Programmation Logique et Par Contraintes Avancée
Cours 6 – Propagateurs

Ralf Treinen

Université Paris Diderot
UFR Informatique
IRIF, équipe PPS

treinen@irif.fr

11 février 2019

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} définie une variable X avec le domaine fini maximal possible : `0...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 :

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

où les X_i sont des variables à domaine fini, et les a_i et b des constantes entières, le propagateur fait la chose suivante :

Propagateurs de bornes

$$inf(X_i) := max\left(inf(X_i), \left\lceil \frac{b - \Sigma_{(1 \leq j \leq n, j \neq i)} a_j\, sup(X_j)}{a_i} \right\rceil\right)$$

$$sup(X_i) := min\left(sup(X_i), \left\lfloor \frac{b - \Sigma_{(1 \leq j \leq n, j \neq i)} a_j\, inf(X_j)}{a_i} \right\rfloor\right)$$

- On obtient donc $2n$ threads différents !
- Il s'agit d'un propagateur pour une relation linéaire entre n variables (et pas d'un propagateur pour la relation d'égalité)

Exemples (linear.oz)

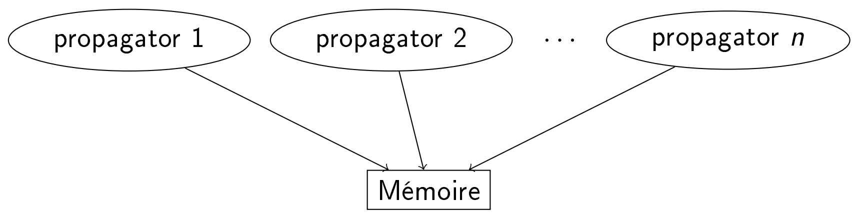
```
% propagateurs pour des relations lineaires

declare X1 X2 X3 X4 X5
L=[X1 X2 X3 X4 X5]
L:::1#100
{Browse L}

2*X1 + 3*X2 =<: 5*X3 - 6*X4 + X5

X3 -17*X4 >: 2*X1 + 5*X2 + 17*X5
```


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 $D\colon F \to 2^{\mathbb{N}}$.
- 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 $p\colon \mathcal{D} \to \mathcal{D}$ 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* : $\forall D_1, D_2 \in \mathcal{D}$: Si $D_1 \sqsubseteq D_2$ 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 α 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.

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
```

Programmation Logique et Par Contraintes Avancée Cours 6 – Propagateurs
└Propagateurs : un peu de théorie

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.

Programmation Logique et Par Contraintes Avancée Cours 6 – Propagateurs
└Propagateurs : un peu de théorie

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.

Programmation Logique et Par Contraintes Avancée Cours 6 – Propagateurs
└Propagateurs : un peu de théorie

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 évidement 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
- $i \to i + 1$: Hypothèse : $D \sqsubseteq D_i$. À montrer : $D \sqsubseteq D_{i+1}$
  - $p_i(D_i) = D_{i+1}$, où p_i propagateur de la i-ème étape.
  - $p_i(D) \sqsubseteq p_i(D_i)$ car p_i est monotone
  - $D = p_i(D)$ car D est un point fixe.
  - 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 est 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.

Exemples (colouring.oz) I

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

{Browse {SearchOne {MapColoring Europe 4}}}
```


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 austria] ]
```

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