

# Programmation Logique et Par Contraintes Avancée

## Cours 7 – Réification et Contraintes Souples

Ralf Treinen

Université Paris Cité  
UFR Informatique  
Institut de Recherche en Informatique Fondamentale



`treinen@irif.fr`

18 février 2025

## Qu'est-ce que c'est la Réification ?

- ▶ Selon Wikipédia :
  - ... *reification is the process by which a user program or any aspect of a programming language that was implicit in the translated program and the run-time system, are expressed in the language itself.*
- ▶ Les *contraintes réifiées* en Oz (et Prolog) : On a une variable à domaine fini  $\{0, 1\}$  liée au fait qu'une contrainte est satisfaite ou pas.

## À quoi sert la Réification ? 1 : Combinaisons de Contraintes

- ▶ Jusqu'à maintenant on a exprimé la *conjonction* des contraintes : toutes les contraintes doivent être satisfaites.
- ▶ La réification permet d'exprimer des combinaisons entre contraintes au delà de la conjonction : négation, équivalence, implication, disjonction.

## À quoi sert la Réification ? 2 : Contraintes Souples

- ▶ La réification permet d'exprimer des contraintes qui sont « souples » (angl. : soft constraints), en opposition aux contraintes dures vues jusqu'à maintenant qui doivent impérativement être satisfaites.
- ▶ Cela permet par exemple d'exprimer qu'on veut satisfaire un nombre maximal de contraintes souples.
- ▶ Les contraintes souples sont nécessaires pour modéliser des problèmes « sur-contraints » (angl. : over-constrained).

## Propagateurs pour contraintes réifiées

- ▶ La *réification* d'une contrainte  $C$  correspond à la formule logique suivante :

$$(C \leftrightarrow x = 1) \quad \text{avec} \quad D(x) = \{0, 1\}$$

où  $x \notin \text{vars}(C)$ .

- ▶ Si la mémoire implique  $x = 1$  alors le propagateur devient un propagateur pour  $C$ .
- ▶ Si la mémoire implique  $x = 0$  alors le propagateur devient un propagateur pour  $\neg C$ .
- ▶ Si la mémoire implique  $C$  alors le propagateur pose  $x = 1$ , puis disparaît.
- ▶ Si la mémoire n'est pas cohérente avec  $C$  alors le propagateur pose  $x = 0$ , puis disparaît.

## Propagateurs pour des contraintes réifiées

- ▶ Voir le document « System Modules », section 5.10 « Reified Constraints »
- ▶ Quelques exemples :
  - ▶  $(X < : Y) = B$
  - ▶  $(X + Y + Z = : 0) = B$
  - ▶  $(X \setminus = : Y) = B$
  - ▶  $(X : : 0 \# 10) = B$
  - ▶ `{FD.reified.distance X Y '=: ' Z B}`

## Exemples (reif1.oz)

```
declare X Y B
[X Y]:::0#10
B:::0#1
(X+Y=:10)=B
{Browse [X Y B]}

X >: 5

B =: 1
% restricts Y to [0..4]
```

## Exemples (reif2.oz)

```
declare X Y B
[X Y]:::0#10
B:::0#1
(X>:Y)=B
{Browse [X Y B]}

X >: 5

B =: 0
% imposes that Y > 5
```



## Exemples (reif3.oz)

```
declare X Y B  
[X Y]:::0#10  
B::0#1  
(X+Y=:10)=B  
{Browse [X Y B]}
```

```
X = 3
```

```
Y = 7
```

```
% imposes B=1
```

## Exemples (reif4.oz)

```
declare X Y B
[X Y]:::0#10
B:::0#1
(X+Y=:10)=B
{Browse [X Y B]}

X >: 3

Y >: 7
% imposes B=0
```

## Abréviations pour les propagateurs réifiés

- ▶ L'*instruction*  $X+Y =: Z$  est un raccourci pour l'instruction `{FD.sum [X Y] '=: ' Z}`
- ▶ L'instruction  $(X+Y '=: ' Z) = D$  est un raccourci pour l'instruction `{FD.reified.sum [X Y] '=: ' Z D}`
- ▶ L'*expression*  $X+Y =: Z$  est un raccourci pour l'expression `{FD.reified.sum [X Y] '=: ' Z}`  
Cette expression a comme valeur la variable booléenne qui est la réification de la contrainte.

## Variables booléennes et Contraintes pseudo-booléennes

- ▶ On appelle souvent une variable avec domaine  $[0,1]$  une *variable 0/1* ou une *variable booléenne*.

$$(C \leftrightarrow x = 1) \quad \text{où } D(x) = \{0, 1\}$$

- ▶ Contrainte *pseudo-booléenne* : les variables sont booléennes, mais on les utilise pour former une expression dont les valeurs peuvent être des entières non booléennes. Exemple :

$$X_1 + X_2 + X_3 + X_4 \leq 2$$

exprime qu'au plus deux des variables sont vraies (supposant que toutes ces variables sont booléennes).

- ▶ Souvent utilisées en connexion avec la réification : contrainte sur le *nombre de contraintes réifiées qui sont satisfaites*.

## Connecteurs booléens

- ▶ Voir le document « System Modules », section 5.9 « 0/1 Propagators »
- ▶ Exemples :
  - ▶ {FD.conj X Y Z} pour  $(X \wedge Y) = Z$
  - ▶ {FD.disj X Y Z} pour  $(X \vee Y) = Z$
  - ▶ {FD.impl X Y Z} pour  $(X \rightarrow Y) = Z$
  - ▶ {FD.equi X Y Z} pour  $(X \leftrightarrow Y) = Z$
  - ▶ {FD.nega X Y} pour  $(\neg X) = Y$

## Exemples (bool1.oz)

```
declare X Y XB YB  
[X Y]:::0#10  
[XB YB]:::0#1  
{Browse [X Y]}  
  
(X=:5)=XB  
(Y=:5)=YB  
{FD.disj XB YB 1}  
  
X=1  
  
% poses Y=5
```

## Réification : profiter de la syntaxe fonctionnelle

- ▶ Puisque  $\{FD.disj\ X\ Y\ Z\}$  est un appel de procédure ...
- ▶  $\{FD.disj\ X\ Y\}$  est un appel de fonction, qui envoie une variable booléenne.
- ▶ Parfois utile car ca évite d'inventer des noms pour des variables booléennes.
- ▶ Pareil :  $(X=:Y)$  est une expression qui donne une variable booléenne qui est liée au fait que X est égal à Y.

## Exemples (bool2.oz)

```
declare X Y
[X Y]:::0#10
{Browse [X Y]}

% syntactic sugar for the previous example
{FD.disj (X=:5) (Y=:5) 1}

X=1

% poses Y=5
```



## Exemples (exactly-by-rectification.oz)

**declare**

*% reimplementation of FD.exactly (see lecture 6)*

**proc** {MyExactly N L I}

*% exactly N elements of L are equal to I}*

{FD.sum {List.map L **fun** {\$ X} X=:I **end**} '=: ' N}

**end**

*% propagation de Dv vers D*

**declare** L N **in**

{FD.list 5 1#10 L}

N::0#20

{Browse L}

{Browse N}

{MyExactly N L 8}

N >=: 4

{Nth L 3} =: 7

## Exemple : la photo de groupe

- ▶ Prendre une photo du groupe Betty, Chris, Donald, Fred, Gary, Mary, and Paul (tous placés un à côté de l'autre).
- ▶ Betty veut être à côté de Gary et Mary.
- ▶ Chris veut être à côté de Betty et Gary.
- ▶ Fred veut être à côté de Mary et Donald.
- ▶ Paul veut être à côté de Fred et Donald.

## Exemples (photo-hard.oz) I

```

declare
proc {Photo Root}
    Persons = [betty chris donald fred gary mary paul]
    Prefs    = [betty#gary betty#mary chris#betty chris#gary
                fred#mary fred#donald paul#fred paul#donald]
    Alignment = {FD.record alignment Persons 1#{Length Persons}}
    proc {Satisfy P#Q}
        {FD.distance Alignment.P Alignment.Q '=' 1}
    end
in
    Root = Alignment
    {FD.distinct Alignment} % no two persons at same position
    {List.forAll Prefs Satisfy}
    Alignment.fred <: Alignment.betty % breaking symmetries
    {FD.distribute ff Alignment}
end
{Browse {SearchOne Photo}}
    
```

## Exemple : la photo de groupe

- ▶ Problème : les préférences sont contradictoires. Ca arrive souvent en pratique !
- ▶ Jusqu'à maintenant les contraintes étaient dures (*hard constraints*) : une contrainte non satisfaite est cause d'échec.
- ▶ Solution : traiter les préférences comme des contraintes souples (*soft constraints*) :
  - ▶ Réifier les contraintes de distances ;
  - ▶ Maximiser le nombre de contraintes souples qui sont satisfaites.

## Exemples (photo-soft.oz) I

```

declare
proc {Photo Root}
    Persons = [betty chris donald fred gary mary paul]
    Prefs= [betty#gary betty#mary chris#betty chris#gary
           fred#mary fred#donald paul#fred paul#donald]
    Alignment = {FD.record alignment Persons 1#{Length Persons}}
    NbSatisfied = {FD.int 0#{Length Prefs}}
    fun {Satisfied P#Q}
        {FD.reified.distance Alignment.P Alignment.Q '=' 1}
        % returns S such that S <=> (|P-Q|=1)
    end
in
    Root = r(satisfaction: NbSatisfied
            alignment: Alignment)
    {FD.distinct Alignment} % no two persons at same position
    {FD.sum {Map Prefs Satisfied} '=' NbSatisfied}
    % Creates one 0/1 variable per preference

```

## Exemples (photo-soft.oz) II

```

% Each such variable is constrained to the fact that
% the corresponding preference is satisfied.
% NbSatisfied is constrained to the number of satisfied
% 0/1 variables.
Alignment.fred <: Alignment.betty    % breaking symmetries
{FD.distribute generic(value:max) [NbSatisfied]}
{FD.distribute naive Alignment}

```

**end**

```

{Browse {SearchOne Photo}}
{ExploreOne Photo}

```