# Algorithme de recherche - transformation de programmes

Gonzague YERNAUX

18 octobre 2016

# Fil rouge : calcul de maximum

```
int max(int x, int y, int z) {
  int max_xy;
  if (x>=y)
    max_xy = x;
  else
    max_xy = y;
  if (max_xy>=z)
    return max_xy;
  else
    return z;
}
```

# Unfolding

## Transformation 1 : unfolding

Permet de rembobiner le programme en remplaçant un appel par son corps effectif.

```
p1(A, B, C, D):- {A > 5}, p2(C).
p2(X) :- {X = 0}.
```

$\xrightarrow{unfold(p1)}$

```
p1(A, B, C, D):- {A > 5, C = X, X = 0}.
p2(X) :- {X = 0}.
```

$\xrightarrow{compress(p1)}$

```
p1(A, B, C, D):- {A > 5, C = 0}.
p2(X) :- {X = 0}.
```

# Unfolding "jusqu'au bout"

## Transformation 1b : unfolding "jusqu'au bout"

Unfolding complet du programme, au maximum, à partir d'un prédicat donné.

```
p1(A, B, C, D):- {A > 5}, p2(C,D).
p2(X, Y) :- {X = 0}, p3(Y).
p3(Q) :- { Q =< 10 }.
```

$\xrightarrow{unfoldAll(p1)}$

```
p1(A, B, C, D):- {A > 5, C = 0, D =< 10}
p2(X, Y) :- {X = 0}, p3(Y).
p3(Q) :- {Q =< 10}.
```

## Observation

L'unfolding complet revient à exécuter le programme CLP

# Pruning

## Transformation 2 : pruning

"Nettoyage" des clauses inutiles. Peut servir à différents stades de la transformation, notamment pour améliorer les performances et comme condition d'arrêt.

```
p1(A, B, C, D):- {A > 5, C = 0, D =< 10}.
p2(X, Y) :- {X = 0}, p3(Y).
p3(Q) :- { Q =< 10 }.
p4(A,B,C,D):- {}.
```

$\overset{prune(p1)}{\Longrightarrow}$ `p1(A, B, C, D):- {A > 5, C = 0, D =< 10}.`

# Slicing

## Transformation 3 : Slicing

Création de tranches de programme isolant l'influence des variables.

```
p1(A, B, C, D):- {A > B}, p2(C, D).
p2(X, Y) :- {X = 0, Y > 1}.
```

$\xrightarrow{slice}$

```
[A, B] p1(A,B_,_) :- {A > B}.
[C, D] p1(_, _, C, D) :- {}, p2(C, D).
[C, D] p2(X, Y) :-  {X = 0, Y > 1}.
```

## Observation

C et D sont liées dans la slice alors qu'en réalité elles sont indépendantes. Solution : *unfolder* avant de *slicer*.

# Comparaison de programmes simples

```
int max1(int x, int
    y, int z) {
  int max_xy;
  if (x>=y)
    max_xy = x;
  else
    max_xy = y;
  if (max_xy>=z)
    return max_xy;
  else
    return z;
}
```

```
int max2(int x, int y,
    int z) {
  if ((x>=y) && (x>=z))
    return x;
  else if ((y>=x) && (y
      >=z))
    return y;
  else
    return z;
}
```

# Algorithme de comparaison - étape 1

## Étape 1

Dans chaque programme, choisir une slice, puis unifier les clauses ayant la même tête dans la slice.

```
[A,B] p1(A, B, _, A):- {A > B}.
[A,B] p1(A, B, _, A) :- {A = B}.
[C] p1(_, _, C, _) :- {C > 0}, p2(C).
[C] p2(X) :- {X < 50}.
```

$\overset{etape1}{\Longrightarrow}$ `[A, B] p1(A,B,C,A) :- {(A > B) \/ (A = B)}.`

# Algorithme de comparaison - étape 2

## Étape 2

- Créer une association de variables du programme 1 vers le programme 2
- Si, pour tout couple de prédicats ayant la même tête (P1, P2) où P1 appartient au programme 1 et P2 au programme 2, cette association rend les contraintes de P1 vraies ssi celles de P2 sont vraies, alors les deux programmes calculent la même chose
- Sinon, créer une autre association de variables et réessayer
- Jusqu'à ce que toutes les associations de variables possibles aient été essayées

# Perspectives

- Généraliser l'algorithme
  - Équivalence de programmes comme défni dans PPDP
  - Faire abstraction du nombre d'arguments et de leur position
  - Ne pas déterminer la slice choisie statiquement
- *Refactorer* le code : transformations robustes (et paramétrables)
- Programmes récursifs
- Heuristique pour l'ordre des transformations