Un nouveau modèle pour l'analyse de programmes récursifs

Albert Cohen

PRiSM, Université de Versailles

Vendredi 13 Novembre 1998

Introduction

- **But : effectuer des transformations de code complexes sur les programmes récursifs**
  - Optimisation en temps et en espace pour langages de haut niveau
  - Parallélisation
- **Condition nécessaire : connaissance précise sur le flot des données, à la compilation.**
  - Déterminer cette information pour chaque instance d'une instruction à l'exécution

    Notion d'opération, généralisée aux programmes récursifs !
  - Décrire des ensembles et des relations entre opérations

Plan de l'exposé

- **Présentation générale**

1. Généralités
2. Un exemple "à la main"
3. Résultats et applications de l'analyse

- **Partie technique**

1. Modélisation
2. Détection des opérations en conflit
3. Calcul des dépendances
4. Calcul des définitions visibles

Propriétés du flot des données, au niveau des opérations (1/2)

- **Boucles : vecteurs d'itération**

Chaque itération génère une nouvelle instance de chaque instruction (opération)

Les ensembles et les relations sont des systèmes d'inégalités affines (modèle polyédrique)

Exemple de propriétés :

$$P(i,j) \iff i + j = N$$
$$P(i,j,i',j') \iff i + i' = N \ \wedge\ 1 \leq i,j,i',j' \leq M$$

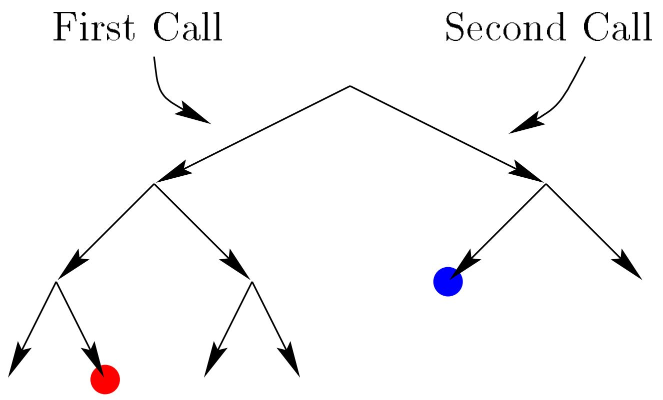
- **Programmes récursifs ?**

Problème : nommer chaque nœud dans l'arbre d'appel

Propriétés du flot des données, au niveau des opérations (2/2)

```
F () {
  F ();
  F ();
}
```

Call Tree



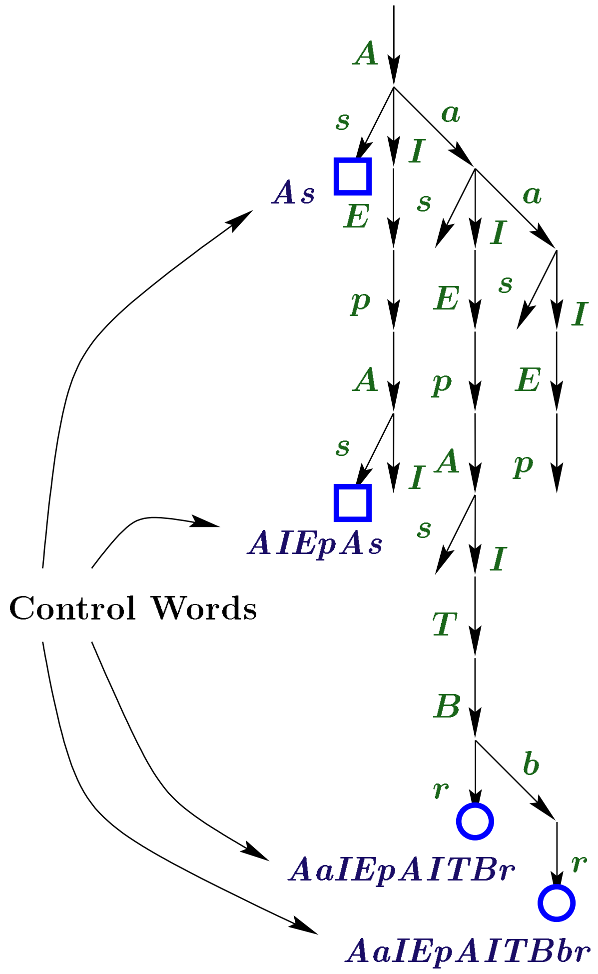
$\cancel{P(\texttt{F})}$

Properties: $P(\texttt{F}$ at ●$)$

$P($between $\texttt{F}$ at ● and at ●$)$

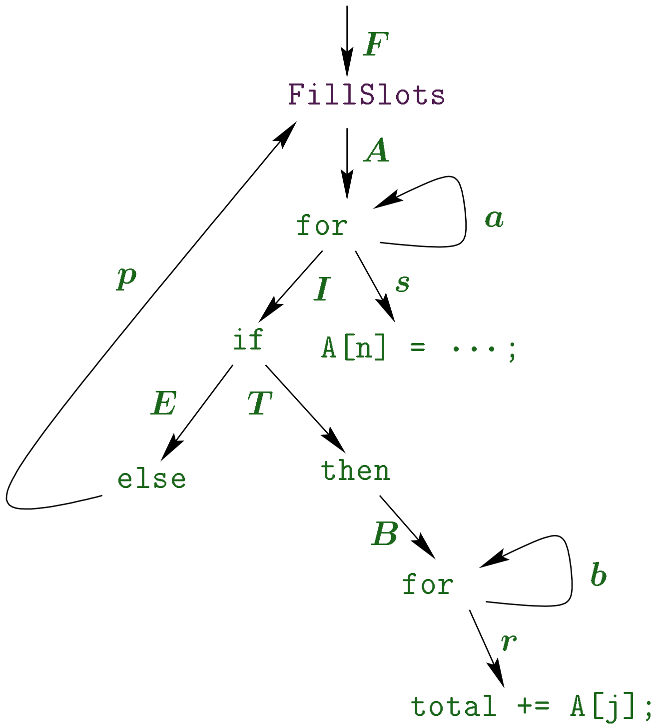
Arbre d'appel et mots de contrôle

```
     void FillSlots (int n) {
A/a    for (i = 0; i <= M; i++) {
s        A[n] = ...;
I        if (n == 0)
T          { /* then */
B/b          for (j = 0; j <= N; j++)
r              ... = ... A[j] ...;
E          } else
p            FillSlots (n-1);
       }
     }
     void main () {
       FillSlots (N);
     }
```



Automate de contrôle

```
    void FillSlots (int n) {
A/a   for (i = 0; i <= M; i++) {
s       A[n] = ...;
I       if (n == 0)
T         { /* then */
B/b         for (j = 0; j <= N; j++)
r             ... = ... A[j] ...;
E         } else
p           FillSlots (n-1);
      }
    }
    void main () {
      FillSlots (N);
    }
```

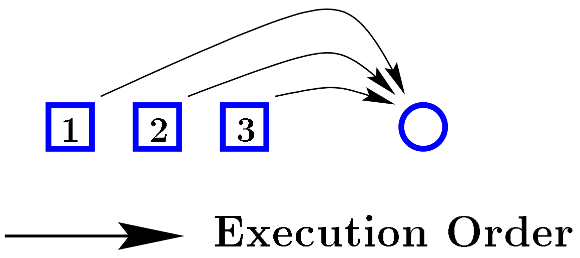


Propriété du flot : dépendances & définitions visibles

- **Dépendance vraie**

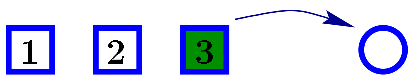
  – Une opération effectuant une écriture (ou définition) et une opération effectuant une lecture (ou utilisation) accèdent à la même case mémoire

  – L'écriture s'exécute avant la lecture



- **Définition visible**

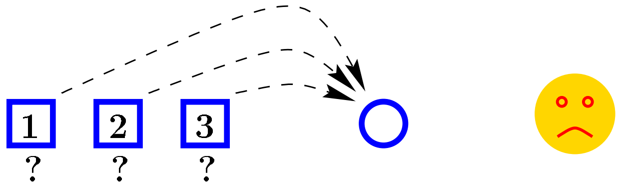
La dernière écriture dans la même case mémoire



Analyse de définitions visibles

- **Problème : le schéma exact de l'exécution n'est pas prévisible**

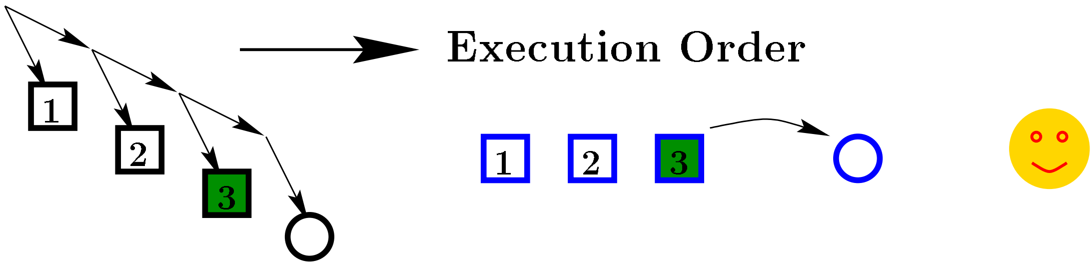
$\Longrightarrow$ *Impossible de prouver ou de réfuter l'exécution d'une instance donnée*



- **Solution : utiliser des connaissances supplémentaires**

Propriétés structurelles, étude de bornes de boucles et d'expressions conditionnelles, etc.

Exemple : les chemins de la racine à l'instance de la lecture



Première étape : calculer les fonctions d'accès

```
      void FillSlots (int n) {
A/a     for (i = 0; i <= M; i++) {
s         A[n] = ...;
I         if (n == 0)
T           { /* then */
B/b           for (j = 0; j <= N; j++)
r               ... = ... A[j] ...;
E           } else
p             FillSlots (n-1);
        }
      }
      int main () {
        FillSlots (N);
      }
```

– Variable j : initialisée à **0**, incrémentée par la boucle interne

– Variable n : initialisée à N, décrémentée à chaque appel récursif

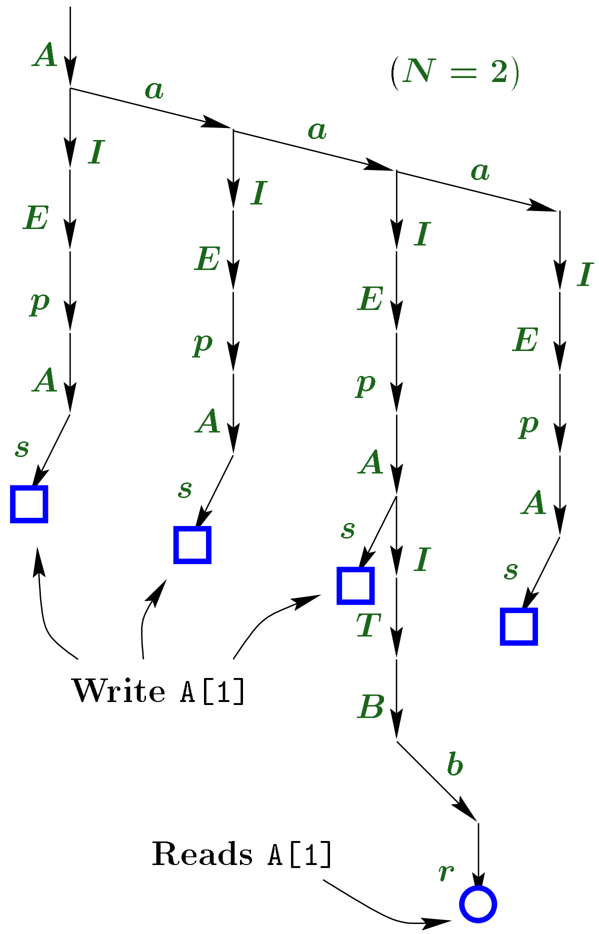
▶ **Calcul de j et n pour une opération donnée :**

– Nombre de bs dans le mot de contrôle $\longrightarrow$ j

– Nombre de ps dans le mot de contrôle $\longrightarrow$ $N - n$

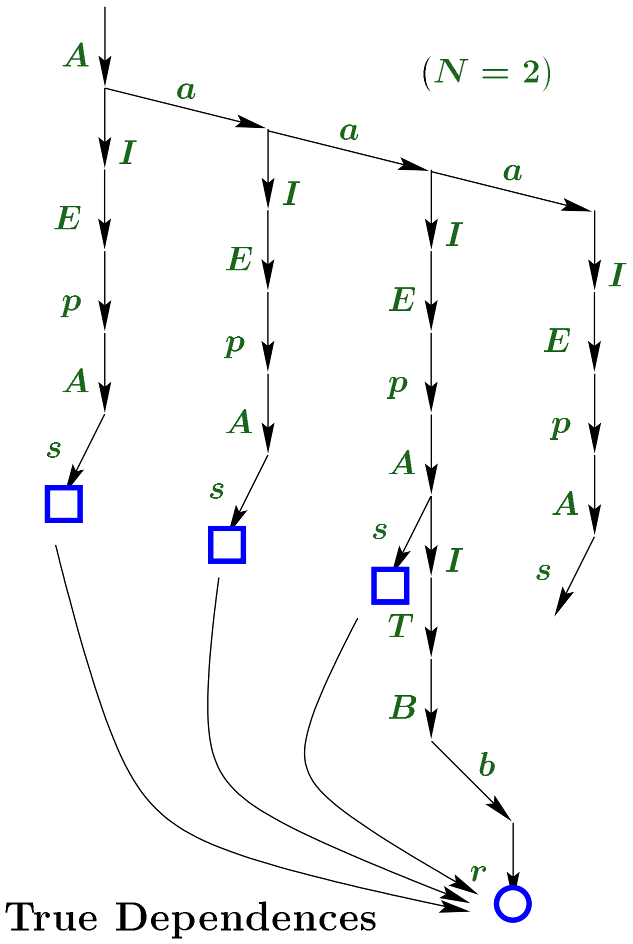
Deuxième étape : détection des conflits d'accès

- **Conflit (même case mémoire) $\iff$ nombre de bs de la lecture plus nombre de ps de l'écriture write vaut N**



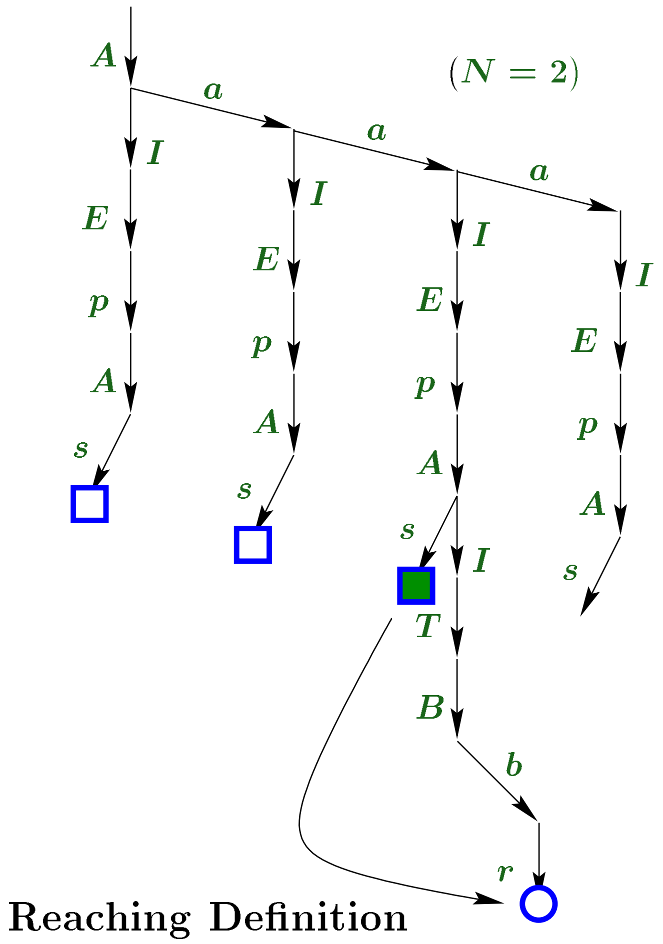
Troisième étape : ordre d'exécution séquentiel

- L'exécution séquentielle des opérations suit l'ordre lexicographique sur les mots de contrôle



Quatrième étape : retirer les écritures écrasées

- **Propriété structurelle : les opérations situées sur un chemin de la racine à une lecture s'exécutent toujours**

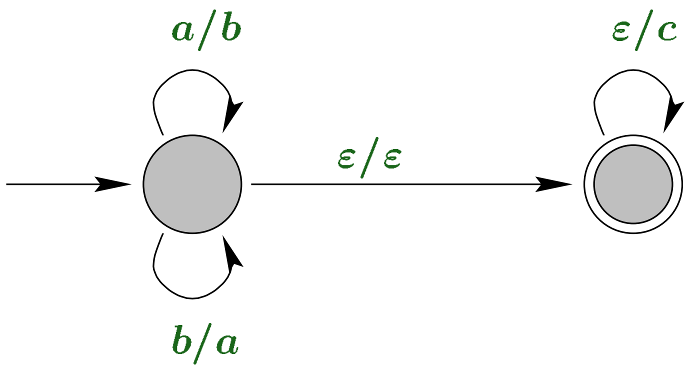


À la recherche d'une bonne abstraction

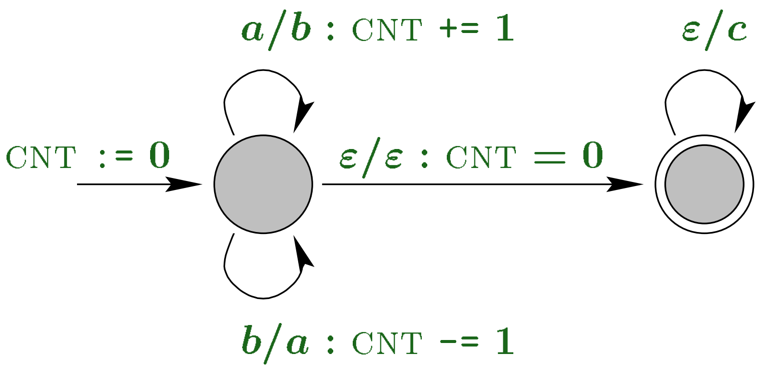
- **Relation entre mots : transduction**
- **Transducteur : automate avec une sortie**

  "Code" une transduction

- **Exemples de transducteur rationnel et à pile**



$aab b$ gives $bbaac^*$
abb gives $baac^*$



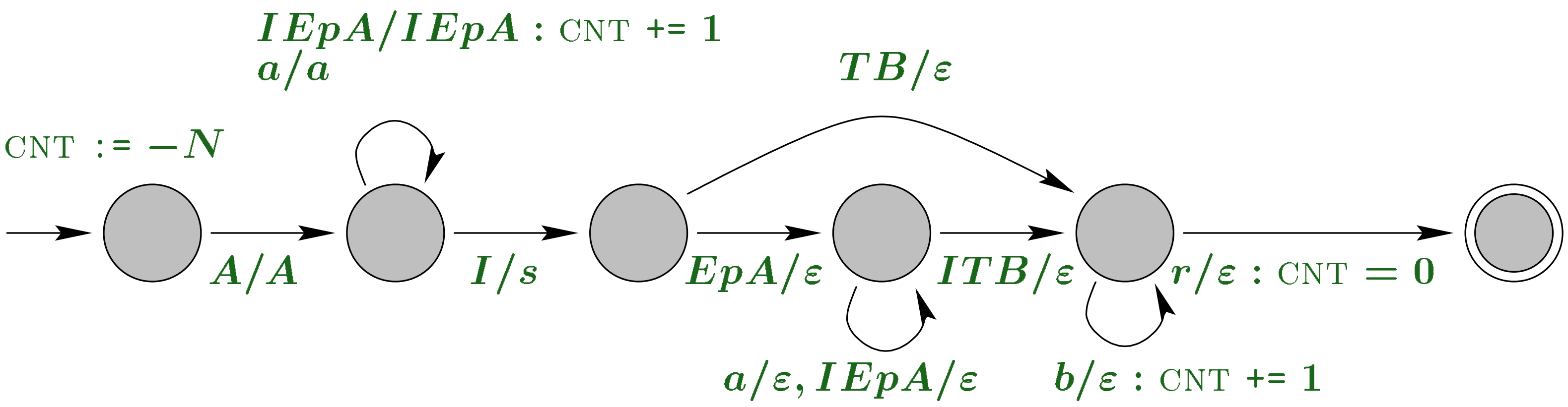
$aabb$ gives $bbaac^*$
abb is not accepted (CNT $\neq 0$)

Quel est le résultat ?

- **Dépendances : transducteur rationnel (arbres) ou à pile (tableaux)**
- **Définitions visibles : approximation par un transducteur rationnel ou à pile**

$\Longrightarrow$ ensembles de définitions visibles possibles

- **Analyse de définitions visibles pour** `FillSlots`



Applications (1/2)

- **Propriétés de décidabilité pour les transducteurs à pile**
  - Vérifier qu'une opération est une définition visible possible d'une autre
  - Pas de définition visible $\longrightarrow$ détection des variables non initialisées
  - Toujours une définition visible unique $\longrightarrow$ détecter les résultats exacts
  - Les propriétés décidables sur un mot de contrôle donné sont toujours décidables pour des langages rationnels de mots de contrôle

Applications (2/2)

- **Parallélisation (semi-) automatique**
  - `FillSlots` ne contient pas d'instructions exécutables en parallèle
  - Mise en assignation unique en utilisant les définitions visibles
  - `FillSlots` en assignation unique possède beaucoup de parallélisme
- **Les transducteurs permettent l'expansion de la mémoire et détectent le parallélisme automatiquement**
- **La génération de code efficace est difficile**

Expansion & parallélisation

Assignation unique

```
       void FillSlots (int n) {
A/a      for (i = 0; i <= M; i++) {
s          T[w] = ···;
I          if (n == 0)
T          { /* then */
B/b          for (j = 0; j <= N; j++)
r              ··· = ··· T[RD(w)] ···;
E          } else
p            FillSlots (n-1);
         }
       }
```

FillSlots parallèle

```
void Writes (int n, int i) {
  { if (n >0) Writes (n - 1, 0); }
  || { if (i <= M) Writes (n, i + 1); }
  T[w] = ···;
}
void Reads (int n, int i) {
  { if (n == 0)
      for (j = 0; j <= N; j++)
        ··· = ··· T[RD(w)] ···;
    else Reads (n-1, 0); }
  || { if (i <M) Reads (n, i + 1); }
}
void main () {
  Writes (N, 0);
  barrier (); /* wait for writes */
  Reads (N, 0);
}
```

Comment ça marche ?

Modèle de programmes

– Contrôle : appels récursifs, boucles et expressions conditionnelles quelconques.
Abstraction : langage des mots de contrôle

– Scalaires, tableaux unidimensionnels, arbres (une racine, pas d'arêtes en arrière).
Abstraction : monoïde.

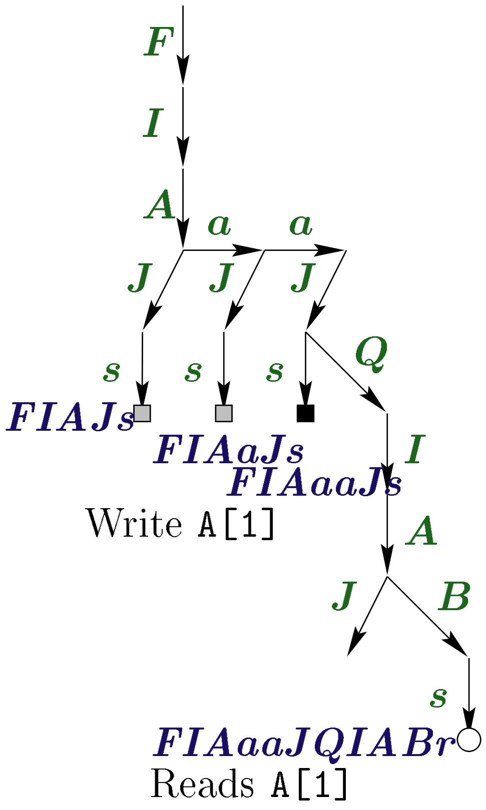
Variables inductives

– Arguments entiers d'une procédure :
`F (i), F (j+1), F (k-3), F (5)`

– Arguments pointeurs-sur-arbre d'une procédure :
`G (p), G (q->right), G(r->right->left->right), G(root)`

Nouvel exemple : problème des n-reines

```
    void Queens (int n, int k) {
I     if (k <= n) {
A/a     for (i=1; i<=n; i++) {
B/b       for (j=1; j<k; j++)
r           ··· = ··· A[j] ···;
J         if (···) {
s           A[k] = ···;
Q           Queens (n, k+1);
          }
        }
      }
    }
    int main () {
F     Queens (n, 1);
    }
```



Calcul des fonctions d'accès

$\texttt{A}[f(w')]\ \texttt{=}\ \cdots$

$\cdots\ \texttt{=}\ \cdots\ \texttt{A[g(w)]}\ \cdots$

- **Trois types d'équations entre variables inductives...**

  $$[\![\mathtt{i}]\!](u.\alpha) = [\![\mathtt{i}]\!](u) \bullet \beta \qquad [\![\mathtt{i}]\!](u.\alpha) = [\![\mathtt{j}]\!](u) \bullet \beta \qquad [\![\mathtt{i}]\!](u.\alpha) = \beta$$

  ($\bullet$ représente l'addition d'entiers ou la concaténation de mots)

- **... Et des équations sur les fonctions affines...**

  $$[\![2 * \mathtt{i} + \mathtt{j}]\!](u) = 2[\![\mathtt{i}]\!](u) + [\![\mathtt{j}]\!](u)$$

- **... Engendrent un système d'équations récurrentes, que l'on sait résoudre automatiquement :**

Les fonctions d'accès sont des transductions rationnelles, du langage des mots de contrôle vers le monoïde des données

Fonctions d'accès pour `Queens`

- **Équations récurrentes**

$$\text{La boucle } \texttt{for} \text{ sur } \mathtt{j} : [\![\mathrm{j}]\!](\boldsymbol{u.B}) = \mathbf{1} \text{ and } [\![\mathrm{j}]\!](\boldsymbol{u.b}) = [\![\mathrm{j}]\!](\boldsymbol{u}) + \mathbf{1}$$

$$\text{L'appel principal} : [\![\mathrm{k}]\!](\boldsymbol{F}) = \mathbf{1}$$

$$\text{Les appels récursifs} : [\![\mathrm{k}]\!](\boldsymbol{u.Q}) = [\![\mathrm{k}]\!](\boldsymbol{u}) + \mathbf{1}$$

- **Exemple : Fonction d'accès en écriture `A[k]`**

$$\{(\boldsymbol{w'}/\boldsymbol{f}(\boldsymbol{w'})) : \ \boldsymbol{f}(\boldsymbol{w'}) = [\![\mathrm{k}]\!](\boldsymbol{w'})\} = (\boldsymbol{FIA}/\mathbf{1}).\left((\boldsymbol{JQIA}/\mathbf{1}) + (\boldsymbol{a}/\varepsilon)\right)^*.(\boldsymbol{Js}/\varepsilon)$$

Calcul du transducteur du conflit

```
A[f(w')] = ...
... = ... A[g(w)] ...
```

- **Conflit d'accès (même case mémoire) lorsque**

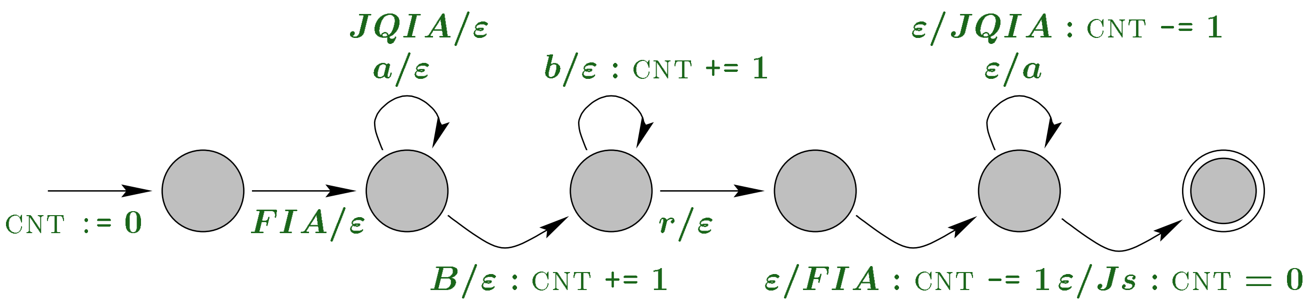
$$g(w) = f(w'), \text{ c'est à dire, } w' \in f^{-1} \circ g(w)$$

- **Composition de deux transducteurs rationnels ?**

Arbres $\longrightarrow$ transducteur rationnel : les données sont dans un monoïde libre, théorème de Elgot et Mezeï

Tableaux $\longrightarrow$ transducteur à pile : les données sont dans $\mathbb{Z}$

Transducteur du conflit pour `Queens`



Calcul du transducteur des dépendances

- **Intersection du transducteur du conflit et de l'ordre lexicographique**
- **Problème de clôture par intersection !**

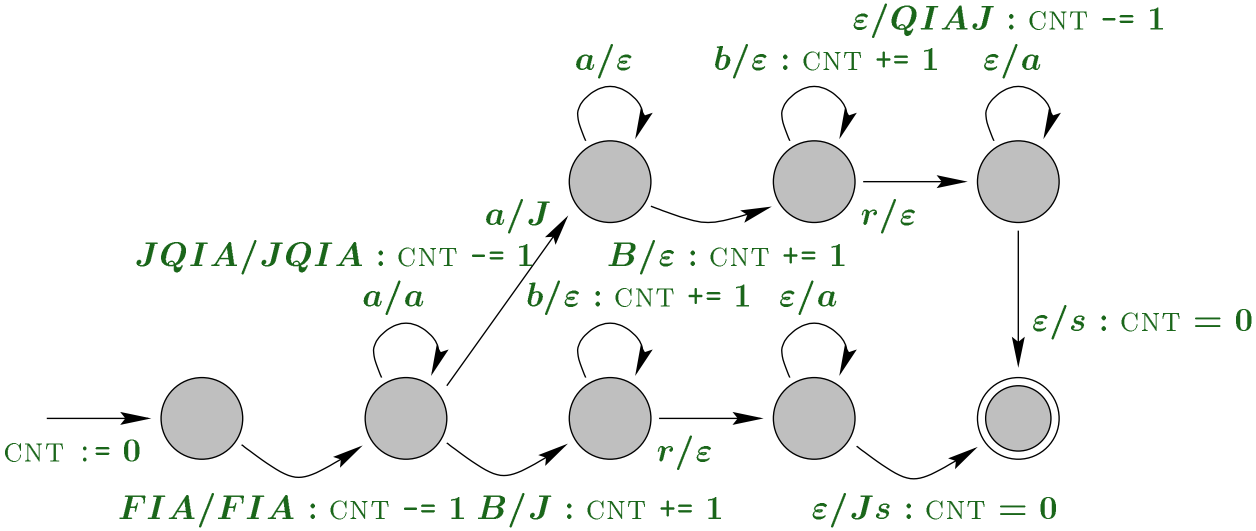
On se ramène à des classes de transducteurs closes pour l'intersection avec l'ordre lexicographique

- **Méthode : élimination partielle des ε dans les transitions**

Arbres $\longrightarrow$ transducteur rationnel, avec parfois des approximations conservatives, de précision contrôlable

Tableaux $\longrightarrow$ transducteur à pile, sans approximation

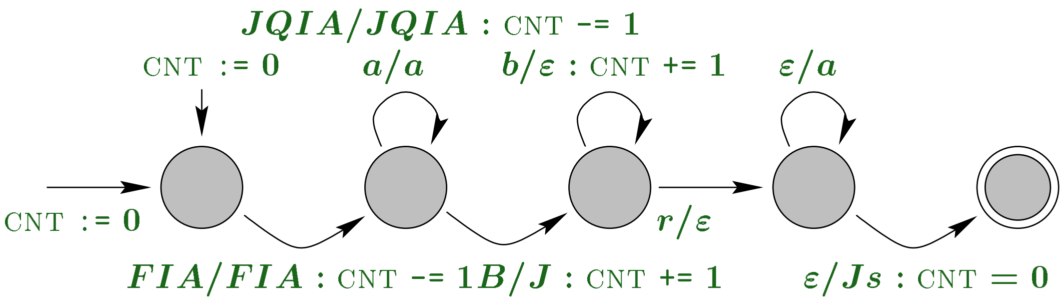
Transducteur des dépendances pour `Queens`



Calcul du transducteur des définitions visibles

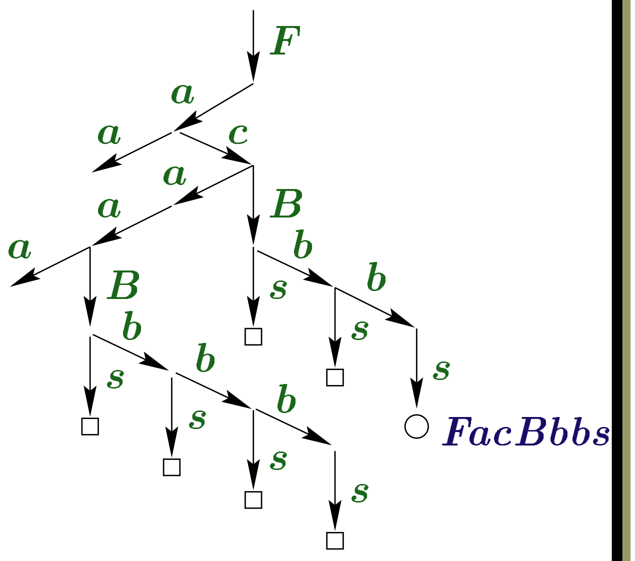
- **Hypothèse : propriétés statiques sur l'exécution ou la non-exécution d'opérations**
- **Application de règles de réécriture sur les transducteurs**

  $\longrightarrow$ Élimination de transitions
- **Exemple : valeurs produites sur le chemin de la racine à la lecture**
- **Résultat pour le programme `Queens` :**



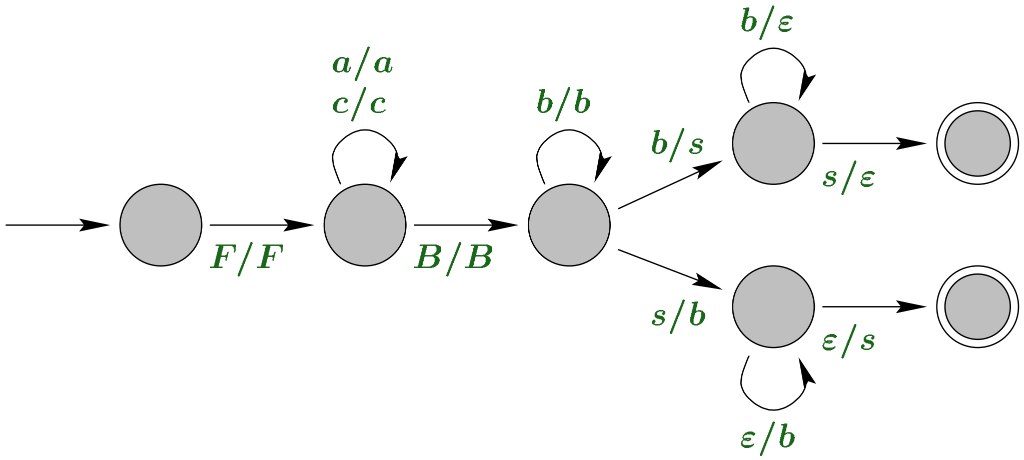
Un exemple avec des arbres

```
      void Sum (tree *p) {
a       if (···) Sum (p->l);
B/b     for (q = p; q; q = q->l)
s         *p = *p + *q;
c       if (···) Sum (p->r);
      }
      main () {
F       Sum (root);
      }
```



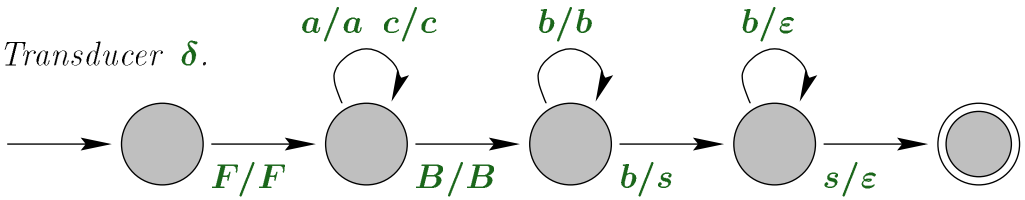
Référence *p

- **Transducteur du conflit**

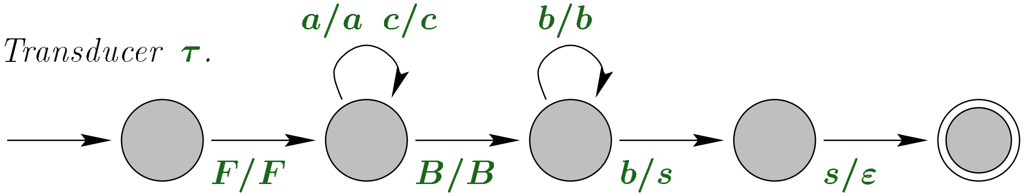


- **Dépendances et définitions visibles**

Transducer δ.

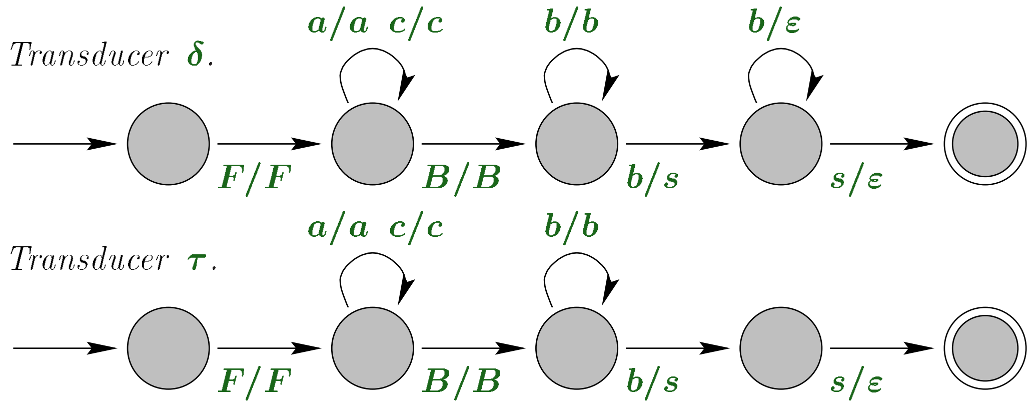


Transducer τ.



Référence *q

- **Transducteur des dépendances**



- **Impossible de raffiner davantage**

  Approximation des définitions visibles par les dépendances vraies

Conclusion

- **Les langages formels sont largement utilisés pour l'analyse statique de programmes récursifs**
- **Mais notre problème et notre modèle sont nouveaux :**

  On calcule des propriétés du flot de données pour chaque instance d'instruction (opération)
- **On dispose d'un algorithme pour l'analyse de définitions visibles de programmes récursifs**
  - Les résultats sont bons sur une classe restreinte
  - En général, on obtient une approximation conservative
  - Des analyses classiques sont parfois utiles pour affiner les résultats (exacts sur `FillSlots`, `Queens`, et d'autres)
- **Utilisable pour expanser et paralléliser des programmes récursifs**