

# Programmation Logique et Par Contraintes Avancée

## Cours 5 – Propagateurs

Ralf Treinen

Université Paris Cité  
UFR Informatique  
Institut de Recherche en Informatique Fondamentale



`treinen@irif.fr`

4 février 2025

## Rappel du cours précédent

- ▶ *domaine*  $D$  : associe à des variables des ensembles finis de valeurs entières (typiquement, mais pas nécessairement, des intervalles).  
Le domaine donne les valeurs possibles des variables.
- ▶ *contrainte* : énoncé mathématique du problème qu'on cherche à résoudre, c'est une relation entre les variables.
- ▶ *propagateur* : réalisation opérationnelle d'une contrainte. C'est une fonction qui envoie un domaine vers un nouveau domaine.

## Imposer des domaines de variables en Oz

- ▶ Imposer le domaine de la variable  $D$  comme étant l'intervalle  $[Lower \dots Upper]$  :

```
D::Lower#Upper
{FD.int Lower#Upper D}
```

- ▶ Imposer le même domaine à toutes les variables d'un vecteur (une liste, par exemple)  $L$  :

```
L:::Lower#Upper
```

## Domaines fini en Oz

- ▶ `FD.sup` est la plus grande borne supérieure possible d'un domaine en Oz.
- ▶ `{FD.decl X}` définit une variable  $X$  avec le domaine fini maximal possible :  $0 \dots \text{FD.sup}$ .

## Interprétation logique d'un domaine

- ▶ On peut voir un domaine, par ex.  $\{4, 5, 6\}$  pour une variable  $X$ , comme une formule logique :

$$X = 4 \vee X = 5 \vee X = 6$$

- ▶ Avec l'avancement du calcul, le domaine d'une variable peut être réduit, par exemple à  $\{5, 6\}$ .
- ▶ On a bien une implication logique entre le nouveau domaine et l'ancien :

$$X = 5 \vee X = 6 \models X = 4 \vee X = 5 \vee X = 6$$

- ▶ On a donc toujours la même propriété comme vue aux cours 2 et 3 : la mémoire accroit logiquement !

## Propagateurs de bornes

- ▶ On parle de l'infimum et du suprémum du domaine d'une variable ( $inf(x), sup(x)$ ). Ces valeurs changent avec l'avancement du programme!
- ▶ Propagateur de bornes : ne fait que croître l'infimum et décroître le suprémum d'un domaine.
- ▶ Un propagateur de bornes transforme toujours un intervalle en un domaine qui est un intervalle, mais peut aussi être appliqué à un domaine qui n'est pas un intervalle.

## Propagateur de bornes pour des équations linéaires

- ▶ Donnée une équation linéaire :

$$\sum_{i=1}^n a_i X_i = b$$

où les  $X_i$  sont des variables à domaine fini, et les  $a_i$  et  $b$  des constantes entières.

- ▶ Notons (sur ces transparents seulement)  $\bar{t}$  pour le suprémum, et  $\underline{t}$  pour l'infimum d'un terme  $t$ .
- ▶ Pour trouver les propagateurs on résoud l'équation dans chacune des variables, et on obtient :

## Propagateurs de bornes

- ▶ Résoudre l'équation dans  $X_i$  :

$$X_i = \frac{b - \sum_{(j \neq i)} a_j X_j}{a_i}$$

- ▶ On note que

$$\sum_{(j \neq i)} a_j X_j \geq \sum_{(j \neq i, a_j > 0)} a_j * \underline{X_j} + \sum_{(j \neq i, a_j < 0)} a_j * \overline{X_j}$$

- ▶ Donc, pour le cas  $a_i > 0$  :

$$\sup(X_i) := \min \left( \sup(X_i), \left[ \frac{b - \sum_{(j \neq i, a_j > 0)} a_j * \underline{X_j} - \sum_{(j \neq i, a_j < 0)} a_j * \overline{X_j}}{a_i} \right] \right)$$

- ▶ On obtient donc  $2n$  threads différents !

## Exemple

```

declare X1 X2 X3
X1::1#100 X2::1#3 X3::1#3
{Browse [X1 X2 X3]}
2*X1 + 2*X2 - 3*X3 =: 20
    
```

- ▶ La nouvelle borne supérieure de  $X1$  est

$$\left\lceil \frac{b - \sum_{(j \neq 1, a_j > 0)} a_j * \underline{X}_j - \sum_{(j \neq 1, a_j < 0)} a_j * \overline{X}_j}{a_1} \right\rceil$$

- ▶ donc sur l'exemple

$$\left\lceil \frac{20 - 2 * 1 - (-3) * 3}{2} \right\rceil = \left\lceil \frac{20 - 2 + 9}{2} \right\rceil = \left\lceil \frac{27}{2} \right\rceil = 13$$

## Les propagateurs arithmétiques

- ▶ Voir le document *System Modules*, chapitre 5 *Finite Domain Constraints*, section 5.7.
- ▶ Procédure de la bibliothèque
 

```
{FD.sumC Iv Dv A D}
```
- ▶ L'écriture sous forme d'une équation linéaire est simplement un raccourci.
- ▶ Oz reconnaît l'équation linéaire et la traduit en un appel de la procédure.

## Exemples (proc.oz)

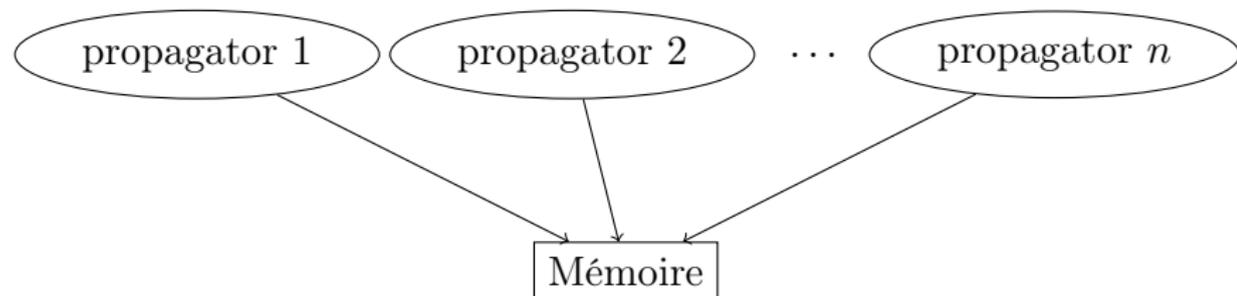
```
declare X Y  
[X Y]:::3#10  
{Browse [X Y]}
```

```
2*X + 3*Y =: 21
```

```
% raccourcie pour  
{FD.sumC [2 3] [X Y] '=: ' 21}
```

```
% cette procédure marche aussi pour des records  
{FD.sumC f(a:2 b:3) f(a:X b:Y) '=: ' 21}
```

## Indépendance des propagateurs



- ▶ Chaque propagateur réagit seulement aux changements dans la mémoire.
- ▶ Il n'y a pas de collaboration entre propagateurs, au delà des informations postées dans la mémoire.

## Exemples (indep1.oz)

```

declare X Y
{Browse X#Y}

{FD.decl X}
{FD.decl Y}

X=<:Y
X>=:Y
% Oz cannot "deduce" that X=Y should hold

X\=:42
% information not propagated to Y
    
```

## Exemples (indep2.oz)

```
declare X Y  
{Browse X#Y}
```

```
{FD.decl X}  
{FD.decl Y}
```

```
X=<:Y
```

```
X>:Y
```

```
% this takes a while ...
```

## Un modèle des propagateurs

- ▶ Donné un ensemble  $F$  de variables de domaine fini.
- ▶ Domaine : fonction  $D: F \rightarrow 2^{\mathbb{N}}$ .
- ▶ Soit  $\mathcal{D}$  l'ensemble de tous les domaines.
- ▶ On a un ordre partiel sur  $\mathcal{D}$  :

$$D_1 \sqsubseteq D_2 \Leftrightarrow \forall x \in F : D_1(x) \subseteq D_2(x)$$

- ▶ Propagateur : Fonction  $p: \mathcal{D} \rightarrow \mathcal{D}$  qui est *monotone* et *décroissante* (voir le transparent suivant).

## Propriétés de propagateurs

Un propagateur doit être

- ▶ *décroissant* : envoie toujours un domaine vers un domaine plus fort (plus restrictif) :  $\forall D \in \mathcal{D} : p(D) \sqsubseteq D$
- ▶ *monotone* :  $\forall D_1, D_2 \in \mathcal{D} : \text{Si } D_1 \sqsubseteq D_2 \text{ alors } p(D_1) \sqsubseteq p(D_2)$

Un propagateur n'est *pas nécessairement*

- ▶ *idempotent* :  $p(p(D)) = p(D)$
- ▶ *complet*

## Correction d'un propagateur par rapport à une contrainte

- ▶ Le propagateur  $p$  est correct par rapport à la contrainte  $c$  si pour tout  $D \in \mathcal{D}$  :
  - ▶ Si  $\alpha \models c$
  - ▶ et  $\forall x \in F : \alpha(x) \in D(x)$
  - ▶ alors  $\forall x \in F : \alpha(x) \in p(D)(x)$
- ▶ Autrement dit :  $p$  ne perd pas de solutions à la contrainte  $c$ .

## Complétude d'un ensemble de propagateurs par rapport à une contrainte

- ▶ L'ensemble  $P$  de propagateurs est complet par rapport à la contrainte  $c$  si pour tout  $D \in \mathcal{D}$  :
  - ▶ Si on a atteint un point fixe :  $p(D) = D$  pour tout  $p \in P$
  - ▶ et  $\alpha$  est une affectation admise par  $D$  :  
 $\forall x \in F : \alpha(x) \in p(D)(x)$
  - ▶ alors  $\alpha \models c$
- ▶ Autrement dit : on élimine toutes les non-solutions seulement par propagation.
- ▶ C'est très rarement le cas qu'un propagateur soit complet.

## Complétude

- ▶ Dans le cas de Oz : la mémoire peut seulement représenter deux types d'informations sur les variables à domaine fini :
  - ▶ domaine d'une variable
  - ▶ égalité entre plusieurs variables
- ▶ Une contrainte peut avoir un propagateur complet (dans le sens du transparent précédent) seulement si l'espace de ses solutions est un produit cartésien.

## Exemples (incomplete.oz)

```
% incompletude des propagateurs de borne
```

```
declare X Y Z
```

```
[X Y Z]:::1#10
```

```
{Browse [X Y Z]}
```

```
X*Y =: Z
```

```
Z=10
```

```
X>:1
```

```
Y>:1
```

## Exemples (ineq.oz)

```
declare X Y
X::0#1
Y::0#1
{Browse [X Y]}

X \=: Y

X >: 0
```

## Un algorithme naïve de propagation de contraintes

- ▶ while  $\exists$  propagateur  $p$  with  $p(D) \neq D$  do  
     $D := p(D)$   
end
- ▶ Il s'agit du calcul d'un *point fixe* commun de tous les propagateurs  $p$ .

## Remarques

- ▶ L'algorithme termine (quand  $F$  est fini), car tous les domaines sont finis et tous les propagateurs sont décroissants.
- ▶ L'algorithme est non-déterministe.
- ▶ Est-ce que le non-déterminisme donne lieu à des résultats non-déterministes ?
- ▶ L'algorithme est naïve car on cherche à chaque itération un propagateur dans l'ensemble de tous les propagateurs disponibles.
- ▶ Dans l'implémentation, chaque propagateur observe les variables qui sont pertinentes pour lui, et devient un candidat à une nouvelle activation seulement quand une de ces variables change son domaine.

## Indépendance du résultat de la stratégie

- ▶ Peut importe la stratégie utilisée dans l'algorithme, à la fin on obtient évidemment un domaine  $D$  qui est
  - ▶ un *point fixe*  $D$  avec  $p(D) = D$  pour tous les propagateurs.
  - ▶ plus petit que le domaine initiale  $D \sqsubseteq D_0$
- ▶ Nous allons montrer :
  - ▶ Si l'algorithme donne  $D_i$  dans la  $i$ -ème itération
  - ▶ et si  $D$  est un point fixe de tous le propagateurs qui est plus petit que  $D_0$
  - ▶ Alors  $D \sqsubseteq D_i$ .

## Démonstration

Par induction sur  $i$ !

- ▶  $i = 0$  : On a  $D \sqsubseteq D_0$  par hypothèse
- ▶  $i \rightarrow i + 1$  : Hypothèse :  $D \sqsubseteq D_i$ . À montrer :  $D \sqsubseteq D_{i+1}$ 
  - ▶  $D = p_i(D)$  car  $D$  est un point fixe.
  - ▶  $p_i(D) \sqsubseteq p_i(D_i)$  car  $p_i$  est monotone et  $D \sqsubseteq D_i$ .
  - ▶  $p_i(D_i) = D_{i+1}$ , où  $p_i$  propagateur de la  $i$ -ème étape.
  - ▶ Donc :  $D \sqsubseteq D_{i+1}$ !

## Conclusion de la preuve

- ▶ Soient  $D_1$  et  $D_2$  obtenues par l'algorithme selon deux stratégies différentes.
- ▶ Donc  $D_1$  et  $D_2$  sont plus petits que  $D_0$ , et des points fixes.
- ▶ Donc  $D_1 \sqsubseteq D_2$  et  $D_2 \sqsubseteq D_1$ .
- ▶ Donc :  $D_1 = D_2$

## Pourquoi cette preuve ?

- ▶ On avait déjà vu que le non-déterminisme en Oz n'est pas observable. N'est ce pas suffisant pour montrer que l'ordre d'exécution des propagateurs n'importe pas ?
- ▶ Non ! La raison est :
- ▶ Les propagateurs eux mêmes ne sont pas programmés dans le langage Oz comme il avait été présenté ici.
- ▶ Pour programmer un propagateur on a besoin de primitives de plus bas niveau qui ne respectent pas la sémantique logique de Oz, par exemple savoir si une variable est liée, ou connaître son domaine.
- ▶ Ces primitives changent de résultat quand la mémoire augmente, leur comportement n'est pas monotone.

## Génération d'un script

- ▶ Le script a un format précis : un argument
- ▶ Or, on souhaite souvent écrire une procédure qui prend des valeurs en paramètre, et construit le script en fonction de ces paramètres.
- ▶ C'est facile à faire grâce aux fonctions (procédures) d'ordre supérieur en Oz : on peut écrire une fonction qui prend des données en paramètre, et retourne le script.

## Exemple : colorer la carte de l'Europe

On suppose donné une carte sous forme d'une liste d'association qui associe à chaque pays la liste de ses voisins, comme

```
declare Europe =  
  [ austria      # [italy switzerland germany]  
    belgium      # [france netherlands germany luxemburg]  
    france       # [spain luxemburg italy]  
    germany      # [austria france luxemburg netherlands]  
    italy        # nil  
    luxemburg    # nil  
    netherlands # nil  
    portugal     # nil  
    spain        # [portugal]  
    switzerland # [italy france germany] ]
```

Colorer une carte avec 4 couleurs (1,2,3,4).

## Exemples (colouring.oz) I

```

declare
fun {MapColoring Data NbColors}
  Countries = {Map Data fun {$ C#_} C end} in % list of countries
  proc {$ Coloration}
    Coloration = {FD.record color Countries 1#NbColors}
    {ForAll Data
      proc {$ A#Bs}
        {ForAll Bs proc {$ B} Coloration.A \=: Coloration.B end}
      end}
    {FD.distribute naive Coloration}
  end
end

{Browse {SearchOne {MapColoring Europe 4}}}

```