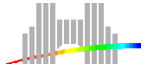
Master Informatique M2
Analyse Statique

Paul Feautrier

ENS de Lyon
Paul.Feautrier@ens-lyon.fr

9 octobre 2006



Pourquoi l'Analyse Statique

- Tout ce que l'on peut apprendre sur le comportement du programme sans l'exécuter.
- Une analyse statique ne peut être qu'incomplète (sauf programme trivial).
- Analyse dynamique : pendant l'exécution.
- Avantages de l'Analyse Statique :
  - Les résultats valent pour un ensemble de jeux de données (parfois pour tous).
  - Amortissement du coût sur de multiples exécutions.
  - Les résultats sont disponibles pour améliorer la compilation.
  - Résultats positifs : une boucle est parallèle, une variable est constante.

Analyse dynamique

- Observer ou instrumenter une exécution du programme, par exemple en utilisant un simulateur.
- Permet de mesurer d'évaluer des phénomènes inaccessibles sur la machine réelle (par exemple, les défauts de cache).
- Nécessite un jeu de données ; les résultats obtenus ne sont en principe valables que pour ce jeu de données ; mais il est parfois possible d'extrapoler.
- Informations quantitatives (e.g. fréquence d'appel d'une procédure).
- Réultats négatifs : une boucle n'est pas parallèle, une variable n'est pas constante.
- De ce point de vue, complémentaire de l'analyse statique.

Qu'apporte l'analyse statique ?

- Des propriétés des variables, valables soit pendant toute l'exécution du programme, soit au moment de l'exécution d'une instruction
  - Etre constante
  - Etre d'un certain signe
  - Appartenir à un certain intervalle
  - Etre dans une certaine relation avec d'autres variables
- Une représentation du flot des données
  - *use-def chains*
  - variables vivantes
  - expressions disponibles
  - forme SSA
  - dépendences.

Pour quoi faire ?

- Connaître des propriétés des variables permet de simplifier le code (*constant folding*, élimination du code mort, élimination des contrôles d'indices)
- De même, connaître les expressions disponibles permet de supprimer les calculs redondants.
- Savoir de quoi dépend un calcul permet de réordonner les opérations du programme, ce qui est la clef aussi bien de l'amélioration de la localité que de la recherche du parallèlisme.
- Très souvent, une première analyse permet d'améliorer l'analyse suivante. Par exemple, la connaissance des relations entre indices améliore le calcul des dépendences.

ANALYSE DU FLOT DES DONNEES
INTERPRETATION ABSTRAITE

Exécution symbolique, I

Que fait le code ci-dessous :

	x_0	y_0
`x = x - y;`	$x_0 - y_0$	y_0
`y = y + x;`	$x_0 - y_0$	$y_0 + x_0 - y_0 = x_0$
`x = y - x;`	$x_0 - (x_0 - y_0) = y_0$	x_0

Exécution symbolique, II

Que s'est il passé ?

- On a exécuté le programme initial ...
- ... mais en changeant le type des données, qui étaient des nombres et sont devenues des expressions algébriques.
- On conserve la valeur symbolique courante de chaque variable.
- On substitue dans le membre droit de chaque opération les valeurs courantes, on simplifie si l'on peut, et on affecte au membre droit.

Exécution symbolique, III

Que se passerait-il si le programme contenait un test ou une boucle ?

	x_0
`if(b)`	
`x = a;`	a_0
`else`	
`x = b;`	b_0
	?

	i_0
`while(i<n)`	?
`i = i+1;`	$i_0 + 1$
...	?

Au delà de l'exécution symbolique

Variables d'induction.

- On trouve une variable x dont la valeur finale est $x_0 + c$ où c est une constante.
- Valeur de x au début de l'itération k : $X + k.c$ (X valeur avant le début de la boucle, k compte-tour).
- On peut substituer cette valeur dans le corps de la boucle, ce qui rend plus précis le calcul des dépendances.

Réductions.

- Si le résultat de l'exécution symbolique est $x_0 + c_k$, et si on peut sortir le calcul de x de la boucle :

$$x = \sum_{k=0}^{n} c_k,$$

  ce qui peut être évalué en parallèle.
- Il suffit que l'opérateur de la réduction soit associatif.

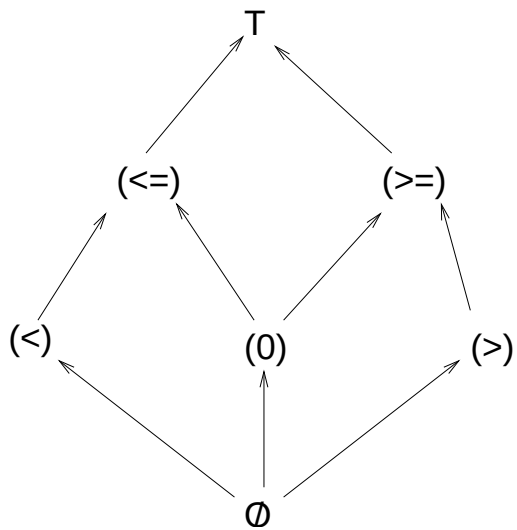
Conclusions provisoires

- On peut obtenir des informations intéressantes en remplaçant les données concrètes du programme par des données abstraites et en interprétant.
- L'espace abstrait doit être une représentation synthétique de l'espace concret.
- Traitement des tests : suivre les deux branches, puis faire la synthèse des renseignements obtenus.
- Traitement des boucles : aucun calcul ne doit pouvoir se prolonger indéfiniment, car on ne connait pas le nombre d'itérations.

Le Signe d'une Variable Scalaire

Le signe d'une variable est un information intéressante : il permet par exemple de supprimer une vérification avant d'extraire une racine carrée. L'espace abstrait est composé de sous-ensembles de $\mathbb{R}$:

$$
\begin{aligned}
\top &= \mathbb{R} \\
(\geq) &= \{x \mid x \geq 0\} \\
(\leq) &= \{x \mid x \leq 0\} \\
(>) &= \{x \mid x > 0\} \\
(0) &= \{0\} \\
(<) &= \{x \mid x < 0\} \\
\bot &= \emptyset
\end{aligned}
$$



La relation d'ordre du treillis, notée $\sqsubset$, signifie "est plus précis que" : $(0) \sqsubset (\leq)$: être nul est plus précis qu'être non-négatif.

Règles de calcul, I

On peut construire des tables de Pythagore pour les calculs sur ces objets. Par exemple, la somme d'un nombre positif et d'un nombre non-négatif est positive, soit :

$$(>=) + (>) = (>).$$

Il y a des cas où le signe du résultat ne peut pas être décidé :

$$(>=) + (<=) = \top.$$

$+$	$\bot$	$(<)$	$(<=)$	(0)	$(>=)$	$(>)$	$\top$
$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$
$(<)$	$\bot$	$(<)$	$(<=)$	$(<=)$	$\top$	$\top$	$\top$
$(<=)$	$\bot$	$(<)$	$(<=)$	$(<=)$	$\top$	$\top$	$\top$
(0)	$\bot$	$(<)$	$(<=)$	(0)	$(>=)$	$(>)$	$\top$
$(>=)$	$\bot$	$\top$	$\top$	$(>=)$	$(>=)$	$(>)$	$\top$
$(>)$	$\bot$	$\top$	$\top$	$(>)$	$(>)$	$(>)$	$\top$
$\top$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$

Régles de calcul, II

- On constate que l'addition reste associative et commutative.
- On peut construire des tables analogues pour les autres opérations (multiplication, division, changement de signe).
- On constate que ces opération sont monotones : plus les arguments sont précis, plus les résultats sont précis.
- Lors de l'interprétation, on peut donc itérer une boucle : on finira toujours par atteindre un point fixe.

Un peu de théorie, d'après Cousot

- C ensemble des données concrètes.
- A ensemble des abstractions, treillis complet avec $\bot$:
  - Il existe une opération borne supérieure $\wedge$ associative, commutative, idempotente ($x \wedge x = x$).
  - $\bot \wedge x = x$.
  - Si on pose $x \sqsubset y$ ssi $x \wedge y = y$, alors $\sqsubset$ est une relation d'ordre partiel. En particulier, $\forall x, \bot \sqsubset x$.
  - Toute chaîne ascendante $x_1 \sqsubset \ldots \sqsubset x_k \ldots$ a une borne supérieure notée $\bigwedge_{k \geq 1} x_k \in A$.

Correspondance de Galois

On postule deux fonction, $\alpha :: \wp(C) \rightarrow A$, l'abstraction, et $\gamma :: A \rightarrow \wp(C)$, la concrétisation.
Les fonction α et γ doivent vérifier les contraintes de cohérence :

$$\begin{aligned} a &= \alpha(\gamma(a)), \\ X &\subseteq \gamma(\alpha(X)) \end{aligned}$$

Opérations

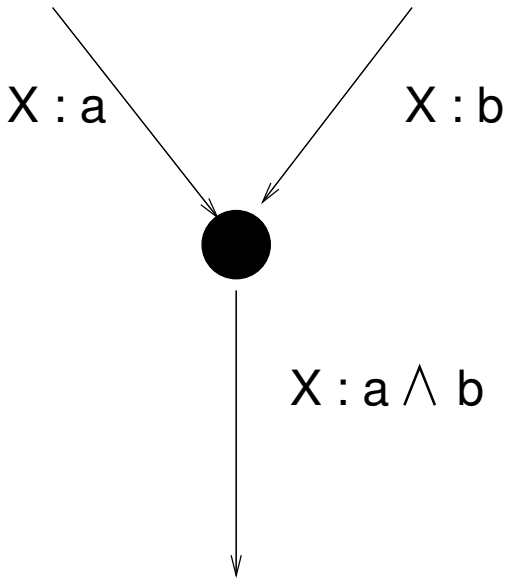
Soit $+$ une opération binaire concrète. On lui associe une opération abstraite $\oplus :: A \times A \longrightarrow A$ par la règle :

$$a \oplus b = \alpha(\gamma(a) * \gamma(b)).$$

Les fonctions ainsi définies sont monotones :

$$a \sqsubset b \Rightarrow a \oplus c \sqsubset b \oplus c...$$

Jointure



- Permet de traiter les tests et les boucles.
- Propriété de la borne supérieure :

$$a \wedge b = \alpha(\gamma(a) \cup \gamma(b)).$$

Treillis des signes, le retour

- La table ci-dessus représente en fait la fonction γ.
- La fonction α est donnée par les règles suivantes :

$$
\begin{aligned}
\alpha X \quad = \quad & \textbf{if } X = \emptyset \textbf{ then } \bot \\
& \textbf{else if } X = \{0\} \textbf{ then } (0) \\
& \textbf{else if } \forall x \in X : x < 0 \textbf{ then } (<) \\
& \textbf{else if } \forall x \in X : x \leq 0 \textbf{ then } (<=) \\
& \textbf{else if } \forall x \in X : x > 0 \textbf{ then } (>) \\
& \textbf{else if } \forall x \in X : x \geq 0 \textbf{ then } (>=) \\
& \textbf{else } \top
\end{aligned}
$$

- On vérifie par énumération les contraintes de cohérence.

Interprétation abstraite

- Remplacer les données par leur abstraction et les opérations $+$ par l'opération abstraite $\oplus$ correspondante.
- Aprés un test, si $\mathtt{x}$ a la valeur a_g et a_d à droite, lui donner la valeur $a_g \wedge a_d$.
- On montre que pendant l'itération d'une boucle, les valeurs successives d'une variable $\mathtt{x}$ forment une chaîne. On la remplace par sa borne supérieure. Cette opération n'est effective que si le treillis est de hauteur finie (c'est le cas pour le treillis des signes).

Analyse d'intervalle, I/III

- Trouver un intervalle encadrant les variables du programme.
- Intérêt : multiple.
  - Si la variable est un indice de tableau, permet de se dispenser du test de débordement, ou au contraire de présumer une erreur.
  - Permet de détecter des dépassements de capacité.
  - Permet de remplacer une variable en virgule flottante par une variable en virgule fixe, et de la loger dans le minimum de place.
  - Permet de simplifier le programme en prédisant le résultat de test.

Analyse d'Intervalles, II/III

- L'ensemble abstrait : intervalles $[a, b]$, $a \leq b$, ordonnés par inclusion.
- On prend pour borne supérieure :

$$[a, b] \wedge [c, d] = [\min(a, c), \max(b, d)].$$

- Les fonctions d'abstraction et de concrétisation sont :

$$\alpha(X) = [\min X, \max X], \quad \gamma([a, b]) = \{x \mid a \leq x \leq b\}$$

- Régles de calcul : pour calculer $U \oplus V$, former $W = \{u + v \mid u \in V, v \in V\}$, puis appliquer α.
- Le calcul est trivial pour l'addition. Multiplication :
  - Si les 4 nombres a, b, c, d sont positifs :

$$[a, b] \times [c, d] = [ac, bd].$$

    On trouve des formules du même genre chaque fois que les deux intervalles ne contiennent pas 0.
  - Sinon, découper chaque intervalle en 2, effectuer les 4 produits et utiliser la formule de la borne supérieure.

Analyse d'intervalles, III/III

- Le problème : l'espace des valeurs abstraites n'est pas de hauteur finie :

$$[-1,1] < [-2,2] < [-3,3] < \dots.$$

- Pour accélérer la convergence, on est obligé d'utiliser un opérateur d'élargissement, (*widening*) qui doit avoir la propriété :

$$u \wedge v < \nabla(u,v).$$

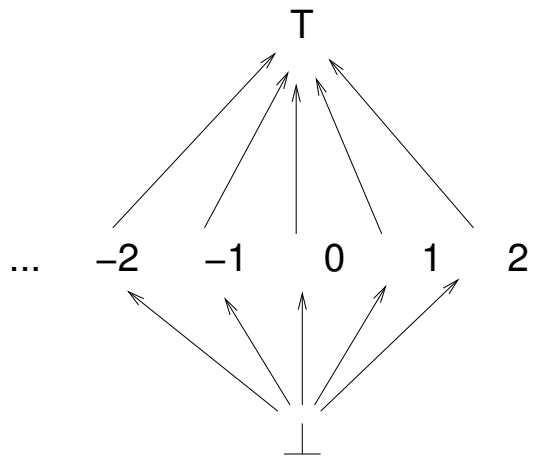
- Par exemple :

$$\begin{aligned}
c < a, d > b &\Rightarrow \nabla([a,b],[c,d]) =]-\infty,+\infty[\\
c = a, d > b &\Rightarrow \nabla([a,b],[c,d]) = [a,+\infty[\\
c < a, d = b &\Rightarrow \nabla([a,b],[c,d]) =]-\infty,b] \\
c = a, d = b &\Rightarrow \nabla([a,b],[c,d]) = [a,b]
\end{aligned}$$

- De cette façon, on parvient toujours à encadrer une variable, mais parfois c'est sans intérêt ($-\infty < x < +\infty$).

Autres analyses

- Propagation des constantes : treillis ci-contre. Permet d'effectuer certains calculs à la compilation et de simplifier le programme.
- Expressions disponibles : expressions qui ont été calculées et dont aucunes des données n'ont été modifiées. Elimination des calculs redondants.
- *use-def chains* permet de relier les lectures à leur source.



Calcul des *use-def chains*

- Soit `x` une variable et T une instruction. Une autre instruction S est une *source* possible pour `x` en T ssi :
  - S est une affectation à `x`.
  - Il existe un chemin d'exécution partant du début du programme, passant par S puis T et tel qu'il n'y ait pas d'affectation à `x` entre S et T.
- Pour calculer les sources dans le cadre de l'interprétation abstraite, on augmente le programme de la façon suivante :
  - A toute variable `x` on associe une variable auxilliaire `@x`.
  - Si S affecte une valeur à `x`, on écrit S dans `@X`.
- Quand on passe par T, il est clair que `@x` contient la source courante de `x`. Il suffit donc de calculer par interprétation abstraite l'ensemble des valeurs possibles de `@x`.

Treillis et règles de calcul

- Le treillis des valeurs abstraites est l'ensemble des parties de l'ensemble des instructions, ordonné par inclusion.
- L'opérateur de fusion est l'union ensembliste.
- Si S est une affectation à `x`, après son exécution, on a `@x` $= S$.
- Sinon, `@x` n'est pas modifié.
- En un point de jonction, on fait l'union des valeurs de `@x` sur tous les arcs entrants.
- On itére jusqu'à obtenir un point fixe. La convergence est garantie parce que e treillis est de hauteur finie.

Recherche de relations entre variables

- Méthode de Cousot-Halbwachs.
- On recherche des contraintes affines portant sur les variables scalaires entières du programme. Exemple :

$$0 \leq 2i + j \leq 100, j = 3k + 7.$$

- Généralise l'analyse d'intervalle et la recherche de variables inductives.
- Utilité :
  - Vérification des accès aux tableaux et suppression des tests de bornes.
  - Pour les langages à tableaux dynamiques (Matlab ...), inférence des tailles de tableaux.
  - Démonstration de propriétés.
  - Calcul des dépendances sur tableaux (voir plus loin).

Digression : polyèdres et polytopes

- Un polyèdre de $\mathbb{R}^n$ est l'ensemble des vecteurs à n dimensions qui satisfont à un système d'inégalités affines :

$$P = \{x \mid Ax + b \geq 0\},$$

où A est une matrice $m \times n$ et b un vecteur de dimension m.

- Un polyèdre est un objet convexe :

$$x, y \in P \Rightarrow \forall 0 \leq \lambda \leq 1 : \lambda x + (1 - \lambda)y \in P.$$

- Un polytope est un polyèdre borné.

Thèorème de Minkowski

- L'ensemble :

$$P = \{\sum \lambda_i x_i + \sum \mu_j y_j + \sum \nu_k z_k \mid \lambda_i \geq 0, \sum \lambda_i = 1, \mu_k \geq 0\}$$

  est un polyèdre
- Les x_i sont les sommets, les y_i sont les rayons et les z_i sont les lignes de P (points extrèmes de P)
- P est un polytope s'il n'a ni rayon ni ligne
- Tout polyèdre peut se mettre de façon unique sous la forme ci-dessus, que l'on note :

$$P = \mathrm{Hull}(X, Y, Z)$$

Algorithmique sur les polyèdres, I

- Si $P = \{x \mid Ax + b \geq 0\}$ et $P' = \{x \mid A'x + b' \geq 0\}$ sont deux polyèdres de même dimension, leur intersection est le polyèdre :

$$P \cap P' = \{x \mid Ax + b \geq 0, A'x + b' \geq 0\}$$

- La construction de l'intersection est triviale, mais il n'est pas évident de décider si elle est vide ou non.
- La représentation obtenue n'est pas nécessairement la plus simple.

Algorithmique sur les polyèdres, II

- L'union de deux polyèdres n'est pas nécessairement un polyèdre, parce que l'union n'est n'est pas nécessairement convexe.
- Dans l'espace des polyèdres, la borne supérieure est la *coque convexe* de l'union (*union convexe*)
- si $P = \mathrm{Hull}(X, Y, Z)$ et si $P' = \mathrm{Hull}(X', Y', Z')$, alors

$$P \wedge P' = \mathrm{Hull}(X \cup X', Y \cup Y', Z \cup Z')$$

- La construction est triviale mais la représentation n'est pas nécessairement minimale.

Algorithmique sur les polyèdres, III

- Méthode de la double représentation : on conserve à la fois la liste des contraintes et celle des sommets, rayons et lignes.
- Pour chaque opération, on utilise la représentation la plus commode.
- Mais il faut ensuite calculer l'autre représentation.
- L'algorithme qui permet de passer de l'une à l'autre est l'algorithme de Chernikova.

Algorithmique sur les polyèdres, IV
Construction de la représentation de Minkowski

En supposant pour simplifier que le polyèdre $Ax + b \geq 0$ n'a que des sommets :

- On montre qua pour chaque sommet x il existe une sous matrice A' de A inversible et un sous vecteur b' de b tels que $A'x + b' = 0$.
- On énumère les C_m^n choix possibles, on calcule le x correxpondant par la méthode de Gauss, et on teste si $Ax + b \geq 0$.
- La complexité est donc $O(C_m^n n^3)$.

Il existe une version optimisée de cet algorithme, l'algorithme de Chernikova. Il peut être utilisé indifféremment pour passer des contraintes aux sommets ou l'inverse. Il existe une implémentation efficace de cet algorithme, la *polylib*.

Algorithmique sur les polyèdres, V
Projection

La projection de P selon la première coordonnée est définie par :

$$Q = \{y \mid \exists x_1 : x_1.y \in P\}.$$

La projection d'un polyèdre est un polyèdre.
Il existe plusieurs algorithmes de projection :

- On peut éliminer x_1 par combinaisons linéaires positives des contraintes de P (algorithme de Fourier-Motzkin).
- On peut projeter les sommets, rayons et lignes de P, puis reconstituer un système de contraintes à l'aide de l'algorithme de Cernikova.

Treillis de Cousot-Halbwachs

- Soit n le nombre de variables "intéressantes". Les éléments du treillis sont les polyèdres à n dimensions, avec l'union convexe comme borne supérieure
- Dire que le programme est dans l'état P, c'est dire que le vecteur $x = (x_1, \ldots, x_n) \in P$, où les x_i sont les valeurs courantes des variables intéressantes.
- On notera que ce treillis n'est pas de hauteur finie.

Affectation

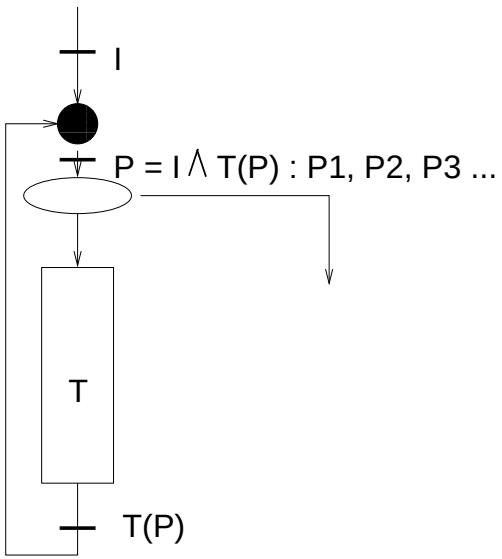
- On peut représenter une affectation par une transformation appliquée au vecteur x (utiliser par exemple l'exécution symbolique).
- L'état du programme après l'affectation s'obtient en appliquant cette transformation à l'état antérieur.
- Si la transformation est est linéaire, on construit l'état suivant en appliquant la transformation aux points extrèmes de P, puis en reconstituant les contraintes par l'algorithme de Chernikova.
- Si la transformation est non linéaire, on élimine par projection les variables affectées.

Tests

- Si le prédicat du test est non linéaire, on ne peut pas l'exploiter.
- Sinon, on l'adjoint aux contraintes de P sur la branche **true**, et on adjoint son opposé sur la branche **false**.
- On analyse les deux branches, sauf si l'algorithme de Chernikova montre que l'un des polyèdres est vide.
- On construit l'union convexe des deux branches au point de jonction.

Boucles

Comme le treillis n'est pas de hauteur finie, un opérateur d'élargissement est nécessaire pour assurer la convergence.



- Dans toute boucle il y a un point de jonction, que l'on peut traiter comme pour un test ...
- ... mais, on enregistre la suite des valeurs du polyèdre P au cours des itérations,
- et on applique un opérateur d'élargissement : on calcule P_k et on le remplace par $P_{k-1} \nabla P_k$.

Opérateur d'élargissement

Pour calculer $P \nabla Q$:

- Représenter P par un système de contraintes.
- Représenter Q par ses points extrèmes.
- Ne conserver que les contraintes de P qui sont satisfaites par tous les points extrèmes de Q.

Un exemple

```
k=7;
i=0;
while(...){
    k += 3;
    i += 1;
}
```

- Une façon sophistiquée de faire de l'analyse de variables inductives.
- Après initialisation, $I = (0, 7)$.
- Aprés un tour, $P_1 \mathrm{Hull}((0, 7), (1, 10))$.
- Après deux tours, $P_2 = \mathrm{Hull}(0, 7), (2, 13)$.
- On élargit : les contraintes de P_1 sont $k = 3i + 7$ et $0 \leq i \leq 1$.
- Il ne reste que $k = 3i + 7$ et $0 \leq i$, et on a trouvé le point fixe.

En conclusion ...

- Méthode très puissante, qui étend l'analyse de signe et l'analyse d'intervalle.
- La méthode se combine bien avec l'interprétation symbolique, qui permet de grouper tous les traitements d'un *bloc de base*.
- Mais complexité élevée, parce que un polyèdre de dimension n peut avoir $O(2^n)$ sommets (le cube).
- Retarder le plus possible l'utilisation de l'algorithme de Chernikova.
- Exploiter le plus possible les informations "gratuites" : bornes des boucles, prédicats des tests, etc.

Analyse des pointeurs, pourquoi ?

Le déréférencement d'un pointeur en écriture bloque la plupart des optimisations :

```
  for(i=0; i<n; i++){
     x = 0;
     *p = i;
     ...
S:   ... = x ...;
  }
```

Impossible de remplacer `x` par 0 dans S, sauf si on a des informations sur `p`.

Aliasing

- Il y a *aliasing* lorsqu'une même cellule de mémoire peut être accédée par plusieurs constructions syntaxiquement différentes.
- Le cas le plus simple d'aliasing : les tableaux. `A[1]` et `A[i]` sont des alias si $i = 1$. On verra plus loin comment traiter ce cas particulier.
- Les possibilités d'aliasing dépendent fortement du langage de programmation.

Revue de langages

- FORTRAN :
  - EQUIVALENCE et COMMON
  - arguments des procédures
- C/C++ :
  - pointeurs
  - arithmétique sur les pointeurs
  - opérateur de calcul d'adresse &
  - unions
- Java : références

C'est en Java que le problème est le plus facile à résoudre.

Méthodes d'analyse

- Méthodes *memory-based* : on cherche à identifier les objets du programme et a approximer la relation d'accessibilité : $O \in \mathrm{reach}(p)$ ssi il existe un chemin de `p` jusqu'a `0`
- Méthodes *memory-less* : on cherche à approximer directement la relation d'aliasing
- Méthodes graphiques : on cherche à reconstituer la cartographie du système d'objet.

Esquisse d'une analyse *memory based*

- Pour simplifier, on ignore les tableaux.
- Objets : on identifie tous les objets créés par la même instruction `p = new Class`
- Le treillis : les sous-ensembles de l'ensemble des objets, y compris l'ensemble vide, ordonnés par inclusion, avec l'union comme jointure.
- Le nombre des objets est fini, donc le treillis est de hauteur finie.
- Etat abstrait du programme en chaque point S on associe à chaque pointeur `p` un ensemble d'objets abstraits noté $\mathrm{reach}(p, S)$.

Opérations

En modifiant légèrement le programme si nécessaire, on peut se borner aux opérations suivantes :

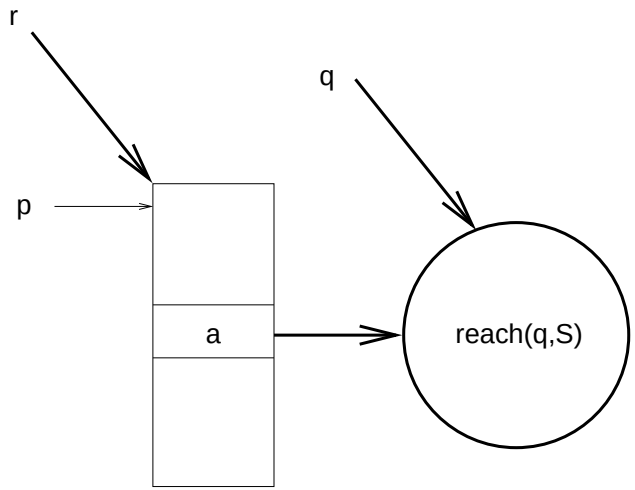
```
1:  p = null;          2:  p = new Class;
3:  p = q;             4:  p = q.a;
5:  p.a = q;
```

On note T l'instruction qui précède l'instruction S. Dans les cas 1 à 4, seul $\mathrm{reach}(p, S)$ est modifié :

$$\begin{aligned} 1 : \mathrm{reach}(p, S) &= \emptyset, \\ 2 : \mathrm{reach}(p, S) &= \{\mathrm{new}_S\}, \\ 3 : \mathrm{reach}(p, S) &= \mathrm{reach}(q, T), \\ 4 : \mathrm{reach}(p, S) &= \mathrm{reach}(q, T) \cap \mathcal{T}(a). \end{aligned}$$

$\mathcal{T}(a)$ est l'ensemble des objets du type de a.

Affectation sur le tas



$$\forall r : \text{reach}(r, T) \quad \cap \quad \text{reach}(p, T) \neq \emptyset$$
$$\Rightarrow \quad \text{reach}(r, S) = \text{reach}(r, T) \cup \text{reach}(q, T).$$

Notez les cas particuliers `r = p` et `q = null` (aucune modification.

DEPENDANCES
METHODES GEOMETRIQUES

Dépendances

- On sait par expérience que l'ordre des instructions d'un programme est (parfois) important pour le résultat.
- Problème : trouver une méthode simple pour décider si :

$$\mathtt{S1}; \mathtt{S2} \approx \mathtt{S2}; \mathtt{S1}$$

où S1 et S2 sont des instructions (de niveau arbitraire).

- Intérêt :
  - Deux instructions qui commutent peuvent être exécutées en parallèle.
  - Echanger deux instructions (et plus) peut améliorer la localité.
  - Echanger deux instructions peut autoriser d'autres optimisations.

Conditions de Bernstein / I

```
S1 : y := f(x);
S2 : u := g(v);
```

- Si $y \equiv u$, alors $S1; S2 \approx u = g(v_0)$, $S2; S1 \approx u = f(x_0)$ et *il n'y pas de raison* pour que ces quantités soient égales. Il y a dépendance.
- Si $y \not\equiv u$ mais $y = v$, $S1; S2 \approx y = f(x_0), u = g(f(x_0))$, $S2; S1 \approx u = g(y_0), y = f(x_0)$, dépendance.
- Situation symmétrique pour $u = x$.

Condition de Bernstein / II

- Condition suffisante d'indépendance :

Theorem

Pour que S1 *et* S2 *soient indépendantes, il suffit que* $y \neq u, y \neq v, u \neq x$.

- Plus généralement, $\mathcal{R}(S)$ ensemble des variables lues, $\mathcal{M}(S)$ ensemble des variables modifiées par S.

Theorem

Pour que S1 *et* S2 *soient indépendantes, il suffit que :*

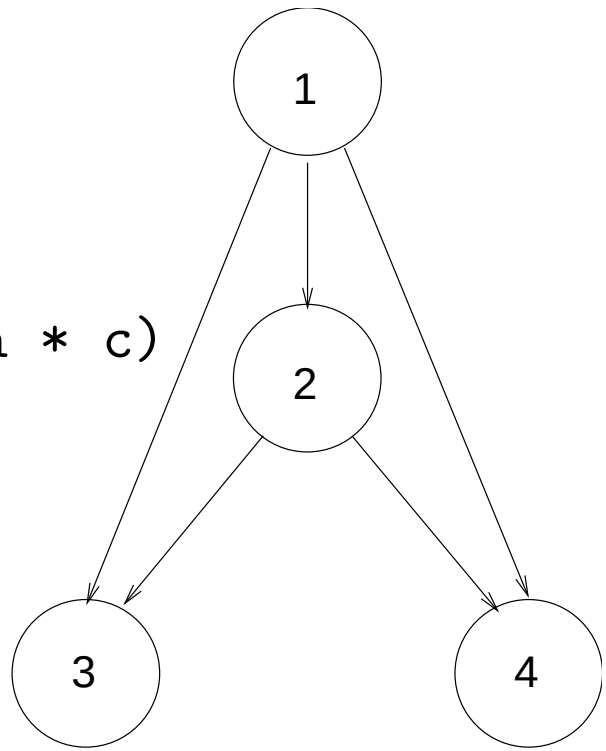
$$\mathcal{M}(S1) \cap \mathcal{M}(S2) = \emptyset,\ \mathcal{M}(S1) \cap \mathcal{R}(S2) = \emptyset,\ \mathcal{R}(S1) \cap \mathcal{M}(S2) = \emptyset.$$

- Ces conditions ne sont pas nécessaires :

```
S1 : x := x + 1;
S2 : x := x + 3;
```

Un Exemple

```
1  read a, b, c
2  delta = sqrt(b*b - 4.0 * a * c)
3  x1 = (-b + delta)/2.0/a
4  x2 = (-b - delta)/2.0/a
```



◂ <par

Propriété fondamentale, I

Soit un programme composé de N opérations $\{S_1, \ldots, \S_N\}$.

- On suppose que l'on a déroulé les boucles et résolu les tests.
- Normalement le programme est exécuté suivant l'ordre séquentiel :

$$S_i <_{\text{seq}} S_j \equiv i < j.$$

- $<_{\text{seq}}$ est un ordre *total*.

Propriété fondamentale, II

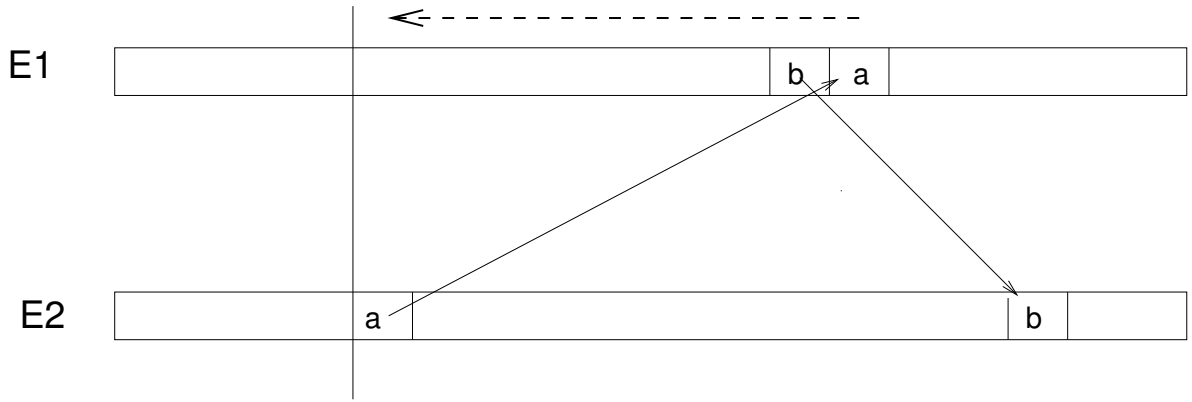
- On note δ la relation de dépendance. $S_i \, \delta \, S_j$ signifie que l'on ne peut pas permuter S_i et S_j, par exemple parce que les conditions de Bernstein sont violées.
- On remarque que δ est symmétrique.
- On considère l'ordre $<_{\text{par}}$, fermeture transitive de l'intersection $<_{\text{seq}} \cap \delta$.
- $<_{\text{par}}$ est en général un ordre partiel, voir l'exemple ▸ DG
- Exécuter le programme selon l'ordre $<_{\text{par}}$, c'est l'exécuter selon une extension totale de l'ordre partiel.

Propriété fondamentale, III

Theorem

Toutes les exécutions possibles selon $<_{\text{par}}$ *conduisent au même résultat.*

Démonstration.



On dit que l'ordre $<_{\text{par}}$ *satisfait* toutes les dépendances. □

Proprité fondamentale, IV

- La preuve ci-dessus utilise implicitement le fait que le programme se termine.
- Dans le cas d'un programme qui ne se termine pas, on ne peut plus parler de résultat.
- Il faut considérer l'histoire de chaque variable.

Définitions

- Trace : l'enregistrement de tout ce qui se passe lors d'une exécution. Tuples $<$ opération, variable affectée, valeur affectée, variables lues $>$.
- Histoire d'une variable x : une trace réduite aux opérations qui affectent x.
- Source de x au point k d'une trace t : l'opération antérieure la plus proche de k qui affecte x :

$$\sigma(x, t, k) = \min\{i \mid i < k, x \in M(t_i)\}.$$

Propriété fondamentale, V

Lemma

Soit t une trace respectant l'ordre $<_{\text{par}}$ et x une variable. Dans l'histoire de v selon t, les opérations apparaissent dans le même ordre que dans le programme séquentiel original.

Démonstration.

En effet, toutes les opérations qui affectent x sont en dépendance, donc sont ordonnées par $<_{\text{par}}$ dans le même ordre que dans $<_{\text{seq}}$. □

Propriété fondamentale, VI

Lemma

Dans les mêmes conditions, la fonction source pour x dans t est le même que dans le programme séquentiel.

Démonstration.

Il suffit d'observer que le calcul d'une source de x ne fait intervenir que les opérations de l'histoire de x et leur ordre. □

Propriété fondamentale, VII

Theorem

Une variable quelconque a la même histoire dans toutes les exécutions selon l'ordre $<_{\text{par}}$.

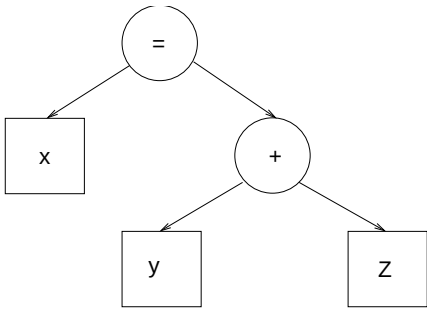
Démonstration.

Soit deux traces t_1 et t_2, soit x une variable ayant pour histoires h_1 et h_2, et supposons que ces histoires divergent en position k. Il correspond à k deux positions k_1 et k_2 dans t_1 et t_2 et nous dirons que la divergence se manifeste à l'instant $\max k_1, k_2$. Comme les opérations sont déterministes et ont les mêmes sources dans les deux exécutions, il faut qu'il y ait une divergence pour l'une des sources aux instants respectifs $k_1' < k_1$ et $k_2' < k_2$. Il en résulte que $\max(k_1', k_2') < \max(k_1, k_2)$. On forme donc ainsi une suite de divergences qui se manifestent de plus en plus tôt. Ceci ne peut se poursuivre indéfiniment, puisque une trace a un début. On a donc trouvé une contradiction.

CALCUL DES DEPENDANCES

Scalaires ; direction de dépendance

On collecte les variables lues et modifiées par visite de l'AST.
Attention cependant aux instructions complexes de C.



- On calcule ensuite les dépendances par intersection d'ensembles finis donnés en extension.
- Les dépendances sont orientées dans l'ordre d'exécution séquentielle.

Classification des dépendances.

PC	CP	PP
flow	anti	output
RAW	WAR	WAW

Problème du calcul d'adresse

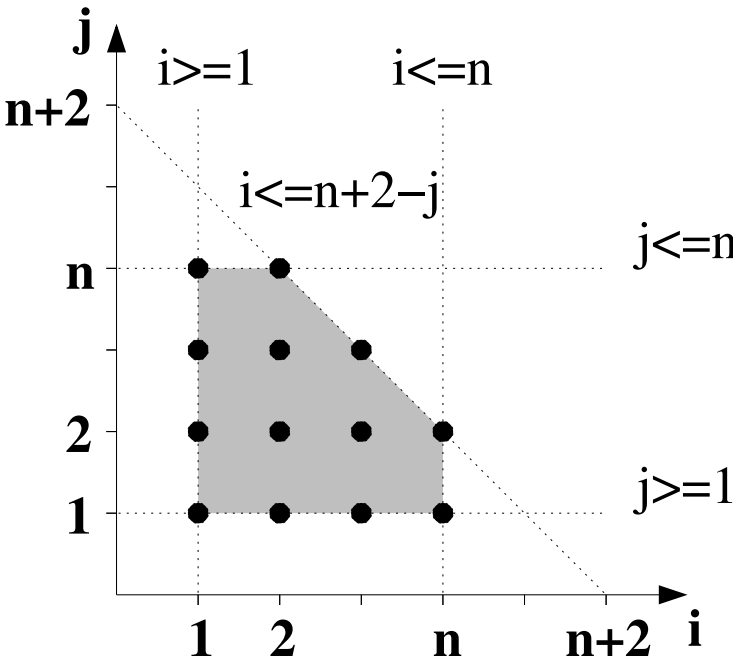
- Que faire quand l'adresse d'une variable est calculée ?
  Exemples : `a[i]` ou `*p`.
- Texte identique, mais adresses différentes :

```
a[i] = ...;
i++;                    Il n'y a pas de dépendance.
a[i] : ...;
```

- l'inverse :

```
a[i] = ...;
j = i;                  Il y a dépendance.
a[j] : ...;
```

Nid de boucles




```
for(i=1 ;i<=n; i++)
  for(j=1; j<=n; j++)
    if(i+j <= 2*n-2)
       S;
```

- Contexte : une instruction est en général partie d'une structure répétitive (nid de boucles ou récursion).
- Chaque instruction engendre autant d'opérations que d'itérations.
- Nommage de l'opération par son vecteur d'itération.

Ordre d'exécution

- Indispensable pour orienter les dépendances.
- Dans un nid de boucle parfait, les opérations sont exécutées dans l'ordre lexicographique :

$$(i_1, \ldots, i_n) \prec (j_1, \ldots, j_n) \equiv i_1 < j_1 \vee (i_1 = j_1 \ \& \ i_2 < j_2) \vee \ldots$$

- Instructions différentes : tenir compte de l'ordre textuel. Ajouter :

$$i_1 = i_2 \ \& \ \ldots \ \& \ i_n = j_n \ \& \ S <_{\text{text}} T.$$

- Nid imparfait : ne tenir compte que des boucles communes.

Interprétation des conditions de Bernstein

- Dans ce cadre, les ensembles lus et modifiés sont fonctions du vecteur d'itération. On écrit $\mathcal{R}(T, \vec{i})$, etc.
- Pour qu'une condition de Bernstein soient violée, il faut que les ensembles lues ou modifiées aient une cellule commune. Si le compilateur est correct :
  - Les tableaux ne se recouvrent pas, donc il y a un tableau commun aux deux listes, A.
  - Les bornes d'indices sont respectées, donc les indices sont égaux.
- Si les deux références à A sont $A[f(\vec{i})]$ et $A[g(\vec{j})]$, il n'y a dépendance que si (équation aux indices) :

$$f(\vec{i}) = g(\vec{j}).$$

Méthodes ad hoc / I

Test de Banerjee

```
for(i=0; i<n; i++)
    a[i] = i;
```

- Deux itérations $i < j$.
- Dépendance seulement si $i = j$.
- Soit f(i) = j-i. On cherche à résoudre $f(i) = 0$ pour $i \in [0, j-1]$.
- $f(0) = j > 0$ et $f(j-1) = 1 > 0$ donc f monotone croissante et continue n'a pas de racine.

Méthodes ad hoc / II

Test du pgcd

```
for(i=0; i<n; i++)
    a[2*i] = i;
    a[2*i+1] = i+1;
```

- Dépendance seulement si $2i = 2j + 1$.
- Impossible : le pgcd des coefficients doit diviser le terme constant.

Forme générale du problème

operation domaine	$(S, \vec{i}) : A[f(\vec{i})]$ $\vec{i} \in D_S$		$(T, \vec{j}) : A[g(\vec{j})]$ $\vec{j} \in D_T$
indices		$f(\vec{i}) = g(\vec{j})$	
ordre	 $i_1 < j_1$	$(S, \vec{i}) \prec (T, \vec{j})$ $i_1 = j_1, i_2 < j_2$	 $i_1 = j_1, i_2 = j_2$
profondeur	0	1	2

La connaissance de la profondeur de dépendance est indispensable pour l'algorithme de Allen et Kennedy, voir plus tard.

Un exemple / I

```
    for(i=1; i<=n; i++){
1:      x = a[i][i];
        for(k=1; k<i; k++)
2:          x = x - a[i][k]*a[i][k];
3:      p[i] = 1.0/sqrt(x);
        for(j=i+1; j<=n; j++){
4:          x = a[i][j];
            for(k=1; k<i; k++)
5:              x = x - a[j][k]*a[i][k];
6:          a[j][i] = x * p[i];
            }
```

Un exemple / II

- Y -a-t-il une dépendance entre 5 et 6 portant sur le tableau a ?

$(5, i, j, k) : A[j][k]]$		$(6, i', j') : A[j'][i']$
$0 \leq i \leq n$ $i+1 \leq j \leq n$ $1 \leq k \leq i-1$		$0 \leq i' \leq n$ $i'+1 \leq j' \leq n$
	$j = j', k = i'$	
$i < i'$	$i = i', j < j'$	$i = i', j = j'$
0	1	2

- Pour chaque système de contraintes, on trouve facilement une contradiction. Par exemple, à la profondeur 0 :

$$i < i' = k \leq i - 1.$$

- Existe-il une méthode systématique ?

Algorithme de Fourier-Motzkin

- On normalise toutes les contraintes sous la forme

$$h.x + k \geq 0,$$

h vecteur de constantes, x vecteur des inconnues, k constantes.

- soit $h_1^1.x + k^1 \geq 0$ et $h_1^2.x + k^2 \geq 0$ deux contraintes telles que $h_1^1 > 0$ et $h_1^2 < 0$. On forme la nouvelle inégalité :

$$(-h_1^2 h_1^1 + h_1^1 h_1^2).x - h_1^2 k^1 + h_1^1 k^2 \geq 0.$$

- Conséquence du système initial. x_1 éliminé.
- Former toutes les combinaisons possibles.
- Eliminer toutes les inconnues. Résultat : inégalités numériques $k \geq 0$. Si l'une de ces inégalités est fausse, c'est la contradiction cherchée. Sinon, le système a des solutions.

Un exemple, III

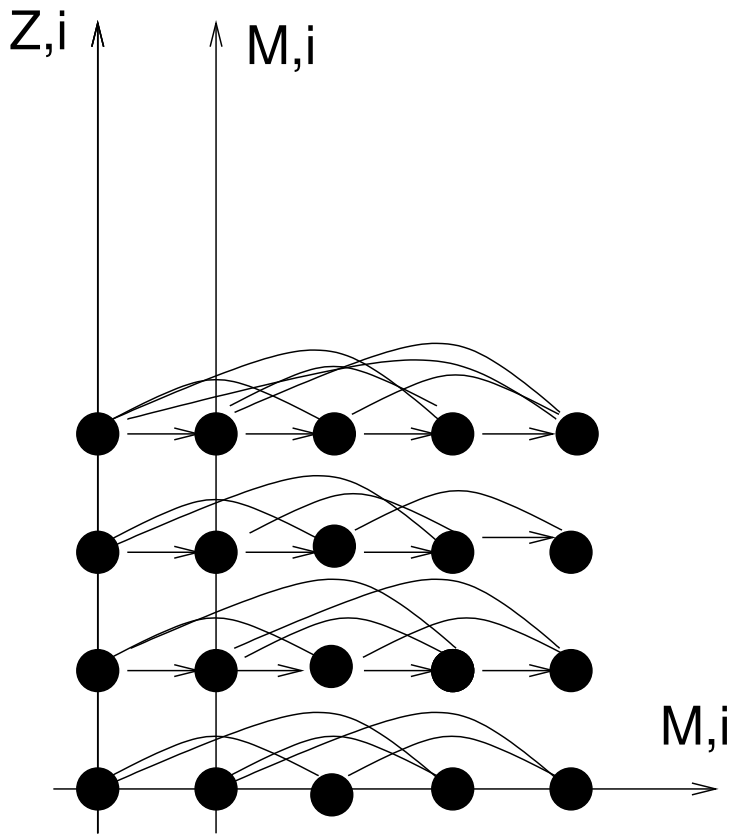
Y a-t-il une dépendance entre 3 et 6 portant sur p ? Si oui, à quelle profondeur ?

Approximation des dépendances

- Que faire si les contraintes ne sont pas affines ?
- On peut se contenter d'un calcul approximatif *conservatif*.
- Si on dit qu'il y a une dépendance alors qu'il n'y en a pas, on perd du parallélisme : ce n'est pas grave.
- Si on dit qu'il n'y a pas de dépendance alors qu'il y en a, on engendre un programme faux : c'est grave.
- En pratique, on ignore les contraintes difficiles à traiter.
- En particulier, on ignore en général le fait que les solutions trouvées doivent être entières.

Dépendances directes, I

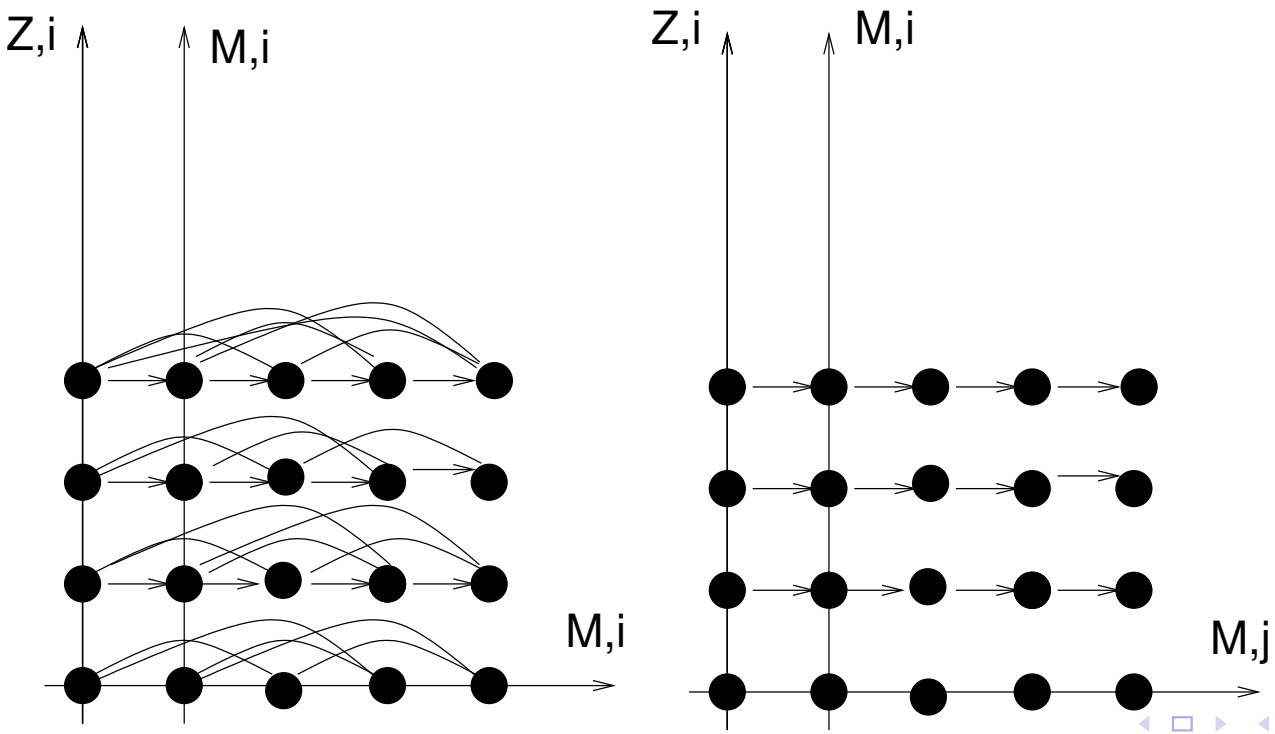
```
for(i=0; i<n; i++){
Z:   c[i] = 0.;
     for(j=0; j<n; j++)
M:       c[i] += a[i][j)*b[j];
}
```



Le graphe de dépendance détaillé a pour sommets les opérations et pour arcs les dépendances entre opérations.

Dépendances directes, II

- Le GDD est très redondant.
- Dépendances directes : ce qui reste après élimination des dépendances qui peuvent être reconstituées par transitivité.



Exemple, I

- On considère une opération de lecture, par exemple (M, i, j), qui lit $c[i]$.
- Quelle est la *source* de la valeur lue ?
- C'est la plus récente écriture dans $c[i]$ qui précède $M(i, j)$.
- Il y a deux possibilités, une instance de Z ou une instance de M.

Exemple, II

```
for(i=0; i<n; i++){
Z:   c[i] = 0.;
     for(j=0; j<n; j++)
M:       c[i] += a[i][j)*b[j];
}
```

- Cas de (Z, i').
- Ecriture dans $c[i]$: $i = i'$.
- Précède (M, i, j) : $i' \leq i$.
- Une seule solution : $i' = i$.

- Cas de (M, i', j').
- Ecriture dans $c[i]$: $i = i'$.
- Précède (M, i, j) : $i' < i \vee (i' = i \ \& \ j' < j)$.
- Le premier sous cas conduit à une contradiction. La plus tardive écriture dans le deuxième sous-cas donne $j' = j - 1$, a condition que $j \geq 1$.

Exemple, III

- Il faut maintenant trouver la plus récente solution.
- Pour $j = 0$, il n'y a qu'une possibilité, (Z, i).
- Pour $j > 0$, il y a le choix entre (Z, i) et $(M, i, j-1)$ et c'est la dernière qui est la plus récente.
- On peut résumer sous la forme :

$$\text{source}(A[i], M, i, j) = \textbf{if } j = 1 \textbf{ then } (Z, i) \textbf{ else } (M, i, j-1).$$

- Comment automatiser ?

Programmes réguliers

- Trouver la dépendance directe est un problème d'optimisation sous contraintes. On ne sait le résoudre facilement que dans le cas linéaire.
- On impose donc les contraintes suivantes :
  - Les indices des tableaux sont fonctions linéaires des compte-tours des boucles englobantes et de paramètres.
  - Les bornes des boucles sont fonctions linéaires des compte-tours des boucles englobantes et de paramètres.
  - Les paramètres ne doivent pas être modifiés dans le programme.
- Les programmes satisfaisant ces contraintes sont dits *réguliers*.
- Contrairement à ce qui se passe pour les dépendances, il est impossible d'approximer.

Construction du problème

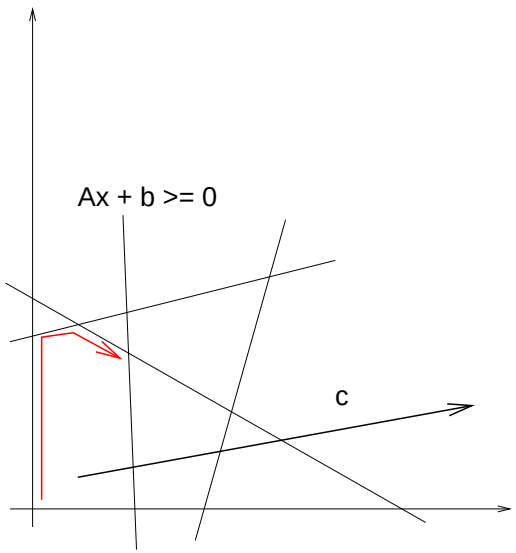
- On considère (S, i) qui lit $A[f(i)]$.
- Une source ne peut être qu'un écriture dans A. En général, il y en a plusieurs et on doit les considérer toutes. On en choisit une, (T, j) qui écrit dans $A[g(j)]$.
- Les indices doivent être égaux : $f(i) = g(j)$.
- (T, j) doit être une itération légale : $j \in D_T$.
- $(T, j) \prec (S, i)$.
- Il faut donc rechercher le maximum dans l'ordre lexicographique de l'ensemble :

$$Q(i) = \{j \mid j \in D_T, (T, j) \prec (S, i), f(i) = g(j)\},$$

qui est un polyèdre. Noter que ce polyèdre dépend de i. Le problème est paramétrique.

Programmation linéaire continue

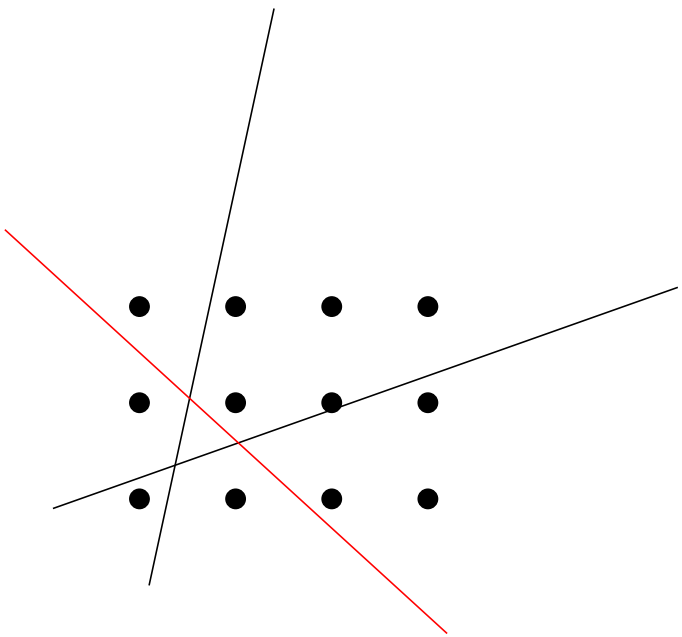
$$\begin{array}{rcl} \min & c.x & \\ Ax + b & \geq & 0 \\ x & \geq & 0 \end{array}$$



- Les contraintes définissent un polyèdre, et la solution est nécessairement en un "coin" du polyèdre.
- On va de coin en coin en suivant les arètes et en essayant de faire croitre $c.x$ le moins possible.
- On s'arrète quand on a atteint un point faisable.

Complexité : en théorie exponentielle, en pratique polynomiale.

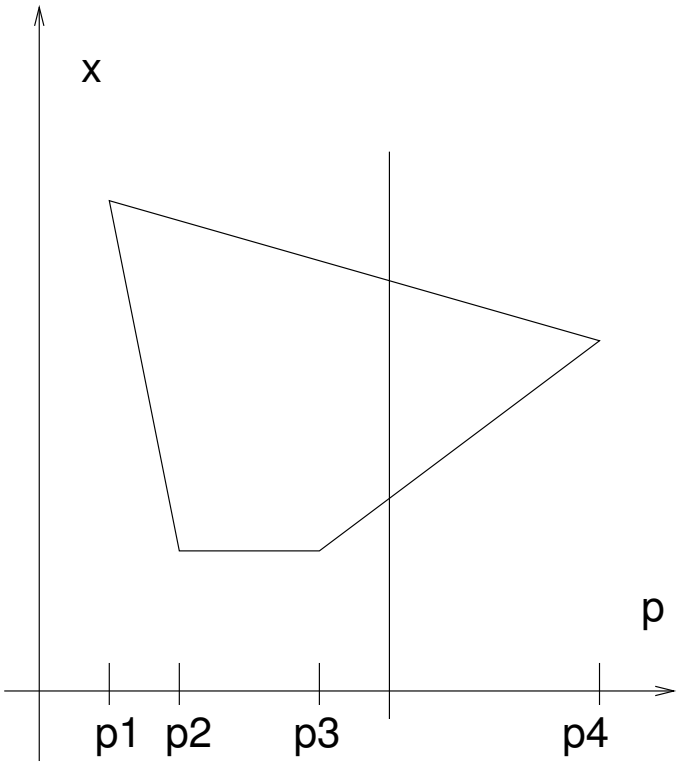
Programmation linéaire en nombres entiers (PLNE)



- On ne s'interesse qu'aux solutions entières.
- On calcule la solution rationnelle. Si elle n'est pas entière, on construit une *coupe* : une contrainte qui élimine la solution rationnelle tout en conservant la soltion entière.
- On recommence jusqu'à convergence.

Le problème devient NP-complet.

Programmation paramétrique



$$\begin{aligned} \min \quad & c.x \\ Ax + Dp + b \quad & \geq \quad 0 \\ x \quad & \geq \quad 0 \end{aligned}$$

- p est le vecteur des paramètres (entiers).

Exercice

```
  for(k=0; k <= m+n; k++)
Z:  c[k] = 0.0;
  for(i=0; i<=m; i++)
    for(j=0; j<=n; j++)
P:    c[i+j] = c[i+j] + a[i]*b[j];
```

Quelle est la source de `a[i+j]` dans l'instruction P ?