorial(N, F) :-
        N > 0,
        N1 is N - 1,
        F is N * F1.
        n_factorial(N1, F1),
```

Jak się okazuje, program nie zadziała, ponieważ predykaty arytmetyczne niższego poziomu są modyfikowane. Oznacza to, że argumenty muszą być zainicjowane w momencie ich wywołania. W rzeczywistości SWI-Prolog nawet nie skompiluje kodu. Dlatego aby naprawić nasz program, musimy zmienić kolejność naszych ograniczeń, np w następujący sposób:

```
n_factorial(0, 1).
n_factorial(N, F) :-
        N > 0,
        N1 is N - 1,
        n_factorial(N1, F1),
        F is N * F1.
```

Po wykonaniu wywołania ukazuje nam się taki oto rezultat:

```
?- n_factorial(6, F).
F = 720 ;
false.
```

I faktycznie wyniki są zgodne z oczekiwaniami, ale pojawia się problem, gdy będziemy chcieli wykonać bardziej ogólne zapytanie, przykładowo:

```
?- n_factorial(N, F).
N = 0,
F = 1 ;
ERROR: n_factorial/2: Arguments are not sufficiently instantiated
```

Ta wersja programu nie nadaje się bezpośrednio do wyliczania więcej niż jednego rozwiązania, co stanowi sporą przewagę CLP (FD) nad predykatami niskiego poziomu.

Przy wykonaniu tego samego zapytania, jednak z wykorzystaniem wersji programu CLP(FD) otrzymamy taki rezultat:

```
?- n_factorial(N, F).
N = 0,
F = 1 ;
N = F, F = 1;
N = F, F = 2;
N = 3,
F = 6;
N = 4,
F = 24;
```

To zapytanie nie zwróci nam błędu, a dodatkowo zwróci nam wszystkie wyniki, dlatego zatem zaleca się aby trzymać się ograniczeń CLP (FD) dla arytmetyki liczb

całkowitych. Dodatkowo ograniczenia można umieszczać w dowolnej kolejności, które nie będą miały znaczenia na program. Zmiana kolejności ograniczeń nie może niepoprawnie prowadzić do niepowodzenia, gdy tak naprawdę istnieje rozwiązanie.

5 Ograniczenia kombinatoryczne

Oprócz sumowania i zastępowania predykatów arytmetycznych niskiego poziomu, ograniczenia CLP (FD) są często stosowane do rozwiązywania problemów kombinatorycznych, takich jak zadania planowania i alokacji. Do najczęściej stosowanych ograniczeń kombinatorycznych należą **all_distinct / 1, global_cardinality / 2 cumulative / 2**. Ta biblioteka zawiera również kilka innych ograniczeń, takich jak **disjoint2 / 1 i automaton / 8**, które są przydatne w bardziej wyspecjalizowanych aplikacjach.

all_distinct(+Vars)
Jest prawdą, gdy zmienne różnią się parami. Pozwala wykryć, że na przykład nie wszystkie zmienne mogą przyjmować różne wartości.

all_different(+Vars)
Funkcja bardzo zbliżona do **all_distinct/1**, lecz z nieco słabszą propagacją (filtrowanie niespójnych elementów dziedziny). Szczególnie przydatny w momencie, gdy potrzebujemy przypisać do każdej zmiennej różną wartość.

sum(+Vars, +Rel, ?Expr)
Suma elementów z listy Vars. Rel jest jednym z ograniczeń #=, #\=, #<, #>, #=<, #>=.

6 Domains - dziedziny rozwiązań

Czym są dziedziny rozwiązań? Każda zmienna CLP (FD) ma powiązany zestaw dopuszczalnych rozwiązań, które zmieniają się wraz z ograniczeniami, które na nie nakładamy. Zatem domenę czyli dziedziny rozwiązań zmiennej możemy określić jako **skończone zestawy, przedziały liczb całkowitych, które są dopuszczalnymi rozwiązaniami każdej zmiennej CLP (FD).**
Dziedzinami mogą być:

- Atom, który reprezentuje pusty zbiór
- Termin od-do, czyli zapis (F,T), gdzie F oraz T są granicami dziedziny. Granice mogą być zarówno skończone, jak i nieskończone. Występuje również notacja (N, inf, sup), gdzie inf określa zakres dolny, a sup - zakres górny (infimum, supremum)
- Podział terminu na (N, lewy, prawy), gdzie N jest liczbą całkowitą, a zakres lewy i prawy są odpowiednio mniejsze i większe od N. Można określić liczbę całkowitą N jako "dziurę", która nie jest elementem dziedziny.

Początkowo zbiorem rozwiązań każdej zmiennej CLP (FD) jest zbiór wszystkich liczb całkowitych. Ograniczenia CLP (FD), takie jak # = /**2,** #> / **2 i** #\= / **2**, mogą co najwyżej zmniejszyć, ale nigdy nie rozszerzą dziedziny ich argumentów.

Ograniczenia dziedzin są wykorzystywane podczas modelowania i rozwiązywania zadań kombinatorycznych. Służą do określania dopuszczalnych zbiorów rozwiązań zmiennych.

- **?Var in +Domain** - zmienna jest elementem dziedziny
- **+Vars ins +Domain** - Zmienne na liście Vars są elementami dziedziny
- **indomain(?Var)** - Łączy zmienną ze wszystkimi możliwymi wartościami jego dziedziny podczas cofana. Dziedzina Var musi być skończona.

```
?- X #> 3.
X in 4..sup.

?- X #\= 20.
X in inf..19\/21..sup.

?- 2*X #= 10.
X = 5.

?- X*X #= 144.
X in -12\/12.

?- 4*X + 2*Y #= 24, X + Y #= 9, [X,Y] ins 0..sup.
X = 3,
Y = 6.

?- X #= Y #<==> B, X in 0..3, Y in 4..5.
B = 0,
X in 0..3,
Y in 4..5.
```

Widzimy, że dzięki ograniczeniom, możemy wyznaczyć wartości, które spełniają dane ograniczenie.

7 Propagacja ograniczeń

Propagacją ograniczeń nazywamy metodę filtrowania niespójnych elementów z dziedziny. Mówiąc prościej jest to po prostu sprawdzanie, które elementy nie są zgodne z danym ograniczeniem. Często jednak sama propagacja nie jest wystarczająca do tego, aby zebrać konkretne rozwiązania, dlatego oprócz tego potrzebujemy jeszcze jakiejś formy wyszukiwania elementów, czyli **labelingu**.

Propagacja ograniczeń jest wykonywana automatycznie przez tę bibliotekę. Gdy zbiór rozwiązań zmiennej zawiera tylko jeden element, wówczas zmienna jest automatycznie ujednolicona z tym elementem. Dziedziny są brane pod uwagę przy określaniu dalszych ograniczeń oraz przez predykaty wyliczania, takie jak **labeling / 2**.

8 Labeling

Jest to forma wyszukiwania, która zawsze się kończy. Ta właściwość ma ogromne znaczenie dla analizy zakończeń i pozwala nam na czyste oddzielenie części modelującej od rzeczywistego wyszukiwania. W praktyce liczy się kolejność, w jakiej zmienne są powiązane z konkretnymi wartościami swoich dziedzin. Z tego powodu labeling / 2, które jest uogólnieniem label / 1, pozwala określić różne strategie przy wyliczaniu dopuszczalnych wartości.

label(+Vars)
odpowiednik **labeling([], Vars)**
labeling(+Options, +Vars)
Przypisuje wartość do każdej zmiennej w Vars. Labeling oznacza systematyczne sprawdzanie wartości zmiennych skończonych dziedziny Vars, aż wszystkie zostaną umieszczone. Zbiór rozwiązań każdej zmiennej w Vars musi być skończona. Options to lista opcji, które pozwalają przejąć kontrolę nad procesem wyszukiwania.

9 Przykład praktyczny - Sudoku

Jako kolejny przykład rozważymy Sudoku: Jest to popularna łamigłówka, którą można łatwo rozwiązać za pomocą ograniczeń CLP (FD).

```
sudoku(Rows) :-
        length(Rows, 9), maplist(same_length(Rows), Rows),
        append(Rows, Vs), Vs ins 1..9,
        maplist(all_distinct, Rows),
        transpose(Rows, Columns),
        maplist(all_distinct, Columns),
        Rows = [As,Bs,Cs,Ds,Es,Fs,Gs,Hs,Is],
        blocks(As, Bs, Cs),
        blocks(Ds, Es, Fs),
        blocks(Gs, Hs, Is).

blocks([], [], []).
blocks([N1,N2,N3|Ns1], [N4,N5,N6|Ns2], [N7,N8,N9|Ns3]) :-
        all_distinct([N1,N2,N3,N4,N5,N6,N7,N8,N9]),
        blocks(Ns1, Ns2, Ns3).

problem(1, [[_,_,_,_,_,_,_,_,_],
            [_,_,_,_,_,3,_,8,5],
            [_,_,1,_,2,_,_,_,_],
            [_,_,_,5,_,7,_,_,_],
            [_,_,4,_,_,_,1,_,_],
            [_,9,_,_,_,_,_,_,_],
            [5,_,_,_,_,_,_,7,3],
            [_,_,2,_,1,_,_,_,_],
            [_,_,_,_,4,_,_,_,9]]).
```

Przykładowe zapytanie:

```
?- problem(1, Rows), sudoku(Rows), maplist(portray_clause, Rows).
[9, 8, 7, 6, 5, 4, 3, 2, 1].
[2, 4, 6, 1, 7, 3, 9, 8, 5].
[3, 5, 1, 9, 2, 8, 7, 4, 6].
[1, 2, 8, 5, 3, 7, 6, 9, 4].
[6, 3, 4, 8, 9, 2, 1, 5, 7].
[7, 9, 5, 4, 6, 1, 8, 3, 2].
[5, 1, 9, 2, 8, 6, 4, 7, 3].
[4, 7, 2, 3, 1, 9, 5, 6, 8].
[8, 6, 3, 7, 4, 5, 2, 1, 9].
Rows = [[9, 8, 7, 6, 5, 4, 3, 2|...], ... , [...|...]].
```

W tym konkretnym przypadku constraint solver jest wystarczająco skuteczny, aby znaleźć unikalne rozwiązanie bez żadnego dodatkowego wyszukiwania (labeling/2).

10 Podstawowe szukanie i relacje

Wykorzystanie biblioteki CLP (FD) do rozwiązywania problemów kombinatorycznych zazwyczaj składa się z dwóch faz: **modelowania i szukania**

Na etapie modelowania deklarowane są wszystkie potrzebne ograniczenia, potrzebne do znalezienia rozwiązania.

W etapie szukania wykorzystane predykaty szukają konkretnych rozwiązań.

Dobrą praktyką jest trzymanie fazy modelowania niezależnie od fazy szukania rozwiązań. Dzięki temu będziemy w stanie obserwować właściwości znajdowania, zakończenia i determinizmu relacji. Prościej mówiąc oznacza to, że będziemy w stanie wypróbować różne strategie wyszukiwania rozwiązań.

Jako przykład modelowania możemy podać znany już nam problem
SEND + MORE = MONEY.
Ograniczenia:

- każda litera może przyjąć tylko jedną wartość od 0 do 9
- pierwsze litery wyrazów nie mogą przyjąć wartości 0
- wszystkie litery muszą się różnić.

```
puzzle([S,E,N,D] + [M,O,R,E] = [M,O,N,E,Y]) :-
    Vars = [S,E,N,D,M,O,R,Y],
    Vars ins 0..9,
    all_different(Vars),
        S*1000 + E*100 + N*10 + D +
        M*1000 + O*100 + R*10 + E #=
    M*10000 + O*1000 + N*100 + E*10 + Y,
      M #\= 0, S #\= 0.
```

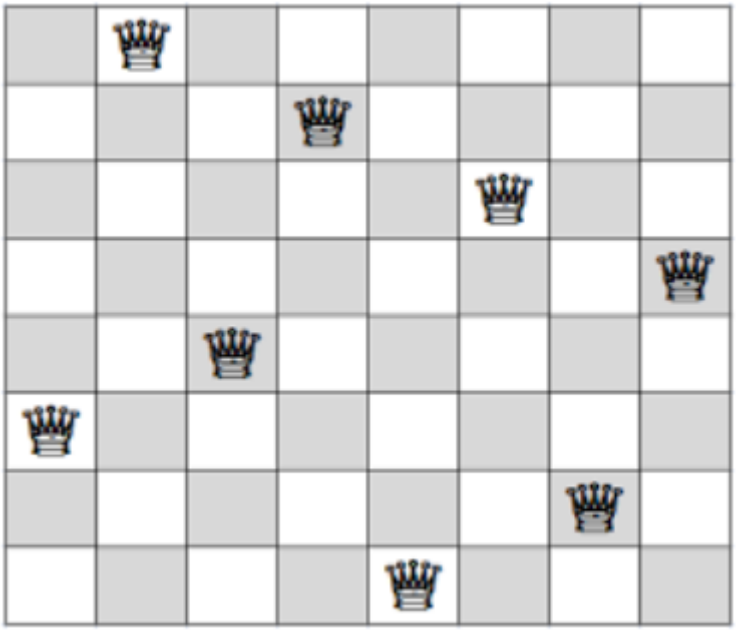
Następnie możemy wykonać zapytanie

```
?- puzzle(As+Bs=Cs), label(As).
As = [9, 5, 6, 7],
Bs = [1, 0, 8, 5],
Cs = [1, 0, 6, 5, 2] ;
```

label jak już wiemy, pozwala na systematyczne sprawdzanie wartości, do momentu, aż wszystkie wartości zostaną sprawdzone.

11 Problem ośmiu królowych

Zadaniem jest umieszczenie królowych w ten sposób, aby żadna z nich nie była atakowana. Oznacza to, że królowe nie mogą dzielić ze sobą jednego rzędu, oraz nie mogą trafić się dwie królowe po przekątnej szachownicy.



Aby wyrazić tę zagadkę za pomocą ograniczeń CLP (FD), musimy najpierw wybrać odpowiednią reprezentację. Ponieważ ograniczenia CLP (FD) wynikają z liczb całkowitych, musimy znaleźć sposób na odwzorowanie pozycji królowych na liczby całkowite i nie jest oczywiste z czego powinniśmy skorzystać. W naszym konkretnym przypadku możemy zauważyć, że dokładnie na jeden rząd musi przypadać jedna królowa. Zatem reprezentacje danego problemu można przedstawić następująco:

Szukamy 8 liczb całkowitych, po jednej na każdą kolumnę, gdzie każda liczba całkowita oznacza rząd królowej umieszczonej w odpowiedniej kolumnie.

W naszym przykładzie wykorzystamy rozwiązanie bardziej dynamiczne, gdzie zmienna N będzie odpowiadała naszej liczbie 8, czyli liczby królowych, które potrzebujemy ustawić.

N - liczba królowych które musimy umieścić. Nasza plansza będzie dokładnie wielkości NxN, czyli w naszym przypadku 8x8 Qs - liczby które są naszymi rozwiązaniami, w naszym przypadku 1...8 (ponieważ nasza liczba reprezentuje rząd w którym królowa się znajduje, potrzebujemy 8 rozwiązań)

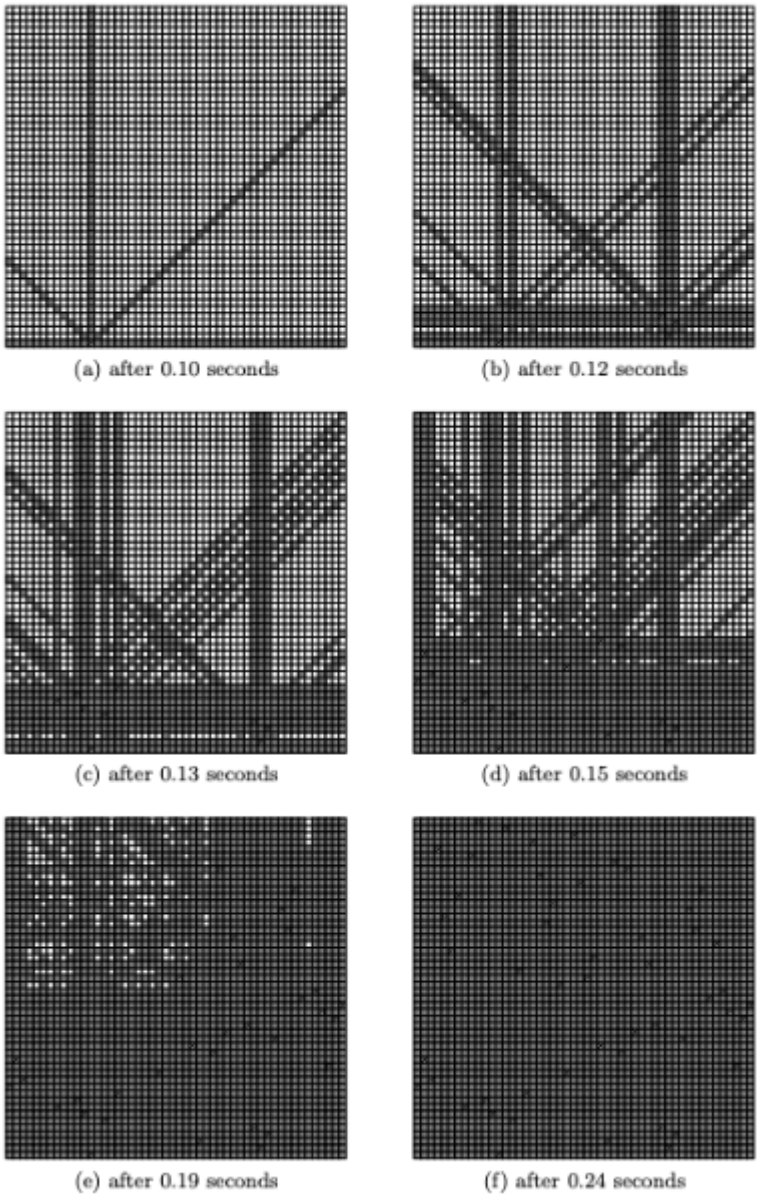
```
n_queens(N, Qs) :-
        length(Qs, N),
        Qs ins 1..N,
        safe_queens(Qs).

safe_queens([]).
safe_queens([Q|Qs]) :- safe_queens(Qs, Q, 1), safe_queens(Qs).

safe_queens([], _, _).
safe_queens([Q|Qs], Q0, D0) :-
        Q0 #\= Q,
        abs(Q0 - Q) #\= D0,
        D1 #= D0 + 1,
            safe_queens(Qs, Q0, D1).
```

Aby rozwiązać nasz problem możemy wykonać:
?- n_queens(8, Qs), label(Qs).
Przez co otrzymujemy: **Qs = [1,5,8,6,3,7,2,4].**
Ustawienie królowych:
1 rząd, 1 kolumna
2 rząd, 5 kolumna
3 rząd, 8 kolumna
4 rząd, 6 kolumna
5 rząd, 3 kolumna
6 rząd, 7 kolumna
7 rząd, 2 kolumna
8 rząd, 4 kolumna

Dany przykład można również rozszerzyć o instrukcje PostScriptu, aby sposób przeszukiwania naszego wyniku był dla nas widoczny.



(a) after 0.10 seconds (b) after 0.12 seconds

(c) after 0.13 seconds (d) after 0.15 seconds

(e) after 0.19 seconds (f) after 0.24 seconds

Jest to przykład rozwiązania dla planszy o powierzchni 50x50

12 Reifikacje

Wszystkie ograniczenia pokazane w poniższej tabeli jak również ograniczenia in/2 mogą zostać zmienione w taki sposób, aby odzwierciedlały wartości jako liczby całkowite 0 i 1, oznaczające odpowiednio fałsz i prawdę. Zabieg ten, nazywamy reifikacją.

Gdy P oraz Q reprezentują ograniczenia, wtedy:

#\Q	Prawda gdy Q jest fałszywe
P #\/ Q	Prawda gdy P lub Q jest prawdziwe
P #/\Q	Prawda gdy P i Q są prawdziwe
P #\Q	Prawda gdy P prawdziwe Q fałszywe lub na odwrót
P #<==> Q	Prawda gdy P i Q są równe
P #==> Q	Prawda gdy P implikuje Q (jeżeli P to Q)

Rozważmy jednak przykład takiej reifikacji:
(X? / 0 #= Y? / 0) #<==> B?
Oczywistym jest fakt, że ta relacja nie może istnieć, ponieważ dzielnik nie może być równy 0. Dlatego ważne jest, aby pamiętać o dodatkowym ograniczeniu, sprawdzającym, czy:
B = 0, X in inf..sup, Y in inf..sup
(B jest zbiorem wszystkich liczb, w pominięciem 0)

Reifikacja zwraca nam tylko wartości true lub false. Pozwala to na wyraźne uzasadnienie, czy dane ograniczenie jest spełnione, czy też nie.

13 Dodawanie monotoniczności

W normalnym trybie wykonywania ograniczenia CLP(FD) czasami wykazują pewne właściwości nierelacyjne. Przykładowo, dodawanie nowych ograniczeń może powodować otrzymanie nowego rozwiązania.
?- X #= 2, X = 1+1.
false.

?- X = 1+1, X #2, X = 1+1.
X = 1+1.

Takie zachowanie może być dosyć uciążliwe z logicznego punktu widzenia i może utrudniać debugowanie takiego kodu.

Jeżeli chcemy, aby biblioteka CLP(FD) była monotoniczna, wystarczy że ustawimy flagę **clpfd_monotonic** na true. Będzie to oznaczało, że nasz kod będzie monotoniczny, a dodawanie nowych ograniczeń nie będzie powodowało nowych rozwiązań. Jedyną zmianą jaką musimy zastosować jest używanie funktora ?/1 lub #/1. Przykładowo:
?- set_prolog_flag(clpfd_monotonic, true).
true.
ustawienie flagi clpfd_monotonic

?- #(X)#=#(Y) + #(Z).
#(Y)+ #(Z)#= #(X).
poprawne zdefiniowanie ograniczenia monotonicznego.

?- X#=2, X = 1+1.
ERROR: Arguments are not sufficiently instantiated
błąd, zła deklaracja argumentów.

Jeżeli liczby są ograniczone do liczb całkowitych, to możemy pominąć notację ?/1 lub #/1

14 Własne, niestandardowe ograniczenia

Używając biblioteki clpfd możemy definiować niestandardowe ograniczenia. Mechanizm definiowania tzw. custom constraints według dokumentacji nie został jeszcze sfinalizowany, więc dany przykład jest umieszczony tylko w celach poglądowych lub informacyjnych.

```
:- multifile clpfd:run_propagator/2.

oneground(X, Y, Z) :-
        clpfd:make_propagator(oneground(X, Y, Z), Prop),
        clpfd:init_propagator(X, Prop),
        clpfd:init_propagator(Y, Prop),
        clpfd:trigger_once(Prop).

clpfd:run_propagator(oneground(X, Y, Z), MState) :-
        (   integer(X) -> clpfd:kill(MState), Z = 1
        ;   integer(Y) -> clpfd:kill(MState), Z = 1
        ;   true
            ).
```

make_propagator jest wykorzystywany do przekształcania zdefiniowanych ograniczeń przez użytkownika. init_propagator inicjuje nam dane ograniczenia i dołącza je do wartości X i Y. W tym momencie za każdym razem, gdy zmieniane są wartości X lub Y, to propagator zostanie wywołany.

Wywołanie przykładu:

```
?- oneground(X, Y, Z), Y = 5.
Y = 5,
Z = 1,
X in inf..sup.
```

15 Predykaty informacyjne

Pozwalają uzyskać w dobrze zdefiniowany sposób informacje, które zwykle są wewnętrzną częścią tej biblioteki. Predykaty te mogą być bardzo przydatne do debugowania naszego kodu.

fd_var(+Var)
Prawda, gdy Var jest zmienną CLP(FD)
fd_inf(+Var, -Inf)
Infinum dziedziny Var
fd_sup(+Var, -Sup)
Supremum dziedziny Var
fd_size(+Var, -Size)
Jest to liczba elementów dziedziny Var
fd_dom(+Var, -Dom)
dom jest to obecna dziedzina zmiennej Var.