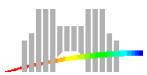
Master Informatique Fondamentale - M1 Compilation

Analyse Statique

Paul Feautrier

ENS de Lyon
Paul.Feautrier@ens-lyon.fr
perso.ens-lyon.fr/paul.feautrier

12 mai 2007



Analyse Statique

- Tout ce que l'on peut savoir sur un programme sans l'exécuter – et en particulier sans connaître ses données
- C'est la seule analyse que puisse faire un compilateur
- Avoir des informations sur un programme permet de mieux le compiler
- Exemple trivial : on peut enlever une instruction si on sait qu'elle ne sera jamais exécutée ou que son résultat ne sera jamais utilisé.

Résultats classiques d'indécidabilité, I

Dès qu'un langage de programmation est assez puissant (présence de boucles `while`), il est indécidable de savoir si un programme se termine [Turing]
Esquisse de la preuve

- Un programme prend comme argument une chaîne de caractères et se termine en émettant une chaîne de caractères, et on note $r := P(w)$ ou ne se termine pas.
- On peut coder un programme P par une chaîne de caractères $[|P|]$
- Raisonnement par l'absurde : il existe une programme T tel que $T([|P|] * w)$ retourne tt si le calcul de $P(w)$ se termine et ff sinon.
- Soit T' le programme $T'(x) = \textbf{while}(T(x * x) = tt);$
- Que fait $T'([|T'|]) = \textbf{while}(T([|T'|] * [|T'|]) = tt);$?

Résultats classiques d'indécidabilité, II

L'équivalence de deux programmes est indécidable.

- Soit Z qui retourne tt quelque soit son argument
- Soit P un programme quelconque. La terminaison de P est équivalent à l'équivalence de $\{P(x); \texttt{return}(tt)\}$ et de Z

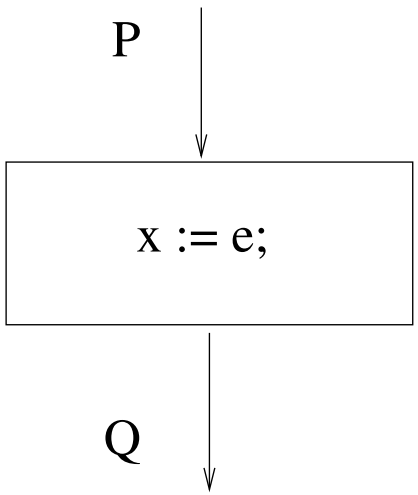
Méthode de Floyd

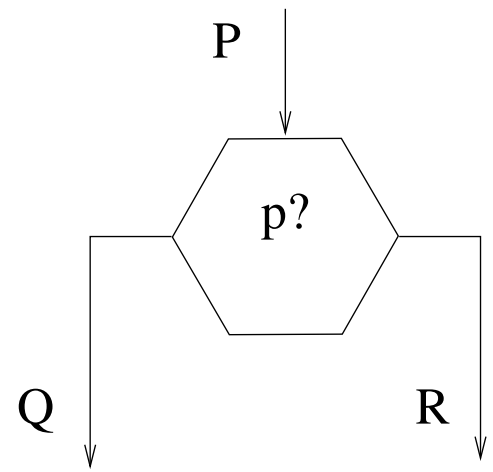
Il existe de nombreuses méthodes pour prouver des propriétés de programme. Méthode de Floyd.

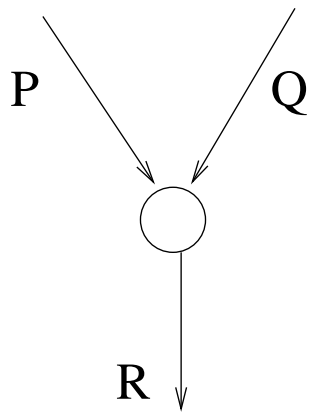
- On réduit le graphe de contrôle à 3 types de sommets
  - affectation
  - test
  - jointure

  plus un sommet de départ et un sommet de fin
- On associe un prédicat à chaque arc du GC. Le prédicat porte sur la valeur des variables du programme et on espère qu'il est vrai chaque fois que l'arc est traversé
- Pour chaque sommet, on doit prouver un théorème sur les prédicats des arcs incidents
- Si on y parvient, tous les prédicats sont vérifiés

Conditions de vérification







$P \Rightarrow Q[x \leftarrow e]$ $\quad P \ \& \ p \Rightarrow Q, P \ \& \ \neg p \Rightarrow R$ $\quad P \Rightarrow R, Q \Rightarrow R$

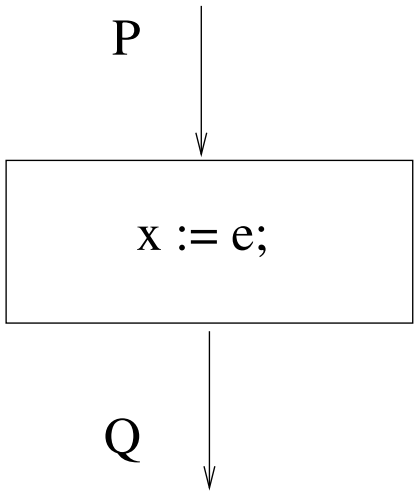
Problèmes

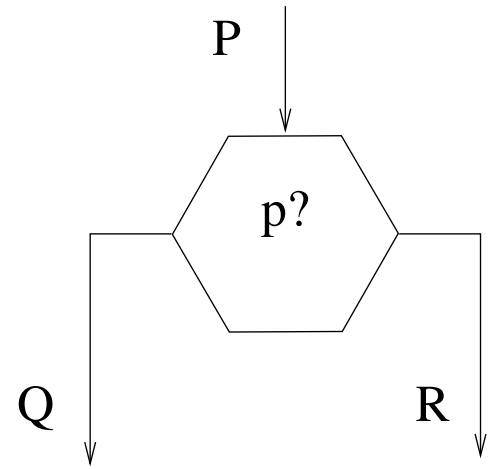
- Toute logique suffisamment puissante est indécidable
- Même si on connait les prédicats d'entrée et de sortie, il est impossible d'automatiser la construction des prédicats internes (spécialement les prédicats relatifs aux boucles)
- La méthode ne peut pas être intégrée à un compilateur

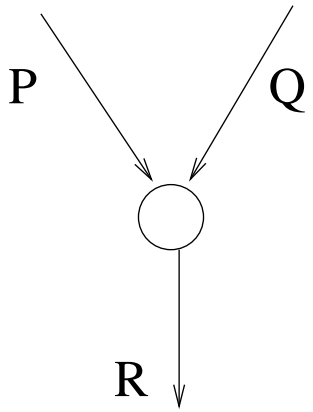
Analyses naïves

- On se restreint à des prédicats de la forme $x \in E$ où E est un sous-ensemble du domaine D_x de la variable x
- Les ensembles E font partie *d'une échelle d'ensembles* prédéfinie, $\mathcal{E}$ qui doit contenir D_x
- Pour que l'on puisse faire marcher une analyse à la Floyd, il faut remplir les conditions suivantes :
  - Jointure : $E, F \in \mathcal{E} \Rightarrow \exists G \in \mathcal{E} : E \cup F \subseteq G$. On pose $G = E \wedge F$
  - Test : pour tout prédicat p du langage, $E \in \mathcal{E} \Rightarrow \exists G \in \mathcal{E} : E \cap p \subseteq G$. On pose $G = p(E)$
  - Affectation : pour toute fonction f du langage, $E \in \mathcal{E} \Rightarrow \exists G \in \mathcal{E} : f(E) \subseteq G$ et des formules analogues pour les fonctions binaires, etc. On pose $G = f(E)$

Conditions de vérification, II







$Q = e(P)$ $\quad$ $Q = p(P), R = \neg p(P)$ $\quad$ $R = P \wedge Q$

Il est facile de voir que si ces équations sont satisfaites, les critères de Floyd sont vérifiées et donc que les propriétés associées aux arcs sont vraies.

Résolution

On utilise en général la méthode des itérations chaotiques :

- A partir du graphe de contrôle, on écrit toutes les équations de l'analyse. Soit $P_1, \ldots, P_n$ les inconnues et $D_1, \ldots, D_n$ leurs domaines
- On initialise P_i à D_i
- On range toutes inconnues $P_1, \ldots, P_n$ dans une file d'attente
- tant que la file d'attente n'est pas vide,
  - extraire une inconnue P et recalculer sa valeur
  - si cette valeur a changé, rajouter à la file d'attente toutes les inconnues qui dépendent de P

Questions de convergence

Pour pouvoir garantir la convergence, on impose des conditions à l'échelle d'ensembles $\mathcal{E}$:

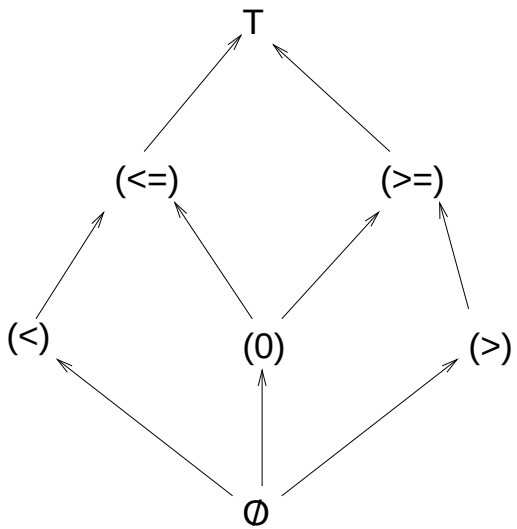
- $\mathcal{E}$ doit être ordonné
- les opérations $\wedge$, $e(\_)$ et $p(\_)$ doivent être monotones
- Il ne doit pas exister dans $\mathcal{E}$ de chaîne infinie décroissante pour l'inclusion

Dans ces conditions l'algorithme de résolution se termine après un nombre fini d'itérations.

Un exemple : Le Signe d'une Variable Scalaire

Le signe d'une variable est un information intéressante : elle permet par exemple de supprimer une vérification avant d'extraire une racine carrée. L'échelle d'ensemble 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, notée $\subset$, signifie "est plus précis que" : $(0) \subset (\leq)$: 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.
- Comme la longueur d'une chaîne est au plus 4, l'algorithme de résolution converge

Cadre d'analyse

Un cadre d'analyse est une structure mathématique permettant de raisonner sur des propriététs de programme. Le critère essentiel : les raisonnements doivent pouvoir être mis en facteurs ou mécanisés. Un cadre se compose de :

- Un ensemble ordonné qui représente les propriétés intéressantes.
- Un ensemble de fonctions qui représentent la façon dont les propriétés se transforment lors de l'exécution du programme (i.e. lors du parcours du graphe de contrôle)
- Des règles pour écrire les équations satisfaites par les propriétés
- Un algorithme de résolution et sa preuve de convergence

Demi treillis des propriétés

On donne en général à l'ensemble des propriétés une structure de demi-treillis $\mathcal{L} = (S, \wedge, \bot)$.

- $\wedge$ est associative, commutative et idempotente ($x \wedge x = x$)
- $\bot$ est élément neutre pour $\wedge$: $\bot \wedge x = x$
- La relation d'ordre associée est $x \leq y \Leftrightarrow x \wedge y = y$. C'est un ordre partiel en général. $x < y = x \leq y \ \ \& \ \ x \neq y$
- En particulier, $\forall x : \bot \leq x$
- On impose le plus souvent la propriété :

  Toute chaîne ascendante $x_1 < x_2 < \ldots < x_n \ldots$ est de longueur finie

Fonctions de transfer

A chaque sommet du graphe de contrôle, on doit pouvoir associer une fonction de transfer de $\mathcal{L}$ dans $\mathcal{L}$. L'espace des fonctions de transfer doit avoir les propriétés suivantes :

- Contenir l'identité (l'instruction NOP)
- être clos par composition (la séquence)
- ne contenir que des fonctions monotones : $x \leq y \Rightarrow f(x) \leq f(y)$

Point fixe

- Ces conditions garantissent que toute fonction a un point fixe – la limite de la suite ascendante $f^n(\bot)$
- et que ce point fixe est le plus petit point fixe :

Démonstration.
Soit z un autre point fixe. $z = f(z)$, donc $z = f^n(z)$.
Comme f^n est monotone, de $\bot \leq z$, on déduit $f^n(\bot) \leq f^n(z) = z$.
CQFD □

Equations

- A chaque sommet du graphe de contrôle on associe une ou plusieurs équations dans $\mathcal{L}$ de l'une des formes :
  - $x = f(y)$ pour une instruction d'affectation on un test
  - $x = y \wedge z$ pour un point de jonction
- Le système obtenu est de la forme $X = F(X)$ où X est un tuple d'éléments de $\mathcal{L}$
- L'algorithme itératif calcule un point fixe solution, et sa convergence est garantie par la condition de chaîne ascendante.

Quelques exemples

- L'analyse des signes remplit toutes les conditions ci-dessus
- Propagation des constantes entières
  - propriétés : l'ensemble des entiers $\mathbb{N}$ plus un élément supplémentaire $\top$.
  - x a la propriété n signifie que x est constante et égale à n
  - x a la propriété $\top$ signifie que x n'est pas constante
  - $x < \top$ et $x \wedge y = \top$ si $x \neq y$
  - fonctions : les règles de calcul usuelles plus $x \otimes \top = \top$.

Le treillis des intervalles, I

- propriétés : $x \in [a, b]$
- $[a, b] \wedge [c, d] = [\min(a, c), \max(b, d)]$
- la condition de chaîne ascendante n'est plus remplie :

$$[-1, 1] < [-2, 2] < \ldots$$

- On doit utiliser un accélérateur de convergence ou opérateur d'élargissement

Opérateur d'élargissement

Un opérateur d'élargissement a les propriétés

- $x \nabla y \geq x, y$
- quelque soit la suite x_n, la suite (acendante) $y_1 = x_1, y_{n+1} = y_n \nabla x_n$ se stabilise en un temps fini

On forme la suite $y_1 = \bot, y_{n+1} = y_n \nabla f(y_n)$ qui se stabilise par hypothèse. On montre que la limite y est une approximation du plus petit point fixe x de f : $x \leq y$.

Le treillis des intervalles, II

Opérateur d'élargissement :

$$[a, b] \wedge [c, d] = [x, y]$$

avec

- $x = a$ si $a = c$ sinon $-\infty$
- $y = b$ si $b = d$ sinon ∞

Analyses non standard

On peut être intéressé par des propriétés qui ne s'expriment pas comme des contraintes sur les valeurs des variables :

- Exemples : les *sources* : S est une source de x en T si S écrit x et s'il existe un chemin allant de S en T et ne passant par aucune écriture de x
- Pour calculer les sources (*use-def chains*) il suffit d'étudier le programme obtenu en remplaçant toute instruction $S : x := e;$ par $x := S;$

Use-def chains

- Soit L l'ensemble des labels du programme et V l'ensemble des variables. Le treillis des propriétés est l'ensemble des fonctions de V dans l'ensemble des parties de L.
- $f \wedge g = \lambda x.f(x) \cup g(x)$
- La fonctionnelle associée à $S : x := e;$ est $F(f) = \lambda y.\textbf{if } x = y \textbf{then } \{S\} \textbf{else } f(y)$. Cette fonction est clairement croissante
- L'ordre associé à $\wedge$ est l'inclusion variable par variable. Toute chaîne ascendante se stabilise parce que les ensembles $f(x)$ sont de taille bornée.

Expressions disponibles

Une expression est disponible en un point du programme si elle a été calculée précédemment et si aucun de ses arguments n'a été modifié. Démarche analogue :

- Le programme transformé a une seule variable, D, l'ensemble des expressions disponibles
- On note $x \in e$ si la variable x figure dans l'expression e
- On remplace une instruction $x := e;$ par $D := \{f \in D \mid x \notin f\} \cup \{e\}$
- On étudie des propriétés de la forme $X \subseteq D$
- En un point de jonction, si les arcs entrants portent X et Y, l'arc sortant porte $X \cap Y$.

Vivacité

Pour construire les ensembles de variables vivantes, il faut imaginer que le graphe de contrôle est parcouru *à l'envers*

- Un point de jonction devient un test, et un test devient un point de jonction
- Une variable devient vivante si elle est lue, et elle cesse d'être vivante quand elle est écrite
- Pour le reste, l'analyse est analogue à celle des expressions disponibles :
  - une seule variable ; L
  - une variable lue est ajoutée à L
  - une variable écrite est enlevée de L

La forme SSA

- La forme SSA peut être vue comme une représentation compacte des *use-def chains* – taille linéaire plutôt que quadratique
- ou comme une mise en facteur de plusieurs analyses de flot de données

Un programme est en forme SSA si :

- Toute variable scalaire ne figure qu'une fois à gauche d'une instruction d'affectation
- En lecture, des pseudo-fonctions ont été insérée pour représenter le fait qu'une même variable peut avoir plusieurs sources

ϕ-fonctions

```
if(x >= 0){          if(x >= 0){
y = ...;             y1 = ...;
} else {             } else {
y = ...;      ⇒      y2 = ...;
}                    }
t = y;               t = φ(y1, y2);
```

- La fonction ϕ est un sélecteur qui choisit la bonne valeur en fonction du chemin dans le graphe de contrôle
- Attention, la forme SSA est une représentation intermédiaire, pas un programme exécutable

Construction de la forme SSA

Méthode naïve :

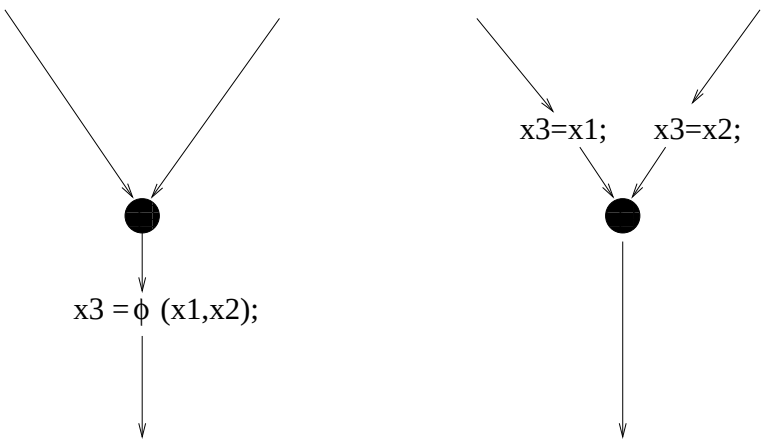
- On construit les use-def chains
- On numérote les écritures
- Pour chaque lecture
  - On détermine la liste des variables sources
  - S'il n'y a qu'une source, on la substitue à la variable lue
  - Sinon, on crée une nouvelle variable, on lui affecte une ϕ-expression sur les sources, on la place au début du bloc de base courant, et on substitue la nouvelle variable à la variable lue
- On élimine les ϕ-expressions redondantes

La méthode ci-dessus n'est pas toujours optimale.

Sortie de SSA, I

- La forme SSA est un intermédiaire sur lequel il est plus facile d'effectuer des optimisations que sur le programme original
- Aprés optimisation, il faut revenir à un programme normal, c'est-à-dire supprimer les ϕ fonctions

Principe : on insère des instructions de copie :



Sortie de SSA, II

Algorithme :

- Soit $y = \phi(a_1, \ldots, a_n)$ une ϕ-expression.
- Pour chaque argument a_k, il existe un unique bloc de base contenant une affectation à a_k (définition de la SSA)
- Sur le dernier arc de chaque chemin simple allant de ce bloc de base à la ϕ-expression à éliminer, insérer une instruction $y = a_k$.

Les instructions ajoutées sur un arc peuvent étre regroupées dans de nouveaux blocs de base ou parfois être insérées dans la source ou la destination de l'arc (quand cela n'entraine pas d'ambiguité).

Analyse de pointeurs

- Très importante : permet de savoir si une variable peut être mise dans un registre, si une expression est vraiment disponible, etc
- Exemple simple : les références de Java. Pas d'arithmétique sur pointeur, pas d'opérateur de calcul d'adresse. Peut être réduit à 5 formes :

```
p = new Class;   p = null;      p = q;
p = q.a;                        p.a = q;
```

- Il existe deux types d'analyses :
  - Analyses avec mémoire : on cherche à schématiser l'ensemble des objets accessibles depuis une référence
  - Analyses sans mémoire : on cherche à approximer directement la relation d'alias

Analyse avec mémoire

- On calcule pour chaque référence p et pour chaque point de programme l'ensemble A_p des objets accessibles à travers le pointeur p
- Un objet est nommé par le nom de sa classe et le label de l'instruction qui le créé.
- Equations :

`L : p = new Class()`	$A_p := \{\texttt{Class(L)}\}$
`p = null ;`	$A_p := \emptyset$
`p = q`	$A_p := A_q$
`p = q.a`	$A_p := A_q$
`p.a = q`	$A_p := A_p \cup A_q$

- On résoud ensuite par une méthode de point fixe.

Analyse sans mémoire, I

L'analyse cherche à estimer des relations d'alias composées d'item de la forme $\langle p.v, q.w \rangle$ où :

- p et q sont des références
- v et w sont des chaînes de sélecteurs (éventuellement vide).
- le sens d'un tel item est que $p.v$ et $q.w$ peuvent pointer vers le même objet du tas
- On notera que d'un tel item on peut en dériver d'autres. Si a est un membre légal de l'objet $p.v$ alors $p.v.a$ ets aliasé avec $q.w.a$. Un tel alias est dit implicite. Attention, la relation d'alias n'est pas transitive
- Le but de l'analyse est d'associer une relation d'alias à chaque arc du graphe de contrôle. On sait déja qu'après un poit de jonction on fait l'union des relations entrantes. Il reste à traiter le cas des 5 instructions ci-dessus.

Analyse sans mémoire, II

Notations :

- *In* et *Out* sont les relations d'alias juste avant et juste aprés l'instruction distinguée
- $\triangleright$ est la relation de préfixe : $e \triangleright f \Leftrightarrow \exists g : f = e.g$
- Δ_e est l'opération d'efffacement des items contenant e :

$$\Delta_e(R) = \{\langle g, h\rangle \mid e \not\triangleright g \ \ \& \ \ e \not\triangleright h\}$$

- $[e \leftarrow f]$ est l'opération de *substitution additive* de e par f :

$$[e \leftarrow f]R = R \cup \{\langle f.x, g\rangle \mid \langle e.x, g\rangle \in R\} \cup \{\langle g, f.x\rangle \mid \langle g, e.x\rangle \in R\}$$

Analyse sans mémoire, III

Effets :

- `p = null ;` et `p = new .. ;` détruisent les alias de `p` :

$$Out = \Delta_p(In)$$

- Les trois autres instructions peuvent être groupées sous la forme `p.v = q.w ;` où `v` et `w` sont des chaînes de sélecteurs éventuellement vides.
- Un premier effet est de détruire tous les alias de `p.v`
- Il faut ensuite ajouter l'item $\langle p.v, q.w \rangle$
- Enfin, tout ce qui est aliasé à `q.w` est maintenant aussi aliasé à `p.v`. Au total :

$$Out = [q.w \leftarrow p.v]\Delta_{p.w}(In) + \langle p.v, q.w \rangle.$$

Analyse sans mémoire, IV

Utilisation.

- Un exemple : l'amélioration du calcul des expressions disponibles.
- Lorsque l'on rencontre une écriture, on doit considérer que toutes les référencess en alias avec la référence modifiée sont modifiées
- On doit donc invalider toutes les expressions qui contiennent un alias, implicite ou explicite
- Ceci permet d'étendre la recherche des calculs redondants à des langages à pointeurs.

Bilan

- Les renseignements obtenus par l'analyse statique sont d'une grande utilité pour le compilateur
- Cependant, la précision de l'analyse est limitée par la méconnaissance des données
  - On est obligé en général de considérer les deux directions possibles d'un test
  - On ne sait pas borner le nombre de tours des boucles