Algorithme de recherche - transformation de programmes

Gonzague YERNAUX

21 novembre 2016

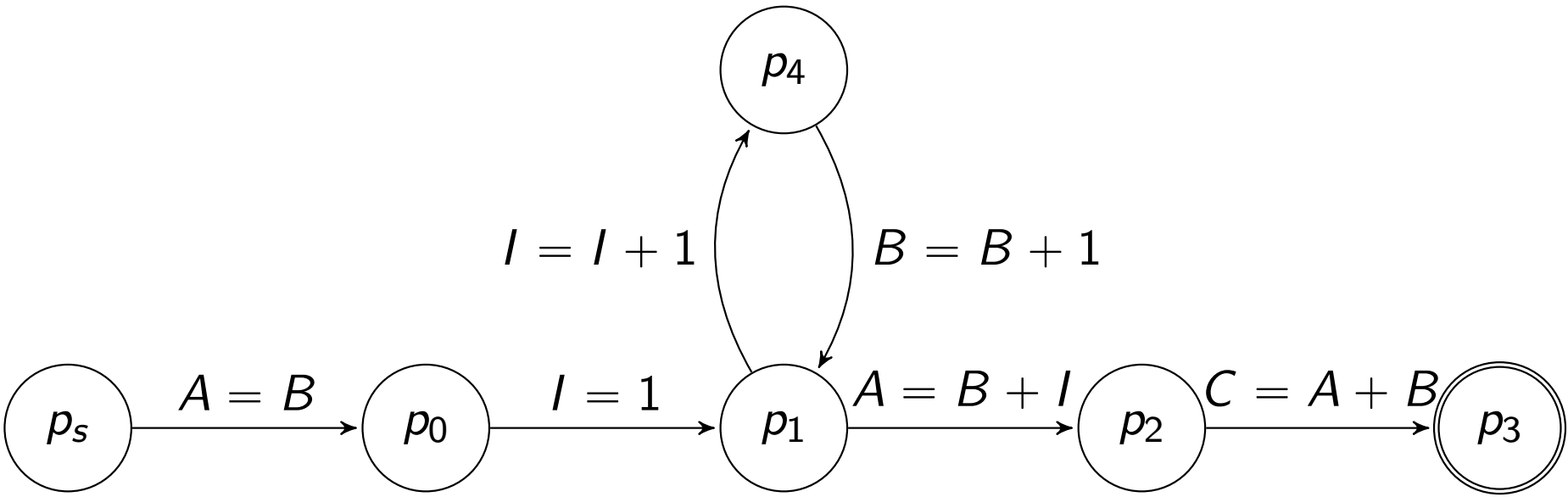
Précédemment

- Transformations pour programmes non-cycliques : simplification, pruning, unfolding, slicing
- Comparaison "naïve" de programmes non-cycliques

Sommaire

Prise en compte des programmes cycliques

Notation "automates"

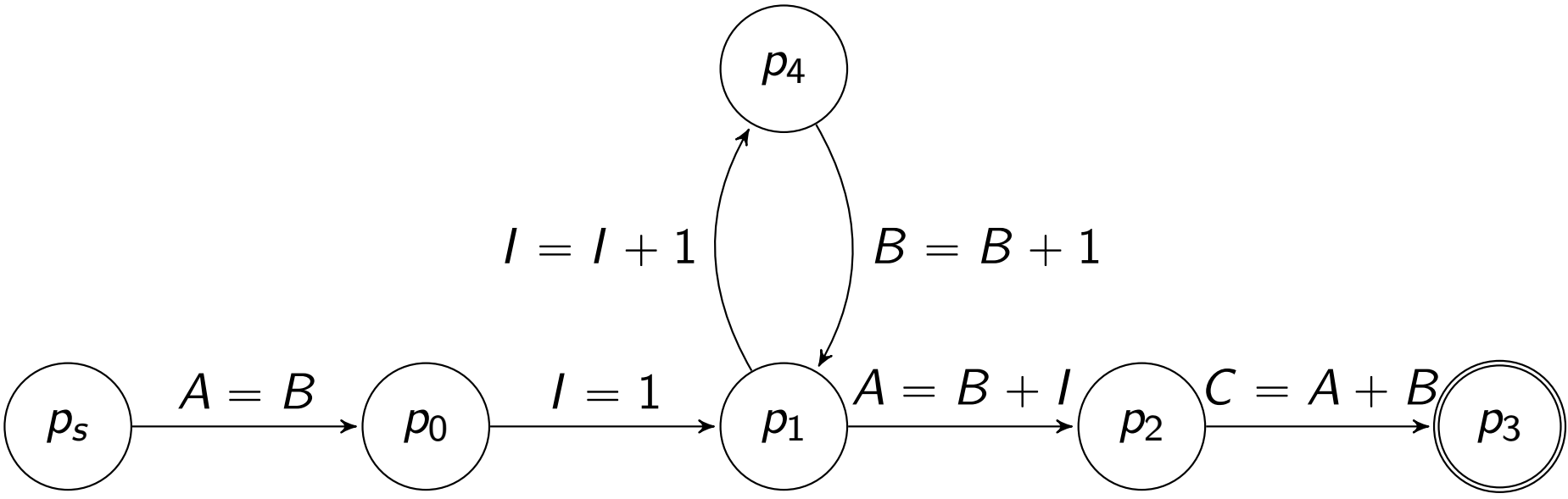


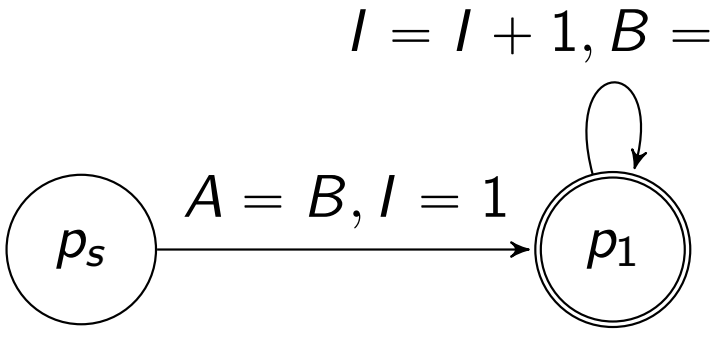
Programmes cycliques

Unfolding pour programmes cycliques

- Unfolder chaque clause d'un prédicat de base X...
- ... jusqu'à ce qu'il ne soit plus possible de trouver, dans les appels de X, de prédicat qui ne fasse pas partie d'un cycle *ou* qui ne soit appelé par aucun autre prédicat.
- $C :=$ l'ensemble des prédicats appelés par X (toutes clauses comprises) et non encore visités. Recommencer l'opération sur chaque prédicat de C.

Programmes cycliques - unfolding





Programmes cycliques - unfolding

Loop unrolling

Déroulage des boucles avec un incrément dont la valeur initiale est fixée.

```
p(A,B):- {I = 0}, p2(I,A,B).
p2(I,A,B):- {I < 5, J = I + 1, C = A + A},
       p2(J,C,B).
p2(I,A,B):- {I = 5, B = A}.
```

$$\downarrow unroll(p)$$

```
p(A, B):- {B = 32*A}.
```

Programmes cycliques

Slicing pour programmes cycliques

Algorithme : pour déterminer quelles variables font partie de la tranche d'une variable X, explorer toutes les transitions possibles dans l'automate du programme, et "marquer" les positions d'arguments par lesquelles passent X ou des variables liées à X par une contrainte.

Programmes cycliques

Slicing pour programmes cycliques

Algorithme : pour déterminer quelles variables font partie de la tranche d'une variable X, explorer toutes les transitions possibles dans l'automate du programme, et "marquer" les positions d'arguments par lesquelles passent X ou des variables liées à X par une contrainte.

Slicing "intelligent"

Union des slices dont les variables d'intérêt sont au moins une fois contraintes. (= Suppression ciblée des variables qui n'ont jamais d'intérêt.)

Nouvelles transformations

Nouvelles transformations

Renaming

- Renommage de prédicats
- Renommage de variables
  - Simple
  - Complet

Nouvelles transformations - renaming

```
p1(A,B,A,X) :- {Z = 3}, p2(A,Z), p3(B).
```

Nouvelles transformations - renaming

```
p1(A,B,A,X) :- {Z = 3}, p2(A,Z), p3(B).
```

Renommage de prédicats

```
q(A,B,A,X) :- {Z = 3}, r(A,Z), s(B).
```

Nouvelles transformations - renaming

```
p1(A,B,A,X) :- {Z = 3}, p2(A,Z), p3(B).
```

Renommage de prédicats

```
q(A,B,A,X) :- {Z = 3}, r(A,Z), s(B).
```

Renommage de variables simple

```
q(V1,V2,V1,V3) :- {V4 = 3}, r(V1,V4), s(V2).
```

Nouvelles transformations - renaming

```
p1(A,B,A,X) :- {Z = 3}, p2(A,Z), p3(B).
```

Renommage de prédicats

```
q(A,B,A,X) :- {Z = 3}, r(A,Z), s(B).
```

Renommage de variables simple

```
q(V1,V2,V1,V3) :- {V4 = 3}, r(V1,V4), s(V2).
```

Renommage de variables complet

```
q(V1,V2,V3,V4) :- {V5 = 3, V1 = V3},
     r(V1,V5), s(V2).
```

Nouvelles transformations

Definition

Définition d'une nouvelle clause qui ne change pas le sens du programme

```
p(A,B,C):- q(A,B), r(B,C).
q(X,X).
r(X,X).
```

Definition

```
p(A,B,C):- q(A,B), r(B,C).
q(X,X).
r(X,X).
def(A,B,C,D):- q(A,C), r(C,D).
```

Nouvelles transformations

Folding

Opération inverse de l'unfolding : recherche d'une clause C ayant comme corps un sous-ensemble du corps de la clause à folder et remplacement de ce dernier par un appel à C.

```
p(A,B,C):- q(A,B), r(B,C).
q(X,X).
r(X,X).
def(A,B,C,D):- q(A,C), r(C,D).
```

$$\downarrow fold(p)$$

```
p(A,B,C):- def(A,B,B,C).
q(X,X).
r(X,X).
def(A,B,C,D):- q(A,C), r(C,D).
```

Comparaison de programmes

Comparaison

1. Predicate pairing
2. Résolution CLP
3. Comparaison "syntaxique"
   - Pistes explorées
   - Piste à explorer pour la suite

Comparaison - predicate pairing

Predicate pairing

Série de transformations fold/unfold permettant d'unir deux programmes à l'aide d'une conjonction.

Comparaison - predicate pairing

```
p1(A,B):- {I = 1}, p2(A,B,I).
p2(X,X,Z):- {Z = 10}.
p2(X,Y,Z) :- {Z < 10, Z1 = Z + 1, X1 = X +
    5}, p2(X1, Y, Z1).

q1(A,B):- q2(A,B).
q2(A,B):- {B = A + 50}.
```

$$\downarrow pair(p1, q1)$$

```
launchingClause(A,B,C,D):- p1(A,B), q1(C,D).
def1(A,B,C,D):- p1(A,B), q1(C,D).
```

Comparaison - predicate pairing

```
p1(A,B):- {I = 1}, p2(A,B,I).
p2(X,X,Z):- {Z = 10}.
p2(X,Y,Z) :- {Z < 10, Z1 = Z + 1, X1 = X +
    5}, p2(X1, Y, Z1).

q1(A,B):- q2(A,B).
q2(A,B):- {B = A + 50}.
```

$$\downarrow pair(p1, q1)$$

```
launchingClause(A,B,C,D):- def1(A,B,C,D).
def1(A,B,C,D):- {I = 1}, p2(A,B,I), q2(C,D).
def2(A,B,C,D,E):- p2(A,B,C), q2(D,E).
```

Comparaison - predicate pairing

```
p1(A,B):- {I = 1}, p2(A,B,I).
p2(X,X,Z):- {Z = 10}.
p2(X,Y,Z) :- {Z < 10, Z1 = Z + 1, X1 = X +
    5}, p2(X1, Y, Z1).

q1(A,B):- q2(A,B).
q2(A,B):- {B = A + 50}.
```

$$\downarrow pair(p1, q1)$$

```
launchingClause(A,B,C,D):- def1(A,B,C,D).
def1(A,B,C,D):- {I = 1}, def2(A,B,I,C,D).
def2(A,B,C,D,E):- {C = 10, A = B, E = D +
    50}.
def2(A,B,C,D,E):- {C = 10, C1 = C + 1, A1 = A
     + 5, E = D + 50}, p2(A1, B, C1).
```

Comparaison - résolution CLP

On va ajouter des contraintes à la clause de lancement qui nous permettront de tester l'égalité des valeurs de retour.

```
launchingClause(A1,B1,C1,A2,B2,C2):-
  {A1 = A2, B1 = B2, C1 \= C2},
  def1(A1,B1,C1,A2,B2,C2).
```

Formalisation

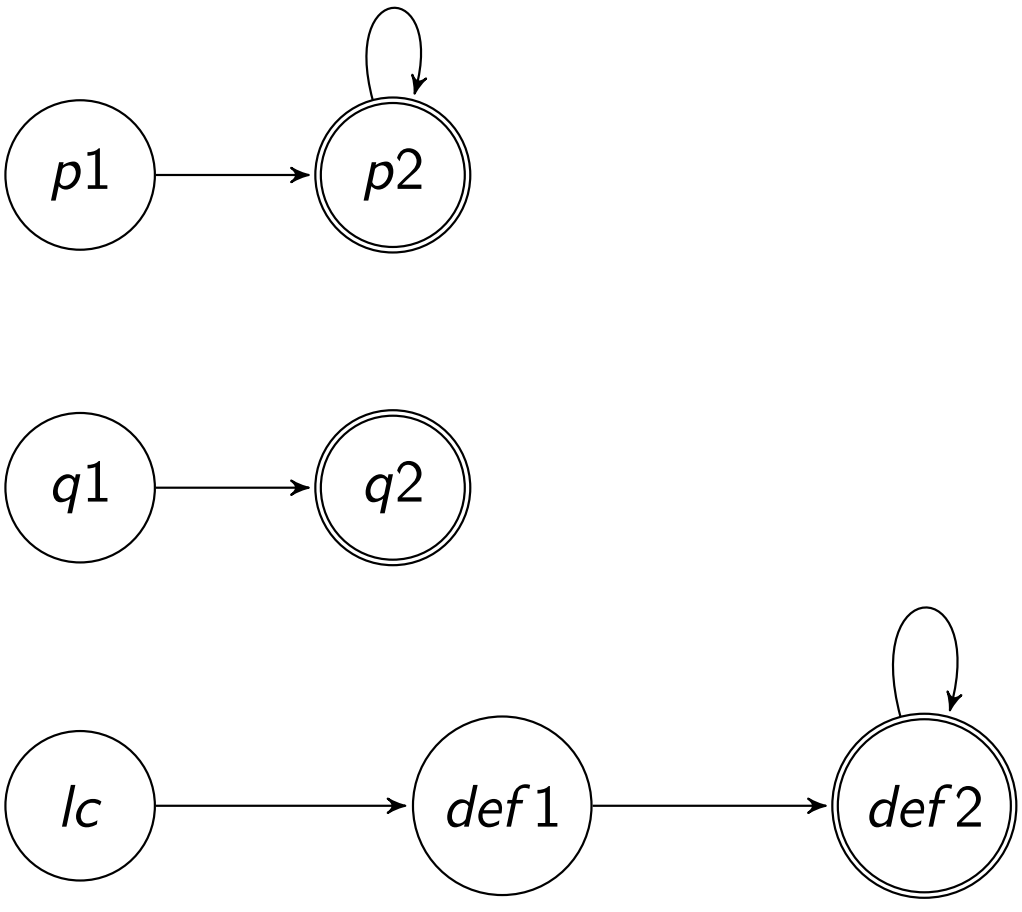
- Sélection des arguments d'entrée et de sortie
- Création de renommages des arguments d'entrée d'un programme vers ceux de l'autre programme.
- But : trouver un renommage qui, une fois appliqué, rend impossible l'obtention de valeurs de sortie différentes.

Comparaison syntaxique - pistes explorées

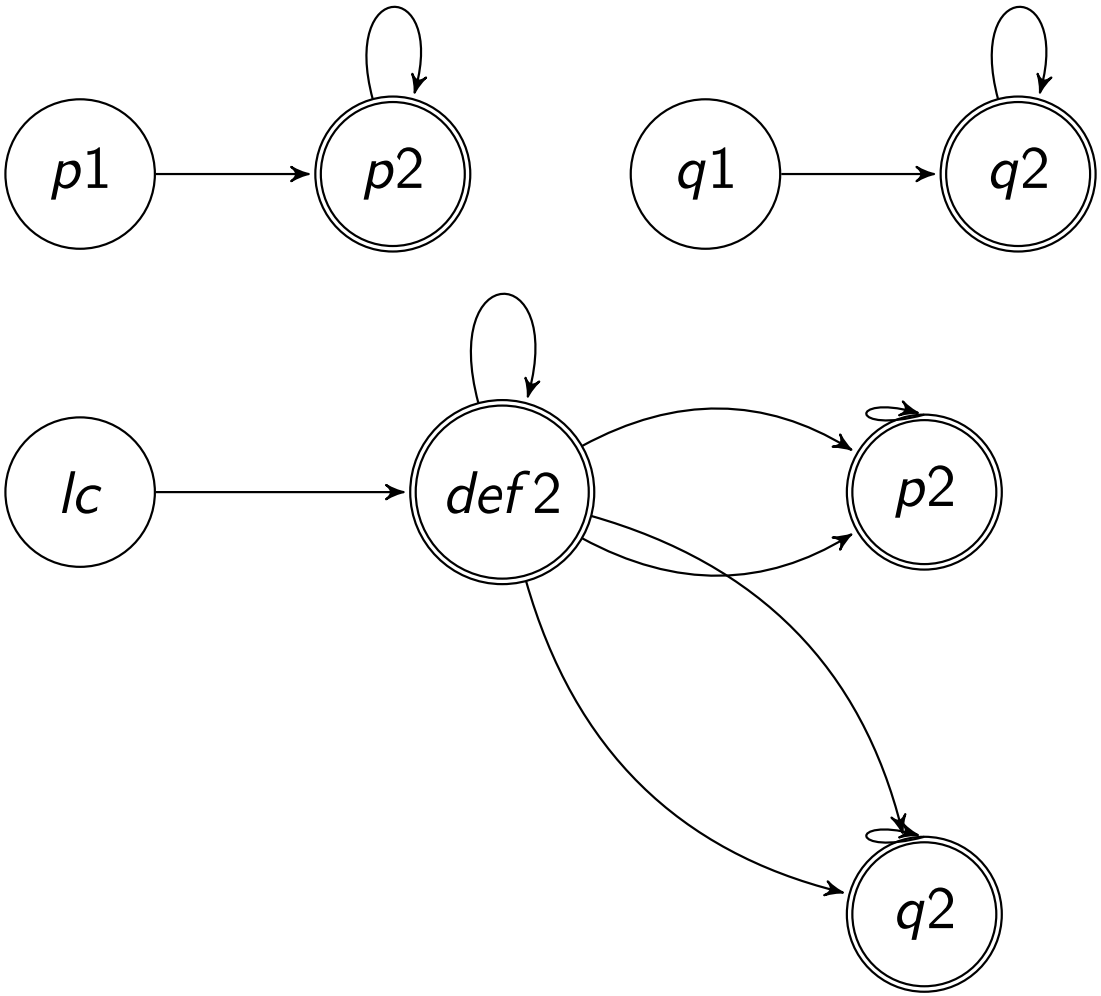
Pistes explorées :

- Approche "automates"
- Predicate pairing syntaxique
- ...

Comparaison syntaxique - predicate pairing (exemple 1)



Comparaison syntaxique - predicate pairing (exemple 2)



Comparaison syntaxique - piste à explorer

Piste à explorer : l'algorithme "naïf" de comparaison

Comparaison syntaxique - piste à explorer

Piste à explorer : l'algorithme "naïf" de comparaison

Soient A et B deux programmes à comparer, et T un ensemble de transformations. Soient $SA := \{A\}$ et $SB := \{B\}$

Comparaison syntaxique - piste à explorer

Piste à explorer : l'algorithme "naïf" de comparaison

Soient A et B deux programmes à comparer, et T un ensemble de transformations. Soient $SA := \{A\}$ et $SB := \{B\}$

Tant que c'est intéressant de continuer et que $SA \cap SB = \emptyset$:
{

Comparaison syntaxique - piste à explorer

Piste à explorer : l'algorithme "naïf" de comparaison

Soient A et B deux programmes à comparer, et T un ensemble de transformations. Soient $SA := \{A\}$ et $SB := \{B\}$

Tant que c'est intéressant de continuer et que $SA \cap SB = \emptyset$:
{
Choisir un programme P dans SA, un programme P' dans SB, une transformation T dans T, une transformation T' dans T, ajouter PT := T(P) et PT' := T'(P') à SA et SB respectivement.
}

Comparaison syntaxique - piste à explorer

Piste à explorer : l'algorithme "naïf" de comparaison

Soient A et B deux programmes à comparer, et T un ensemble de transformations. Soient $SA := \{A\}$ et $SB := \{B\}$

Tant que c'est intéressant de continuer et que $SA \cap SB = \emptyset$:
{
  Choisir un programme P dans SA, un programme P' dans SB, une transformation T dans T, une transformation T' dans T, ajouter PT := T(P) et PT' := T'(P') à SA et SB respectivement.
}

Si $SA \cap SB \neq \emptyset$ alors on a trouvé. Sinon les deux algorithmes sont différents.

Comparaison syntaxique - piste à explorer

Questions que soulève l'algorithme naïf de comparaison :

- "Tant que c'est intéressant de continuer"
- Quelles transformations choisir ? (Aide du predicate pairing ?)
- Y a-t-il vraiment des transformations utilisables plus d'une fois ?
- Performances : utiliser les indices de la résolution CLP ?
- Boucles impossibles à dérouler
- Simplification et propagation de contraintes
- ...

Remarques sur le paradigme CLP

Remarques sur le paradigme CLP

L'exécution d'un programme CLP et le *constraint store* :

- La comparaison se fait sur le plus petit domaine possible de chaque variable.
- Optimisation : à l'unfolding (par exemple), supprimer les clauses dont les contraintes sont insatisfiables.
- Possibilité d'utiliser des contraintes **réifiées**.
- Création d'un petit outil de test des programmes CLP.

Importance de l'ordre :

- des clauses d'un prédicat (récursif) ;
- des *goals* d'une clause.