Ecole Normale
Supérieure de Lyon

Ecole Nationale
des Ponts et Chaussées

L'analyse d'alias : de l'interprétation abstraite à l'analyse de flot

Pascal ZIMMER, sous la direction de René LALEMENT et Hervé GRALL

Stage de fin de première année du Magistère d'Informatique
et Modélisation, effectué du 1er Juin au 11 Juillet 1998
au laboratoire CERMICS de l'Ecole Nationale des Ponts et Chaussées

Table des matières

Chapitre 1

Présentation du stage

1.1 Le CERMICS

Le CERMICS (Centre d'Enseignement et de Recherche en Mathématiques, Informatique et Calcul Scientifique) est un laboratoire commun à l'ENPC (Ecole Nationale des Ponts et Chaussées) et à l'INRIA. Son effectif est installé pour moitié à Sophia-Antipolis, dans les Alpes Maritimes, sur l'un des sites de l'INRIA, et pour moitié dans les locaux de l'ENPC, à Champs-sur-Marne. L'activité scientifique du CERMICS est regroupée en quatre axes de recherche :

- Calcul scientifique
- Informatique et Applications
- Géométrie et Mécanique
- Probabilités et Applications

Les chercheurs du CERMICS sont pour certains, par ailleurs, enseignants à l'ENPC mais aussi en DEA. Le CERMICS est également sollicité par les entreprises tant publiques que privées.

L'équipe d'informatique s'intéresse plus particulièrement aux algorithmes de résolution des problèmes décrits à l'aide de contraintes, à la vision par ordinateur, aux bases de données sur les réseaux et au génie logiciel, en tant que méthodes formelles de développement et d'analyse de programmes. C'est dans le cadre de ce dernier thème que se plaçait le stage.

1.2 Le sujet

Ce stage s'inscrit dans le cadre d'une thèse préparée par Hervé Grall : "*Vérification de propriétés concernant la sécurité lors de l'exécution de codes mobiles*". Avant d'exécuter un bout de code mobile (par exemple, une applet Java), on voudrait pouvoir s'assurer qu'il ne pourra pas causer de torts à l'environnement d'exécution. Pour cela, on peut entre autres vérifier qu'à aucun moment le code mobile ne possèdera de pointeur lui permettant d'écrire dans une zone mémoire de l'environnement d'exécution local à laquelle il ne devrait normalement pas avoir accès. On a recours pour cela au problème plus général de l'*analyse d'alias* : étant donné un programme permettant la manipulation de pointeurs, on cherche à connaître tous les chemins d'accès à une variable donnée. Par exemple, après l'exécution du programme suivant : `(x := &z; y := &z)`, `*x` et `*y` sont dits *synonymes* (ou *alias*) puisqu'ils représentent tous deux un chemin d'accès à une même variable `z`.

Pour effectuer cette analyse, on s'inspirera des algorithmes d'*analyse de flot* décrits dans [1]. A l'origine, cette technique fut développée pour optimiser le code produit par un compilateur. L'analyse d'alias montre qu'elle peut aussi servir dans d'autres domaines.

Malheureusement, la plupart des propriétés sémantiques étudiées par l'analyse de flot sont des propriétés indécidables. On ne peut donc pas obtenir l'information exacte et on doit se contenter d'approximations (supérieures afin de garantir la sûreté de notre algorithme). Ainsi, dans le cas de l'analyse d'alias, si deux chemins d'accès sont réellement synonymes, ils seront déclarés comme tels par notre algorithme. Mais cet algorithme pourra également déclarer synonymes deux chemins qui ne le sont pas en réalité. On obtient bien une approximation supérieure de la solution souhaitée. Afin d'étudier plus précisément l'approximation effectuée et, par la même occasion, de prouver le bon fonctionnement de notre algorithme, on fera appel au formalisme introduit par l'*interprétation abstraite*. En particulier, on suivra la présentation et les notations utilisées dans [2].

Le but de ce stage était double : à la fois effectuer une étude théorique de l'analyse d'alias, mais également proposer une implémentation des algorithmes en *Caml*, afin d'avoir réellement à notre disposition un outil pratique d'analyse. Le code source de ce programme est donné en annexe (section 7.2).

Chapitre 2

L'analyse de flot

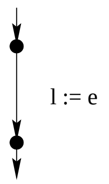
L'analyse de flot est une technique consistant à rassembler des informations en chacun des points d'un programme pour une caractéristique donnée. Parmi les applications courantes utilisées pour l'optimisation de code, citons l'étude des variables actives, des définitions visibles, des expressions disponibles, etc...

2.1 Le graphe du flot de contrôle

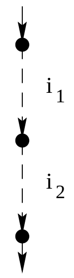
Dans tous les cas, la première étape est la construction du graphe de flot de contrôle associé à un programme, dans lequel les sommets représentent les points du programme et les arcs sont étiquetés par les instructions ou tests élémentaires reliant les différents sommets. Dans le cas de notre langage, les instructions élémentaires seront les affectations (du type `x := y`, cf. figure 4.1 pour la syntaxe exacte) ainsi que l'instruction `skip` pour gérer facilement les boucles. Un graphe de flot est donc la donnée d'un graphe $([1, n], E)$ (où n est le nombre de sommets, le sommet 1 représente le début du programme, le sommet n le point de sortie du programme et $E \subseteq [1, n]^2$ est l'ensemble des arcs) et des instructions $C_{i,j}$ associées à chaque arc $(i, j) \in E$.

Plus précisément, le graphe de flot de contrôle est défini par récurrence sur la syntaxe du langage par les dessins ci-dessous. Dans ces dessins, chaque graphe possède un sommet d'entrée et un sommet de sortie, matérialisés par de petites flèches entrantes et sortantes. Les traits pleins indiquent des arcs dans le graphe final, étiquetés par les instructions d'affectation ou les tests. Les traits en pointillés sont étiquetés par des instructions et sont à remplacer par le sous-graphe qui les représentent, en prenant comme sommets d'entrée et de sortie les sommets de l'arc en pointillés.

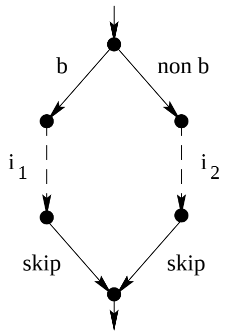
```
l := e
```



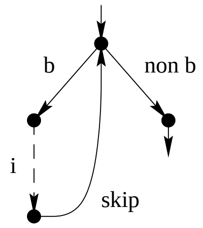
`i1;i2`



`if b then (i1) else (i2)`



`while b do (i)`



Exemple 2.1 *Le programme suivant :*

```
(while x < 100 do
   (if x = 0 then (x := 200)
             else (skip);
    x := x - 1
   )
)
```

est représenté par le graphe de la figure 2.1.

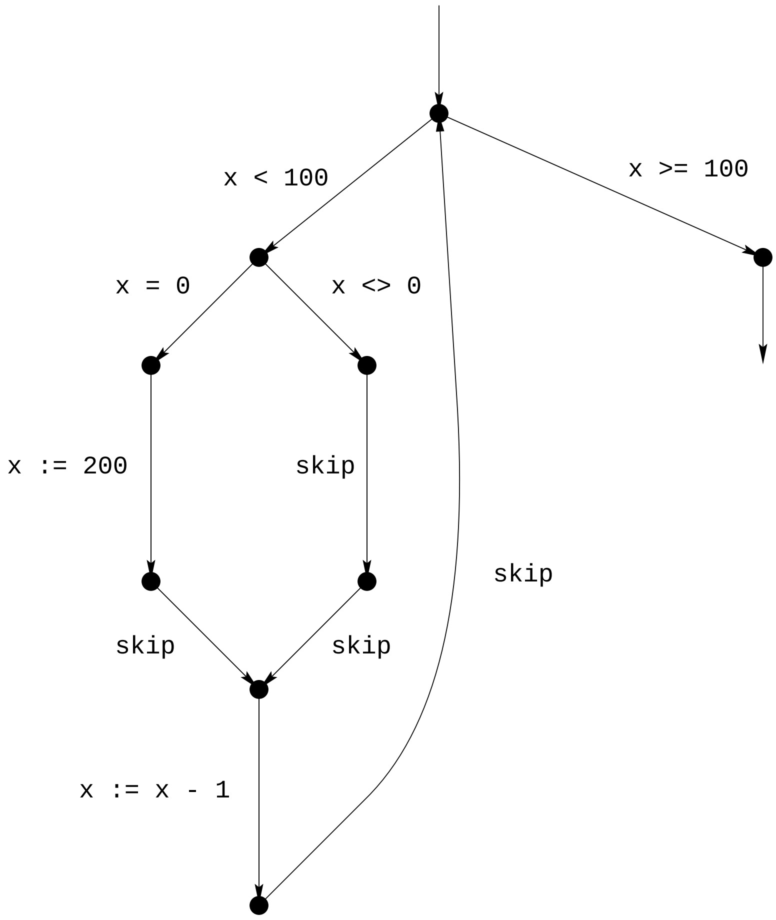
2.2 L'analyse

On va représenter l'information connue en chaque point du programme par des éléments d'un treillis complet $A(\leq, \perp, \top, \vee, \wedge)$. Pour chaque instruction et test élémentaire C, on se donne $t(C)$: $A \to A$ qui représente la transformation induite par l'instruction C sur un élément de A. Ces fonctions seront le plus souvent de la forme :

$$t(C)(\phi) = (\phi \setminus Supp(C)) \cup Prod(C)$$

où $Supp(C)$ est l'ensemble des informations supprimées par C et $Prod(C)$ celui des informations produites par C. En d'autres termes, ou bien les informations à la fin d'une instruction sont produites à l'intérieur de cette instruction, ou bien elles entrent au début et ne sont pas supprimées au cours de l'instruction.

FIG. 2.1 – *Graphe de flot de contrôle du programme d'exemple*



Soit également $\phi_0 \in A$ l'information connue au début du programme. L'information exacte au point i du programme (celle qu'on aimerait obtenir) est alors donnée par la réunion sur tous les chemins d'exécution possibles depuis la racine jusqu'au point i :

$$P_i = \bigvee_{p \in path(i)} \widetilde{t}(p)(\phi_0)$$

où $\widetilde{t}(p)$ désigne la composition des fonctions $t(C_{i,j})$ le long du chemin p et $path(i)$ est l'ensemble des chemins du sommet 1 au sommet i. Le n-uplet $P = (P_1, \ldots, P_n) \in A^n$ est noté plus simplement $MOP(t, \phi_0)$ (pour *merge over all paths*).

Malheureusement, pour peu qu'il y ait une boucle dans le graphe de flot, le nombre de chemins depuis la racine jusqu'au point i peut être infini. Le but de l'analyse de flot est de calculer un élément plus grand que $MOP(t, \phi_0)$ qui en sera une approximation. Soit $F(t, \phi_0)$ la fonction suivante sur A^n :

$$F(t,\phi_0)(P) = P' \text{ où } \left\{ \begin{array}{lcl} P'_1 & = & \phi_0 \vee \bigvee_{i \in pred(1)} t(C_{i,1})(P_i) \\ P'_j & = & \bigvee_{i \in pred(j)} t(C_{i,j})(P_i) \quad \text{pour } 2 \leq j \leq n \end{array} \right.$$

Théorème 2.2 *Si on suppose que les fonctions $t(C)$ sont croissantes pour toute commande C (ce qui est le cas dans toutes les applications pratiques étudiées), alors :*

$$MOP(t, \phi_0) \leq lfp\ (F(t, \phi_0))$$

où lfp désigne le plus petit point fixe (least fixpoint).

Dém.: Comme les fonctions $t(C)$ sont croissantes, $F(t, \phi_0)$ est croissante. Puisque A est un treillis complet, $F(t, \phi_0)$ admet donc bien un plus petit point fixe.
Soit $path_n(i)$ l'ensemble des chemins de longueur n de la racine jusqu'au sommet i. Soit P^n défini par :

$$P_i^n = \bigvee_{p \in path_n(i)} \tilde{t}(p)(\phi_0)$$

$P^0 = (\phi_0, \perp, \ldots, \perp)$ par définition. $P^{n+1} \leq F(t, \phi_0)(P^n)$ en développant le membre de droite. Puis $P^n \leq lfp\ (F(t, \phi_0))$ par récurrence sur n. Comme $MOP(t, \phi_0) = \vee P^n$, le résultat est démontré. □

Théorème 2.3 *Avec l'hypothèse plus forte que $t(C)$ est un morphisme complet pour $\vee$ pour toute commande C, on a l'égalité :*

$$MOP(t, \phi_0) = lfp\ (F(t, \phi_0))$$

Dém.: Il suffit de montrer que $M = MOP(t, \phi_0)$ est un point fixe de $F = F(t, \phi_0)$. Pour $2 \leq j \leq n$:

$$\begin{aligned}
F(M)_j &= \bigvee_{i \in pred(j)} t(C_{i,j}) \left(\bigvee_{p \in path(i)} \tilde{t}(p)(\phi_0) \right) \\
&= \bigvee_{i \in pred(j)} \bigvee_{p \in path(i)} t(C_{i,j}) \circ \tilde{t}(p)(\phi_0) \\
&= \bigvee_{q \in path(j)} \tilde{t}(q)(\phi_0) \quad \begin{array}{l}\text{car tout chemin de la racine au} \\ \text{sommet } j \text{ est de longueur } \geq 1\end{array} \\
&= M_j
\end{aligned}$$

Pour $j = 1$, le calcul est similaire en ce qui concerne les chemins q de longueur ≥ 1 ; et le terme supplémentaire ϕ_0 apparaissant dans $F(M)_1$ correspond quant à lui au chemin q de longueur nulle entre la racine et elle-même. □

2.3 Le calcul de points fixes

Nous allons commencer par donner quelques résultats généraux sur le calcul de points fixes dans un ensemble ordonné.

Définition 2.4 *Soit A un ensemble ordonné.*

- *Une partie X de A est* dirigée *si l'opérateur binaire $\vee$ réduit à $X \times X$ forme une loi de composition interne sur X :*

  $$\forall (x, y) \in X \times X, x \vee y \in X$$

- *Une* chaîne *de A est une partie de A totalement ordonnée. Toute chaîne est dirigée.*

- *A est* complet *si toute partie dirigée admet une borne supérieure. On montre l'équivalence de la définition suivante : toute chaîne admet une borne supérieure. Comme l'ensemble vide est dirigé, A possède un plus petit élément : $\perp$*

- *Une application F de A dans A est* continue *si* $F\left(\bigvee_{x\in X} x\right) = \bigvee_{x\in X} F(x)$ *pour toute partie dirigée X. Une définition équivalente est obtenue en se restreignant aux chaînes. Tout morphisme complet pour $\vee$ est donc continu.*

Théorème 2.5 *Si A est complet et $F : A \to A$ continue, alors F admet un plus petit point fixe.*

Dém.: Supposons F continue. Alors F est croissante (prendre $X = \{x, y\}$, avec $x \leq y$). On définit la suite (x_n) par récurrence :

$$\begin{cases} x_0 & = & \perp \\ x_{n+1} & = & F(x_n) \end{cases}$$

La suite (x_n) est croissante :

$$x_0 = \perp \leq x_1$$

$$x_n \leq x_{n+1} \Rightarrow x_{n+1} \leq x_{n+2}, \quad \text{puisque } F \text{ est croissante}$$

Soit $x = \bigvee_{n\in\mathbb{N}} x_n$ (la borne supérieure est bien définie puisque l'ensemble $\{x_n, n \in \mathbb{N}\}$ est une chaîne et que A est complet).

$$F(x) = F\left(\bigvee_{n\in\mathbb{N}} x_n\right) = \bigvee_{n\in\mathbb{N}} F(x_n) = \bigvee_{n\in\mathbb{N}} x_{n+1} = \bigvee_{n\geq 1} x_n \vee \perp = x$$

x est donc un point fixe.
Soit l un point fixe de F.

$$\perp \leq l$$

$$x_n \leq l \Rightarrow x_{n+1} \leq l, \quad \text{puisque } F \text{ est croissante}$$

Par récurrence, $x_n \leq l$, et $x \leq l$. □

Le calcul du point fixe est fondé sur le calcul de la suite (x_n). La démonstration permet de définir des hypothèses plus faibles, auxquelles on se ramène en pratique :

- si $\bigvee_{n\in\mathbb{N}} x_n$ existe et $F\left(\bigvee_{n\in\mathbb{N}} x_n\right) = \bigvee_{n\in\mathbb{N}} F(x_n)$, alors $\bigvee_{n\in\mathbb{N}} x_n$ est un point fixe.
- si de plus F est croissante, alors $\bigvee_{n\in\mathbb{N}} x_n$ est le plus petit point fixe.

On peut donc affaiblir les hypothèses de complétude et de continuité en se restreignant aux suites croissantes. Pratiquement, pour obtenir un point fixe calculable, on calcule dans un ensemble A particulier :

Théorème 2.6

- *Si A vérifie la condition de chaîne ascendante (toute suite croissante est stationnaire) et F croissante, alors $\bigvee_{n\in\mathbb{N}} x_n$ est le plus petit point fixe, atteint par un terme de la suite*
- *En particulier, si A est fini, il vérifie cette condition*

Revenons maintenant au cas particulier de l'analyse de flot. Au lieu d'une suite (x_n) avec $x_0 = \perp$, on va calculer la suite :

$$\begin{cases} x_0 & = & (\phi_0, \perp, \ldots, \perp) \\ x_{n+1} & = & F(t, \phi_0)(x_n) \end{cases}$$

Etant donné la forme particulière de $F(t, \phi_0)$, le terme ϕ_0 n'est pas gênant (il serait de toute façon introduit dans x_1), et tous les résultats et démonstrations précédents restent valables.

Chapitre 3

Interprétation abstraite

L'interprétation abstraite est une technique générale permettant de formaliser la notion d'approximation d'un calcul et de fournir des résultats quant à la qualité de ces approximations. Il existe de nombreuses approches différentes pour la décrire. Dans un souci de clarté, nous exposerons ici la vision qui nous a paru la plus intuitive. Dans le prochain chapitre, nous utiliserons une autre approche, plus directe et formelle. Toutes ces approches, ainsi que d'autres, sont décrites dans l'article [2]. Les principaux théorèmes de ce chapitre sont issus de ce même article ; par contre, les démonstrations ont dû être retrouvées puisqu'elles n'y figuraient pas, et certains théorèmes erronés ont dû être corrigés.

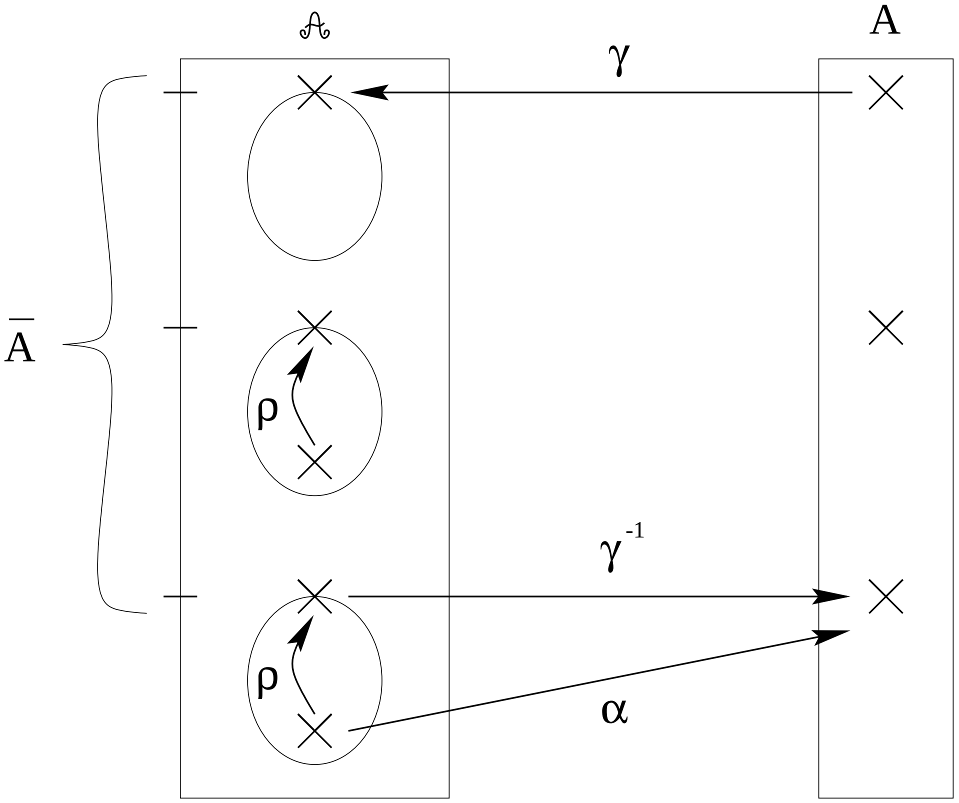
Le principe de l'interprétation abstraite est résumé dans la figure 3.1. Soit $\mathcal{A}$ un treillis dont les éléments sont les assertions sur lesquelles on veut travailler. Pour résoudre un problème donné, on voudrait pouvoir effectuer des calculs dans ce treillis en vue d'obtenir un certain résultat. Malheureusement, ceci est souvent impossible pour diverses raisons (indécidabilité du calcul, temps de calcul trop long...). On est donc amené à faire une approximation. Pour cela, on regroupe les assertions en classes d'équivalence et on choisit un représentant dans chaque classe. Soit $\overline{\mathcal{A}} \subseteq \mathcal{A}$ l'ensemble de ces représentants. On notera $\rho : \mathcal{A} \to \overline{\mathcal{A}}$ l'application qui à une assertion associe son représentant. Puis, au lieu d'effectuer les calculs dans $\mathcal{A}$, on va mettre $\overline{\mathcal{A}}$ en bijection avec un treillis A dans lequel il sera plus facile d'effectuer les calculs. On note $\gamma : A \to \overline{\mathcal{A}}$ un tel isomorphisme et $\alpha = \gamma^{-1} \circ \rho$.

Voilà pour la description intuitive ; il faut maintenant donner un cadre plus formel à cette présentation. En premier lieu, la relation d'équivalence sur $\mathcal{A}$ ne peut être quelconque pour décrire correctement l'approximation, mais doit vérifier quelques propriétés relatives à la structure de treillis. Nous allons supposer que θ vérifie les conditions suivantes :

- θ est une relation d'équivalence
- θ est régulière pour la loi $\vee$, i.e. $\forall x, y, u \in \mathcal{A},\ x \equiv_\theta y \Rightarrow x \vee u \equiv_\theta y \vee u$
- θ est complète pour la loi $\vee$, i.e. $\forall x \in \mathcal{A},\ x \equiv_\theta \bigvee_{y \in [x]} y$ où $[x]$ désigne la classe d'équivalence de x pour la relation θ

Théorème 3.1 *Soit θ vérifiant les conditions ci-dessus. On définit ρ par $\rho(x) = \bigvee_{y \in [x]} y$. Alors ρ est un opérateur de clôture supérieur sur $\mathcal{A}$ (i.e. ρ est extensif, croissant et idempotent) et $\overline{\mathcal{A}} = \rho(\mathcal{A})$ est l'ensemble des représentants cherchés.*

FIG. 3.1 – *Le principe de l'interprétation abstraite*



Dém.:

- $\forall x, x \in [x]$ donc $x \leq \rho(x)$. ρ est extensif.
- Soit $x \leq y$. Par définition de ρ, $x \equiv_\theta \rho(x)$. Donc, $y = y \vee x \equiv_\theta y \vee \rho(x)$. Alors, $y \vee \rho(x) \in [y]$. Et donc, $y \vee \rho(x) \leq \rho(y)$. Finalement, $\rho(x) \leq y \vee \rho(x) \leq \rho(y)$. ρ est croissant.
- Si $x \equiv_\theta y$, $[x] = [y]$ donc $\rho(x) = \rho(y)$. Comme $x \equiv_\theta \rho(x)$, il en résulte que $\rho(x) = \rho(\rho(x))$. Donc, ρ est idempotent.

$\square$

Théorème 3.2 *La fonction ρ peut aussi s'écrire sous la forme : $\rho(x) = \bigwedge_{y \in \overline{A} \text{ tel que } x \leq y} y$*

Dém.:

$$\bigwedge_{y \in \overline{A}\ /\ x \leq y} y = \bigwedge_{z \in \mathcal{A}\ /\ x \leq \rho(z)} \rho(z) \leq \rho(x)$$

Et quel que soit $z \in \mathcal{A}$ tel que $x \leq \rho(z)$, on a : $\rho(x) \leq \rho(\rho(z)) = \rho(z)$, et donc :

$$\rho(x) \leq \bigwedge_{z \in \mathcal{A}\ /\ x \leq \rho(z)} \rho(z)$$

□

Définition 3.3 *Soient $L_1(\leq_1)$ et $L_2(\leq_2)$ des ensembles partiellement ordonnés. $<\alpha,\gamma>$ est une* paire de fonctions adjointes *entre L_1 et L_2 si α et γ sont croissantes et si :*

$$\forall x \in L_1, \forall y \in L_2,\ x \leq_1 \gamma(y) \ \Leftrightarrow\ \alpha(x) \leq_2 y$$

Théorème 3.4 *Si ρ est un opérateur de clôture supérieur sur $\mathcal{A}$, si γ est un isomorphisme de treillis de A vers $\gamma(A) = \rho(\mathcal{A})$ et $\alpha = \gamma^{-1} \circ \rho$, alors :*

- *$<\alpha,\gamma>$ est une paire de fonctions adjointes entre $\mathcal{A}$ et A*
- *α est surjective*

Dém.: Puisque γ est un isomorphisme, γ est croissante ($x \leq y \ \Rightarrow\ \gamma(x) \leq \gamma(y)$). Alors, γ^{-1} est croissante et, comme ρ aussi, α est croissante.
On a les implications suivantes :

$$\begin{array}{rcl} x \leq \gamma(y) & \Rightarrow & \rho(x) \leq \rho(\gamma(y)) = \gamma(y) \quad \text{car } \gamma(y) \in \overline{A} \\ & \Rightarrow & \alpha(x) \leq y \\ \alpha(x) \leq y & \Rightarrow & \rho(x) \leq \gamma(y) \\ & \Rightarrow & x \leq \gamma(y) \quad \text{car } x \leq \rho(x) \end{array}$$

α est surjective car pour tout $x \in A$:

$$\begin{array}{rcl} \alpha(\gamma(x)) & = & \gamma^{-1}(\rho(\gamma(x))) \\ & = & \gamma^{-1}(\gamma(x)) \quad \text{car } \gamma(x) \in \overline{A} \\ & = & x \end{array}$$

□

Les calculs à effectuer sont donnés sous la forme de règles de transformation pour chaque commande C. Soient $\tau(C) \ :\ \mathcal{A} \to \mathcal{A}$ les règles correspondants au calcul exact dans le treillis $\mathcal{A}$. Dans toute la suite, on supposera que les fonctions $\tau(C)$ sont croissantes. Ceci correspond à l'idée intuitive que si P contient les informations de Q (i.e. $P \geq Q$), alors, après la commande C, $\tau(C)(P)$ doit contenir les informations de $\tau(C)(Q)$ (i.e. $\tau(C)(P) \geq \tau(C)(Q)$).

Nous allons maintenant introduire une hiérarchie dans les différentes approximations qu'on peut réaliser. Ces approximations seront également données sous la forme de règles de transformation sur $\overline{A}$ ou A pour chaque commande C. On les notera respectivement $\overline{t}$ et t. Au chapitre précédent, nous avons vu que l'assertion valable en un point i d'un programme pour une assertion d'entrée $\phi_0 \in \mathcal{A}$ était donnée par :

$$P_i = \bigvee_{p \in path(i)} \widetilde{\tau}(p)(\phi_0)$$

Par conséquent, la meilleure approximation qu'on puisse espérer obtenir est :

$$\rho(P_i) = \rho\left(\bigvee_{p \in path(i)} \widetilde{\tau}(p)(\phi_0)\right)$$

Mais, comme on ne peut effectuer le calcul dans $\mathcal{A}$ en pratique, on va prendre $\overline{\phi_0} \in \overline{A}$ et calculer :

$$Q_i = \rho\left(\bigvee_{p \in path(i)} \widetilde{\overline{t}}(p)(\overline{\phi_0})\right)$$

Il faut cependant s'assurer que $\overline{t}$ correspond bien à une approximation de τ.

Définition 3.5

- *Une règle de transformation $\overline{t}$ sur $\overline{A}$ est une* approximation supérieure correcte *de la règle τ sur $\mathcal{A}$ si pour tous $\phi_0 \in \mathcal{A}$ et $\overline{\phi_0} \in \overline{A}$ tels que $\phi_0 \leq \overline{\phi_0}$, on a $MOP(\tau, \phi_0) \leq MOP(\overline{t}, \overline{\phi_0})$ pour tout programme.*
- *De même, lorsque $< \alpha, \gamma >$ est une paire de fonctions adjointes entre $\mathcal{A}$ et A, une règle de transformation t sur $A = \alpha(\mathcal{A})$ est une* approximation supérieure correcte *de la règle τ sur $\mathcal{A}$ si pour tous $\phi_0 \in \mathcal{A}$ et $\phi_0' \in A$ tels que $\phi_0 \leq \gamma(\phi_0')$, on a $\alpha(MOP(\tau, \phi_0)) \leq MOP(t, \phi_0')$ pour tout programme.*

Vérifier ce genre de condition par union sur tous les chemins serait très fastidieux à réaliser. Heureusement, il est possible d'aboutir au même résultat en ne considérant que les approximations locales :

Théorème 3.6

- *$\overline{t}$ est une approximation supérieure correcte de τ si et seulement si $\forall P \in \overline{A}$, $\tau(C)(P) \leq \overline{t}(C)(P)$ pour toute commande C.*
- *Lorsque $< \alpha, \gamma >$ est une paire de fonctions adjointes, t est une approximation supérieure correcte de τ si et seulement si $\forall P \in A$, $\alpha(\tau(C)(\gamma(P))) \leq t(C)(P)$ pour toute commande C.*

Dém.: La démonstration se fait sans difficulté par récurrence sur la longueur des chemins de $path(i)$ et en utilisant la définition de MOP. □

Sous certaines conditions, il est possible d'obtenir l'égalité entre l'approximation des calculs exacts et le résultat des calculs approchés :

Théorème 3.7

- *Si $\overline{t}$ est une approximation supérieure correcte de τ et si $\forall P \in \mathcal{A}$, $\rho(\tau(C)(P)) = \overline{t}(C)(\rho(P))$ pour toute commande C, alors $\forall \phi_0$, $\rho(MOP(\tau, \phi_0)) = MOP(\overline{t}, \rho(\phi_0))$ pour tout programme.*
- *Lorsque $< \alpha, \gamma >$ est une paire de fonctions adjointes, si t est une approximation supérieure correcte de τ et si $\forall P \in \mathcal{A}$, $\alpha(\tau(C)(P)) = t(C)(\alpha(P))$ pour toute commande C, alors $\forall \phi_0$, $\alpha(MOP(\tau, \phi_0)) = MOP(t, \alpha(\phi_0))$ pour tout programme.*

Maintenant que nous avons défini ce qu'était une approximation correcte, nous pouvons les comparer entre elles :

Définition 3.8 *Si $\overline{t_1}$ et $\overline{t_2}$ sont des approximations supérieures correctes de τ dans $\overline{A}$, on dira que $\overline{t_1}$ est* meilleure *que $\overline{t_2}$ si $\forall \overline{\phi_0} \in \overline{A}$, on a $MOP(\overline{t_1}, \overline{\phi_0}) \leq MOP(\overline{t_2}, \overline{\phi_0})$ pour tout programme.*

Lemme 3.9 *Soient $\overline{t_1}$ et $\overline{t_2}$ des approximations supérieures correctes de τ dans $\overline{A}$. Si $\overline{t_1}(C) \leq \overline{t_2}(C)$ pour toute commande C et si, pour toute commande C, l'une des deux fonctions $\overline{t_1}(C)$ ou $\overline{t_2}(C)$ est croissante, alors $\overline{t_1}$ est meilleure que $\overline{t_2}$. (A noter que cette condition de croissance est une condition suffisante mais non nécessaire).*

Dém.: Soit $\overline{\phi_0} \in \overline{A}$. Montrons par récurrence sur la longueur du chemin $p \in path(i)$ que $\tilde{\overline{t_1}}(p)(\overline{\phi_0}) \leq \tilde{\overline{t_2}}(p)(\overline{\phi_0})$:
Si p est le chemin vide, alors $\tilde{\overline{t_1}}(p)(\overline{\phi_0}) = \overline{\phi_0} = \tilde{\overline{t_2}}(p)(\overline{\phi_0})$.
Sinon $p = q. < i, j >$. Supposons $\overline{t_1}(C_{i,j})$ croissante.

$$\begin{array}{rcll} \tilde{\overline{t_1}}(p)(\overline{\phi_0}) & = & \overline{t_1}(C_{i,j})(\tilde{\overline{t_1}}(q)(\overline{\phi_0})) & \\ & \leq & \overline{t_1}(C_{i,j})(\tilde{\overline{t_2}}(q)(\overline{\phi_0})) & \text{par croissance de } \overline{t_1}(C_{i,j}) \text{ et par hypothèse de récurrence} \\ & \leq & \overline{t_2}(C_{i,j})(\tilde{\overline{t_2}}(q)(\overline{\phi_0})) & \text{par hypothèse} \\ & = & \tilde{\overline{t_2}}(p)(\overline{\phi_0}) & \end{array}$$

Dans le cas où c'est $\overline{t_2}(C_{i,j})$ qui est croissante, il suffit de changer l'ordre des majorations:

$$\begin{array}{rcll} \tilde{\overline{t_1}}(p)(\overline{\phi_0}) & = & \overline{t_1}(C_{i,j})(\tilde{\overline{t_1}}(q)(\overline{\phi_0})) & \\ & \leq & \overline{t_2}(C_{i,j})(\tilde{\overline{t_1}}(q)(\overline{\phi_0})) & \text{par hypothèse} \\ & \leq & \overline{t_2}(C_{i,j})(\tilde{\overline{t_2}}(q)(\overline{\phi_0})) & \text{par croissance de } \overline{t_2}(C_{i,j}) \text{ et par hypothèse de récurrence} \\ & = & \tilde{\overline{t_2}}(p)(\overline{\phi_0}) & \end{array}$$

Puis, en faisant la réunion pour tous les chemins $p \in path(i)$, on obtient : $MOP(\overline{t_1}, \overline{\phi_0}) \leq MOP(\overline{t_2}, \overline{\phi_0})$. □

Théorème 3.10 *Soit $\overline{\tau}$ la règle de transformation définie sur $\overline{A}$ par $\overline{\tau} = \rho \circ \tau$.*

- *$\overline{\tau}(C)$ est croissante pour toute commande C.*
- *$\overline{t}$ est une approximation supérieure correcte de τ sur $\overline{A}$ si et seulement si $\overline{\tau}(C) \leq \overline{t}(C)$ pour toute commande C.*
- *$\overline{\tau}$ est la* meilleure approximation supérieure correcte *de τ sur $\overline{A}$.*

Dém.:

- Comme ρ et $\tau(C)$ sont croissantes, $\overline{\tau}(C)$ l'est aussi, et ceci pour toute commande C.
- Supposons que $\overline{t}$ soit une approximation supérieure correcte de τ. Alors, d'après le théorème 3.6, $\tau(C) \leq \overline{t}(C)$ pour toute commande C. Donc, $\overline{\tau}(C) = \rho \circ \tau(C) \leq \rho \circ \overline{t}(C) = \overline{t}(C)$ puisque $\overline{t}(C)(P) \in \overline{A}$.
  Supposons maintenant $\overline{\tau}(C) \leq \overline{t}(C)$. Comme ρ est extensif, $\tau(C) \leq \rho \circ \tau(C) = \overline{\tau}(C)$. Alors, $\tau(C) \leq \overline{t}(C)$ et donc $\overline{t}$ est une approximation supérieure correcte de τ d'après le théorème 3.6.
- En prenant $\overline{t} = \overline{\tau}$, le résultat précédent nous donne que $\overline{\tau}$ est une approximation supérieure correcte de τ. Si $\overline{t}$ est une autre approximation supérieure de τ, on a $\overline{\tau}(C) \leq \overline{t}(C)$ pour toute commande C. Et, comme $\overline{\tau}(C)$ est croissante pour toute commande C, le lemme 3.9 s'applique : $\overline{\tau}$ est une meilleure approximation que $\overline{t}$. C'est donc la meilleure de toutes les approximations.

□

Corollaire 3.11 *Lorsque $< \alpha, \gamma >$ est une paire de fonctions adjointes entre $\mathcal{A}$ et A, soit :*

$$t(C)(P) = \alpha(\tau(C)(\gamma(P))) \quad \textit{défini pour } P \in A$$

t est croissante et c'est la meilleure approximation supérieure correcte de τ dans A.

Dém.: Par isomorphisme entre $\overline{A}$ et A, on déduit du théorème précédent que $\gamma^{-1} \circ \overline{\tau} \circ \gamma$ est la meilleure approximation de τ dans A. Or,

$$\begin{aligned} \gamma^{-1}(\overline{\tau}(C)(\gamma(P))) &= \gamma^{-1}(\rho(\tau(C)(\gamma(P)))) \\ &= \alpha(\tau(C)(\gamma(P))) \\ &= t(C)(P) \end{aligned}$$

$\square$

L'intérêt de ce résultat est que pour un problème donné, il suffit d'avoir défini le treillis A et l'isomorphisme γ pour obtenir les règles de transformation à appliquer dans A afin d'avoir la meilleure approximation possible des calculs dans $\mathcal{A}$.

Chapitre 4

L'analyse d'alias

4.1 Présentation du langage utilisé

Pour pouvoir effectuer une analyse de flot, il fallait avant tout se définir un langage de travail. Le langage choisi est réduit au minimum et s'inspire du code à trois opérandes décrit dans [1]. Il contient :

- les structures de contrôle `if` et `while`, ainsi que l'opérateur de séquencement `;`
- l'affectation à une variable ou au contenu d'un pointeur
- les 4 opérations arithmétiques de base : `+`, `-`, `*` et `/`
- les opérateurs sur les pointeurs : adresse d'une variable `&` et déréférencement `*`
- les tests booléens de base

Plus précisément, la syntaxe utilisée est donnée figure 4.1 sous forme BNF.

4.2 Définitions et notations

On notera V l'ensemble des variables du programme. Pour l'algorithme, V sera l'ensemble fini des identificateurs de variables déclarées dans le programme en cours d'analyse (ceci permet de prouver la terminaison de l'algorithme). Mais, dans la théorie, rien n'empêche de prendre V infini. On pourra par exemple considérer que V est l'ensemble de tous les identificateurs possibles ou, ce qui revient au même, l'ensemble des adresses de la mémoire. Les éléments de V seront notés `x`, `y` ou `z`, éventuellement indicés.

Ces variables prendront leurs valeurs dans un ensemble noté $Value$ dont il n'est pas nécessaire de donner une définition précise en ce qui concerne la théorie. On supposera seulement qu'il contient l'ensemble des booléens $\mathbb{B} = \{True, False\}$ et un ensemble numérique afin de donner quelque chose à manipuler à notre langage. Pour fixer les idées, on pourra prendre par exemple $\mathbb{Z} \cup \mathbb{B} \cup \perp$ ($\perp$ représente la valeur indéfinie, i.e. le contenu d'une variable qui n'a pas encore été affectée). Un élément générique de $Value$ sera noté v.

Un état de la mémoire σ est un élément de $Mem = V \to V \cup Value$ avec la convention que $\sigma(\mathtt{x}) \in V$ si `x` pointe sur une autre variable ($\sigma(\mathtt{x})$ est l'adresse de cette variable) et $\sigma(\mathtt{x}) \in Value$ si `x` contient une valeur.

FIG. 4.1 – *Syntaxe BNF du langage utilisé*

$$
\begin{array}{rcl}
Ident & ::= & \texttt{a} \mid \texttt{b} \mid \ldots \mid \texttt{x11} \mid \ldots \\
 & & \text{(toute chaîne alphanumérique commençant par une lettre)} \\
Entier & ::= & \texttt{0} \mid \texttt{1} \mid \texttt{2} \mid \ldots \\
Simple & ::= & Ident \mid Entier \\
Expr & ::= & Simple \\
 & \mid & \texttt{\&}\ Ident \\
 & \mid & \texttt{*}\ Ident \\
 & \mid & Simple\ \texttt{+}\ Simple \\
 & \mid & Simple\ \texttt{-}\ Simple \\
 & \mid & Simple\ \texttt{*}\ Simple \\
 & \mid & Simple\ \texttt{/}\ Simple \\
 & \mid & Simple\ \texttt{=}\ Simple \\
 & \mid & Simple\ \texttt{<>}\ Simple \\
 & \mid & Simple\ \texttt{<}\ Simple \\
 & \mid & Simple\ \texttt{>}\ Simple \\
Loc & ::= & Ident \mid \texttt{*}\ Ident \\
Instr & ::= & \texttt{if}\ Expr\ \texttt{then (}Instr\texttt{) else (}Instr\texttt{)} \\
 & \mid & \texttt{while}\ Expr\ \texttt{do (}Instr\texttt{)} \\
 & \mid & Instr\ \texttt{;}\ Instr \\
 & \mid & Loc\ \texttt{:=}\ Expr
\end{array}
$$

Ces états-mémoires peuvent être modifiés par les instructions d'affectation. La sémantique exacte du langage est donnée à la figure 4.2 pour la sémantique des expressions et à la figure 4.3 pour celle des instructions. Il est à noter qu'afin de gérer les erreurs de type (par exemple, `x := y + 1` quand `y` est un pointeur ou un booléen), on a introduit la valeur particulière `err`, renvoyée par les fonctions $[\![.]\!]$ lorsqu'une erreur s'est produite et provoquant l'arrêt immédiat de l'algorithme. Cependant, afin de ne pas alourdir l'exposé, on ne détaillera pas toujours le cas particulier de `err` dans la suite, et on supposera que les programmes à analyser sont bien typés et ne produisent pas d'erreurs de type.

Nous avons maintenant défini toute la sémantique du langage. Cependant, nous n'allons pas travailler directement sur les expressions syntaxiques, mais sur le graphe de flot de contrôle, et associer une assertion à chacun de ses sommets. Ces assertions seront notées Σ et seront éléments de $\mathcal{A} = \mathcal{P}(Mem)$. L'utilité de prendre des ensembles d'états mémoire à la place d'un seul est qu'on va pouvoir ainsi représenter le fait qu'il est possible d'avoir deux états mémoire différents en un point donné (correspondant par exemple aux deux branches d'un `if`). On vérifie que $\mathcal{A}(\subseteq, \emptyset, Mem, \cup, \cap)$ forme un treillis complet.

Il nous faut maintenant définir les règles de transformation $\tau(C)$ pour chaque commande C

FIG. 4.2 – *Sémantique des expressions*

$$
\begin{array}{rcl}
[\![\mathtt{v}]\!](\sigma) & = & \mathtt{v} \text{ pour } \mathtt{v} \in Value \\
[\![\mathtt{x}]\!](\sigma) & = & \sigma(\mathtt{x}) \text{ pour } \mathtt{x} \in V \\
[\![\mathtt{\&x}]\!](\sigma) & = & \mathtt{x} \text{ pour } \mathtt{x} \in V \\
[\![\mathtt{*x}]\!](\sigma) & = & \begin{cases} \sigma(\sigma(\mathtt{x})) & \text{si } \sigma(\mathtt{x}) \in V \\ \mathtt{err} & \text{sinon} \end{cases} \\
[\![\mathtt{s}_1 \mathtt{\ +\ } \mathtt{s}_2]\!](\sigma) & = & \begin{cases} [\![\mathtt{s}_1]\!](\sigma) + [\![\mathtt{s}_2]\!](\sigma) & \text{si } [\![\mathtt{s}_1]\!](\sigma) \in \mathbb{Z} \text{ et } [\![\mathtt{s}_2]\!](\sigma) \in \mathbb{Z} \\ \mathtt{err} & \text{sinon} \end{cases} \\
 & & \text{et de même pour les sept autres opérations binaires}
\end{array}
$$

FIG. 4.3 – *Sémantique des instructions*

$$
\begin{array}{rcl}
[\![\mathtt{x\ :=\ e}]\!](\sigma) & = & \sigma[\mathtt{x} \leftarrow [\![\mathtt{e}]\!](\sigma)] \\
[\![\mathtt{*x\ :=\ e}]\!](\sigma) & = & \begin{cases} \sigma[\sigma(\mathtt{x}) \leftarrow [\![\mathtt{e}]\!](\sigma)] & \text{si } \sigma(\mathtt{x}) \in V \\ \mathtt{err} & \text{sinon} \end{cases} \\
[\![\mathtt{i}_1\mathtt{;\ i}_2]\!](\sigma) & = & [\![\mathtt{i}_2]\!]([\![\mathtt{i}_1]\!](\sigma)) \\
[\![\mathtt{if\ b\ then\ (i}_1\mathtt{)\ else\ (i}_2\mathtt{)}]\!](\sigma) & = & \begin{cases} [\![\mathtt{i}_1]\!](\sigma) & \text{si } [\![\mathtt{b}]\!](\sigma) = True \\ [\![\mathtt{i}_2]\!](\sigma) & \text{si } [\![\mathtt{b}]\!](\sigma) = False \\ \mathtt{err} & \text{sinon (par exemple quand } [\![\mathtt{b}]\!](\sigma) \in V) \end{cases} \\
[\![\mathtt{while\ b\ do\ (i)}]\!](\sigma) & = & \begin{cases} \sigma & \text{si } [\![\mathtt{b}]\!](\sigma) = False \\ [\![\mathtt{while\ b\ do\ (i)}]\!]([\![\mathtt{i}]\!](\sigma)) & \text{si } [\![\mathtt{b}]\!](\sigma) = True \\ \mathtt{err} & \text{sinon} \end{cases}
\end{array}
$$

pouvant étiqueter un arc du graphe. Ces règles se déduisent principalement de la sémantique des instructions d'affectation. Elles sont données à la figure 4.4.

En fait, nous n'avons fait que définir une sémantique déductive pour notre langage. Elle est similaire à la définition de la plus forte post-condition de Floyd.

On vérifie également au passage que les fonctions $\tau(C)$ sont toutes croissantes comme on l'a supposé depuis le début.

4.3 Premier algorithme et première approximation

Cette fois-ci, nous n'allons pas commencer par définir une relation d'équivalence θ sur $\mathcal{A}$ (ce serait possible, mais cela ne donne pas une vision claire de l'approximation). Nous allons au contraire introduire directement le treillis A ainsi que les fonctions γ et α.

Un des points-clés à résoudre pour obtenir un algorithme utilisable en pratique est de trouver une représentation des informations qui soit finie. Les informations qui nous intéressent dans le cas présent sont des affirmations du type "la variable `x` peut pointer sur la variable `y`". Par conséquent, nous allons représenter l'ensemble de ces informations en un point du programme par un graphe

FIG. 4.4 – *Règles de transformation dans $\mathcal{A}$*

$$
\begin{aligned}
\tau(C_a)(\Sigma) &= \{[\![C_a]\!](\sigma) \;/\; \sigma \in \Sigma\} \quad \text{lorsque } C_a \text{ est une instruction d'affectation}\\
\tau(\mathtt{b})(\Sigma) &= \{\sigma \;/\; \sigma \in \Sigma \text{ et } [\![\mathtt{b}]\!](\sigma) = True\} \quad \text{lorsque } \mathtt{b} \text{ est un test booléen}\\
\tau(\mathtt{skip})(\Sigma) &= \Sigma
\end{aligned}
$$

sur V : x peut pointer sur y si et seulement si $(\mathtt{x}, \mathtt{y})$ est un arc de ce graphe. Puisque V est fixé pour un programme donné, nous allons représenter ce graphe par un élément de $G = \mathcal{P}(V^2)$. Un élément générique de G sera noté E ou F. Et pour notre première approximation, nous prendrons $A = G$. On vérifie que $A(\subseteq, \emptyset, V^2, \cup, \cap)$ forme bien un treillis.

4.3.1 L'algorithme

Commençons par écrire les règles de transformation que nous allons utiliser dans A. Ces règles sont données figure 4.5 et elles ont été écrites "de façon intuitive".

FIG. 4.5 – *Règles de transformation dans A*

$$
\begin{aligned}
t(\mathtt{x\ :=\ y})(E) &= (E \setminus (\{\mathtt{x}\} \times V)) \ \cup\ \{(\mathtt{x}, \mathtt{z}) \;/\; (\mathtt{y}, \mathtt{z}) \in E\}\\
t(\mathtt{x\ :=\ v})(E) &= (E \setminus (\{\mathtt{x}\} \times V)) \quad \text{pour } \mathtt{v} \in Value\\
t(\mathtt{x\ :=\ s_1\ +\ s_2})(E) &= (E \setminus (\{\mathtt{x}\} \times V)) \quad \text{et de même pour les autres opérations binaires}\\
t(\mathtt{x\ :=\ \&y})(E) &= (E \setminus (\{\mathtt{x}\} \times V)) \ \cup\ \{(\mathtt{x}, \mathtt{y})\}\\
t(\mathtt{x\ :=\ *y})(E) &= \begin{cases} \mathtt{err} & \text{si } succ(\mathtt{y}) = \emptyset \\ (E \setminus (\{\mathtt{x}\} \times V)) \ \cup\ \{(\mathtt{x}, \mathtt{z'}) \;/\; (\mathtt{y}, \mathtt{z}) \in E \text{ et } (\mathtt{z}, \mathtt{z'}) \in E\} & \text{si } succ(\mathtt{y}) \neq \emptyset \end{cases}\\
t(\mathtt{*x\ :=\ e})(E) &= \begin{cases} \mathtt{err} & \text{si } succ(\mathtt{x}) = \emptyset \\ \displaystyle\bigcup_{(\mathtt{x},\mathtt{y}) \in E} t(\mathtt{y\ :=\ e})(E) & \text{si } succ(\mathtt{x}) \neq \emptyset \end{cases}\\
t(\mathtt{b})(E) &= E \text{ pour tout test booléen } \mathtt{b}\\
t(\mathtt{skip})(E) &= E
\end{aligned}
$$

Puis, on calcule la suite suivante (il s'agit en fait de la suite (x_n) définie au paragraphe 2.3, avec $\phi_0 = \bot = \emptyset$) :

$$
\begin{cases}
P^0 = (\emptyset, \ldots, \emptyset)\\
P_j^{n+1} = \displaystyle\bigcup_{i \in pred(j)} t(C_{i,j})(P_i^n)
\end{cases}
$$

On s'arrête lorsqu'on a trouvé n tel que $P^{n+1} = P^n$. Les fonctions $t(C)$ sont toutes croissantes et P^0 est minimal, donc la suite P^n est croissante. Comme A est fini (puisque V l'est), l'algorithme s'arrête bien sur le plus petit point fixe de $F(t, P^0)$ (d'après le théorème 2.6).

(Le code source de cet algorithme est donné en annexe page 32).

4.3.2 Relations entre $\mathcal{A}$ et A

Soit $\alpha \ : \ Mem \rightarrow A$ définie par :

$$\alpha(\sigma) = \{(\mathtt{x}, \mathtt{y}) \ / \ \mathtt{x} \in V, \mathtt{y} \in V \text{ et } \sigma(\mathtt{x}) = \mathtt{y}\}$$

$\alpha(\sigma)$ est le graphe qui représente l'état mémoire σ. On remarque que de tout sommet de ce graphe ne peut partir au plus qu'un seul arc.

Puis on étend α en $\alpha \ : \ \mathcal{A} \rightarrow A$ par :

$$\alpha(\Sigma) = \bigcup_{\sigma \in \Sigma} \alpha(\sigma)$$

$\alpha(\Sigma)$ donne la représentation abstraite de l'assertion Σ. On remarque que cette fois-ci $\alpha(\Sigma)$ peut être un graphe quelconque. En fait, α est même surjective : pour tout $(\mathtt{x}, \mathtt{y}) \in V^2$, soit

$$\sigma_{(\mathtt{x},\mathtt{y})} = \begin{cases} \mathtt{x} \longmapsto \mathtt{y} \\ \mathtt{z} \longmapsto \perp \text{ pour } \mathtt{z} \neq \mathtt{x} \end{cases}$$

Alors, $\alpha(\sigma_{(\mathtt{x},\mathtt{y})}) = \{(\mathtt{x}, \mathtt{y})\}$. Donc,

$$\begin{aligned} \alpha(\{\sigma_{(\mathtt{x},\mathtt{y})} \ / \ (\mathtt{x}, \mathtt{y}) \in E\}) &= \bigcup_{(\mathtt{x},\mathtt{y}) \in E} \alpha(\sigma_{(\mathtt{x},\mathtt{y})}) \\ &= \bigcup_{(\mathtt{x},\mathtt{y}) \in E} \{(\mathtt{x}, \mathtt{y})\} \\ &= E \end{aligned}$$

Définissons maintenant la fonction $\gamma \ : \ A \rightarrow \mathcal{A}$ par :

$$\gamma(E) = \{\sigma \ / \ \alpha(\sigma) \subseteq E\}$$

On pose : $\overline{A} = \gamma(A)$. Pour vérifier que γ est un isomorphisme entre $\overline{A}$ et A, il suffit de vérifier que γ est injective. Supposons $\gamma(E) = \gamma(F)$. Soit $(\mathtt{x}, \mathtt{y}) \in E$. Alors, $\sigma_{(\mathtt{x},\mathtt{y})} \in \gamma(E) = \gamma(F)$ et donc $(\mathtt{x}, \mathtt{y}) \in F$. De même, on montrerait $F \subseteq E$, et donc $E = F$.

$< \alpha, \gamma >$ forme une paire de fonctions adjointes entre $\mathcal{A}$ et A. En effet, on a :

$$\begin{aligned} \Sigma \subseteq \gamma(E) &\Leftrightarrow \forall \sigma \in \Sigma, \ \sigma \in \gamma(E) \\ &\Leftrightarrow \forall \sigma \in \Sigma, \ \alpha(\sigma) \subseteq E \\ &\Leftrightarrow \alpha(\Sigma) \subseteq E \end{aligned}$$

Montrons également une propriété qui va nous être utile par la suite : γ est un morphisme pour $\cap$. En effet :

$$\begin{aligned} \gamma\left(\bigcap_{E \in S} E\right) &= \left\{\sigma \ / \ \alpha(\sigma) \subseteq \bigcap_{E \in S} E\right\} \\ &= \{\sigma \ / \ \forall E \in S, \ \alpha(\sigma) \subseteq E\} \\ &= \bigcap_{E \in S} \{\sigma \ / \ \alpha(\sigma) \subseteq E\} \\ &= \bigcap_{E \in S} \gamma(E) \end{aligned}$$

Il nous reste à définir l'opérateur d'approximation ρ. Utilisons la formule du théorème 3.2 :

$$\begin{aligned}
\rho(\Sigma) &= \bigcap_{\substack{\Sigma' \in \overline{A} \\ \Sigma \subseteq \Sigma'}} \Sigma' = \bigcap_{\substack{E \in A \\ \Sigma \subseteq \gamma(E)}} \gamma(E) \\
&= \gamma\left(\bigcap_{\substack{E \in A \\ \Sigma \subseteq \gamma(E)}} E \right) \\
&= \gamma\left(\bigcap_{\substack{E \in A \\ \alpha(\Sigma) \subseteq E}} E \right) \\
&= \gamma(\alpha(\Sigma))
\end{aligned}$$

On retrouve bien la relation : $\rho = \gamma \circ \alpha$.

Et, puisque $\gamma \circ \alpha \circ \gamma = \rho \circ \gamma = \gamma$, on a aussi : $\alpha \circ \gamma = Id_A$.

4.3.3 Etude de l'approximation

Nous allons maintenant étudier la qualité de notre approximation en comparant $t(C)(E)$ et la meilleure approximation $\alpha(\tau(C)(\gamma(E)))$. Pour les premières commandes, nous allons en fait montrer le résultat un peu plus fort : $t(C)(\alpha(\Sigma)) = \alpha(\tau(C)(\Sigma))$.

Pour l'instruction `skip`, c'est immédiat :

$$\alpha(\tau(\texttt{skip})(\Sigma)) = \alpha(\Sigma) = t(\texttt{skip})(\alpha(\Sigma))$$

Pour la suite, nous allons avoir besoin d'un petit calcul préalable :

$$\begin{aligned}
\alpha(\sigma[\texttt{x} \leftarrow \texttt{e}]) &= \begin{cases} \alpha(\sigma) & \text{si } \sigma(\texttt{x}) \notin V \text{ et } \texttt{e} \notin V \\ \alpha(\sigma) \setminus \{(\texttt{x}, \sigma(\texttt{x}))\} & \text{si } \sigma(\texttt{x}) \in V \text{ et } \texttt{e} \notin V \\ \alpha(\sigma) \cup \{(\texttt{x}, \texttt{e})\} & \text{si } \sigma(\texttt{x}) \notin V \text{ et } \texttt{e} \in V \\ \alpha(\sigma) \setminus \{(\texttt{x}, \sigma(\texttt{x}))\} \cup \{(\texttt{x}, \texttt{e})\} & \text{si } \sigma(\texttt{x}) \in V \text{ et } \texttt{e} \in V \end{cases} \\
&= \begin{cases} \alpha(\sigma) \setminus (\{\texttt{x}\} \times V) & \text{si } \texttt{e} \notin V \\ \alpha(\sigma) \setminus (\{\texttt{x}\} \times V) \cup \{(\texttt{x}, \texttt{e})\} & \text{si } \texttt{e} \in V \end{cases}
\end{aligned}$$

car :

- si $\sigma(\texttt{x}) \in V$, $(\texttt{x}, \sigma(\texttt{x}))$ est l'unique couple commençant par `x` dans $\alpha(\sigma)$
- si $\sigma(\texttt{x}) \notin V$, aucun couple dans $\alpha(\sigma)$ ne commence par `x`

Commençons par l'affectation d'une valeur (c'est le cas le plus simple) :

$$\begin{aligned}
\alpha(\tau(\texttt{x := v})(\Sigma)) &= \alpha(\ \{[\![\texttt{x := v}]\!](\sigma) \ / \ \sigma \in \Sigma\}\ ) \\
&= \bigcup_{\sigma \in \Sigma} \alpha([\![\texttt{x := v}]\!](\sigma)) \\
&= \bigcup_{\sigma \in \Sigma} \alpha(\sigma[\texttt{x} \leftarrow \texttt{v}]) \\
&= \bigcup_{\sigma \in \Sigma} (\alpha(\sigma) \setminus (\{\texttt{x}\} \times V)) \\
&= \alpha(\Sigma) \setminus (\{\texttt{x}\} \times V) \\
&= t(\texttt{x := v})(\alpha(\Sigma))
\end{aligned}$$

Le calcul est exactement le même pour les affectations avec une opération binaire du type $\mathtt{x\ :=\ s_1\ +\ s_2}$.

Puis, pour l'affectation de l'adresse d'une variable (le début du calcul est toujours le même) :

$$\begin{aligned}
\alpha(\tau(\mathtt{x\ :=\ \&y})(\Sigma)) &= \bigcup_{\sigma\in\Sigma} \alpha(\sigma[\mathtt{x} \leftarrow \mathtt{y}]) \\
&= \bigcup_{\sigma\in\Sigma} (\alpha(\sigma) \setminus (\{\mathtt{x}\} \times V) \cup \{(\mathtt{x},\mathtt{y})\}) \\
&= \alpha(\Sigma) \setminus (\{\mathtt{x}\} \times V) \cup \{(\mathtt{x},\mathtt{y})\}) \\
&= t(\mathtt{x\ :=\ \&y})(\alpha(\Sigma))
\end{aligned}$$

On continue par l'affectation du contenu d'une variable :

$$\begin{aligned}
\alpha(\tau(\mathtt{x\ :=\ y})(\Sigma)) &= \bigcup_{\sigma\in\Sigma} \alpha(\sigma[\mathtt{x} \leftarrow \sigma(\mathtt{y})]) \\
&= \alpha(\Sigma) \setminus (\{\mathtt{x}\} \times V) \cup \bigcup_{\sigma\in\Sigma\ /\ \sigma(\mathtt{y})\in V} \{(\mathtt{x},\sigma(\mathtt{y}))\}
\end{aligned}$$

Or, on a :

$$\begin{aligned}
\{(\mathtt{x},\mathtt{z})\ /\ (\mathtt{y},\mathtt{z}) \in \alpha(\Sigma)\} &= \left\{(\mathtt{x},\mathtt{z})\ /\ (\mathtt{y},\mathtt{z}) \in \bigcup_{\sigma\in\Sigma} \alpha(\sigma)\right\} \\
&= \bigcup_{\sigma\in\Sigma} \{(\mathtt{x},\mathtt{z})\ /\ (\mathtt{y},\mathtt{z}) \in \alpha(\sigma)\} \\
&= \bigcup_{\sigma\in\Sigma} \{(\mathtt{x},\mathtt{z})\ /\ \sigma(\mathtt{y}) = \mathtt{z} \text{ et } \mathtt{z} \in V\} \\
&= \bigcup_{\sigma\in\Sigma\ /\ \sigma(\mathtt{y})\in V} \{(\mathtt{x},\sigma(\mathtt{y}))\}
\end{aligned}$$

Et donc :

$$\begin{aligned}
\alpha(\tau(\mathtt{x\ :=\ y})(\Sigma)) &= \alpha(\Sigma) \setminus (\{\mathtt{x}\} \times V) \cup \{(\mathtt{x},\mathtt{z})\ /\ (\mathtt{y},\mathtt{z}) \in \alpha(\Sigma)\} \\
&= t(\mathtt{x\ :=\ y})(\alpha(\Sigma))
\end{aligned}$$

Le dernier cas d'affectation simple, celui du contenu d'un pointeur, se démontre de la même façon (à ceci près que le calcul est encore plus long). On obtient bien :

$$\alpha(\tau(\mathtt{x\ :=\ *y})(\Sigma)) = t(\mathtt{x\ :=\ *y})(\alpha(\Sigma))$$

Il ne nous reste plus que deux familles d'instruction à étudier : les tests booléens et les affectations au contenu d'un pointeur (`*x := ...`). Commençons par les tests booléens (on essaie de montrer l'égalité avec la meilleure approximation) :

$$\begin{aligned}
\alpha(\tau(\mathtt{b})(\gamma(E))) &= \alpha(\tau(\mathtt{b})(\ \{\sigma\ /\ \alpha(\sigma) \subseteq E\}\ )) \\
&= \alpha(\ \{\sigma\ /\ \alpha(\sigma) \subseteq E \text{ et } [\![\mathtt{b}]\!](\sigma) = True\}\ ) \\
&= \bigcup_{\sigma/\alpha(\sigma)\subseteq E \text{ et } [\![\mathtt{b}]\!](\sigma)=True} \alpha(\sigma)
\end{aligned}$$

$$
\begin{aligned}
&\subseteq \bigcup_{\sigma/\alpha(\sigma)\subseteq E} \alpha(\sigma) \\
&= E \text{ puisque } \alpha \text{ est surjective} \\
&= t(\mathtt{b})(E)
\end{aligned}
$$

On ne parvient pas à prouver qu'il y a égalité à cause du fait qu'on ne tient pas compte des tests dans notre algorithme. Etant donné que savoir si un test vaut vrai ou faux est un problème indécidable, on a peu d'espoir d'arriver un jour à atteindre la meilleure approximation pour cette instruction.

Exemple 4.1 *On prend* $V = \{\mathtt{x}, \mathtt{y}\}$ *et* $C =$ `(if 1=1 then (skip) else (x := &y))`. *On trouve :*

$$\alpha(\widetilde{\tau}(C)(\gamma(\emptyset))) = \emptyset \;\subset\; \{(\mathtt{x}, \mathtt{y})\} = \widetilde{t}(C)(\emptyset)$$

Il reste un cas à traiter, celui de la modification du contenu d'un pointeur.

$$
\begin{aligned}
t(\mathtt{*x\ :=\ e})(E) &= \bigcup_{(\mathtt{x},\mathtt{y})\in E} t(\mathtt{y\ :=\ e})(E) \\
&= \bigcup_{(\mathtt{x},\mathtt{y})\in E} \alpha(\tau(\mathtt{y\ :=\ e})(\gamma(E))) \quad \text{d'après les calculs précédents} \\
&= \bigcup_{(\mathtt{x},\mathtt{y})\in E} \; \bigcup_{\sigma/\alpha(\sigma)\subseteq E} \alpha(\llbracket \mathtt{y\ :=\ e} \rrbracket(\sigma)) \\
&= \bigcup_{\sigma/\alpha(\sigma)\subseteq E} \; \bigcup_{(\mathtt{x},\mathtt{y})\in E} \alpha(\sigma[\mathtt{y} \leftarrow \llbracket \mathtt{e} \rrbracket(\sigma)]) \\
&\supseteq \bigcup_{\sigma/\alpha(\sigma)\subseteq E} \; \bigcup_{(\mathtt{x},\mathtt{y})\in \alpha(\sigma)} \alpha(\sigma[\mathtt{y} \leftarrow \llbracket \mathtt{e} \rrbracket(\sigma)]) \\
&= \bigcup_{\sigma/\alpha(\sigma)\subseteq E} \alpha(\sigma[\sigma(\mathtt{x}) \leftarrow \llbracket \mathtt{e} \rrbracket(\sigma)]) \\
&= \bigcup_{\sigma\in\gamma(E)} \alpha(\llbracket \mathtt{*x\ :=\ e} \rrbracket(\sigma)) \\
&= \alpha(\tau(\mathtt{*x\ :=\ e})(\gamma(E)))
\end{aligned}
$$

A nouveau, on ne parvient pas à obtenir l'égalité. Et on ne pourra l'obtenir par aucun calcul d'après le contre-exemple suivant :

Exemple 4.2 *On prend :* $V = \{\mathtt{x}, \mathtt{y}\}$, $E = \{(\mathtt{x}, \mathtt{x}), (\mathtt{x}, \mathtt{y})\}$, $C =$ `(*x := 0)` *(*E *peut par exemple être obtenu par le programme :* `(if x=y then (x := &x) else (x := &y))`*). D'un côté :*

$$t(C)(E) = t(\mathtt{x\ :=\ 0})(E) \cup t(\mathtt{y\ :=\ 0})(E) = \emptyset \cup E = E$$

D'un autre côté, comme dans un état mémoire, une variable ne peut pointer que sur au plus une autre variable, $\gamma(E)$ *se compose de trois types d'états mémoire :*

$$
\begin{aligned}
\gamma(E) \;=\; & \{\sigma \;/\; \sigma(\mathtt{x}) = \mathtt{x} \text{ et } \sigma(\mathtt{y}) \in Value\} \\
& \cup \{\sigma \;/\; \sigma(\mathtt{x}) = \mathtt{y} \text{ et } \sigma(\mathtt{y}) \in Value\} \\
& \cup \{\sigma \;/\; \sigma(\mathtt{x}), \sigma(\mathtt{y}) \in Value\}
\end{aligned}
$$

Pour σ dans chacun de ces trois ensembles, on trouve respectivement :

$$\alpha(\llbracket C \rrbracket (\sigma)) = \begin{cases} \emptyset \\ \{(\mathtt{x}, \mathtt{y})\} \\ \emptyset \end{cases}$$

Et donc :

$$\alpha(\tau(C)(\gamma(E))) = \{(\mathtt{x}, \mathtt{y})\} \;\subset\; E = t(C)(E)$$

Nous échouons donc à obtenir la meilleure approximation à cause de deux types d'instruction. Cependant, tout n'est pas perdu. Nous avons quand même montré que pour toute commande C :

$$\alpha(\tau(C)(\gamma(E))) \;\subseteq\; t(C)(E)$$

Et donc, d'après le théorème 3.6, la fonction t est une approximation supérieure correcte de τ. Ceci prouve que notre algorithme est correct et qu'il est "sûr" en ce sens qu'il calcule bien une approximation supérieure.

4.4 Deuxième algorithme et deuxième approximation

Revenons sur le problème de l'approximation. Si nous ne pouvons pas faire grand chose dans le cas des tests booléens, nous pouvons par contre résoudre celui des affectations au contenu d'un pointeur. Les problèmes apparaissent lorsqu'on produit des graphes réunions d'autres graphes, de sorte que `x` puisse pointer sur plusieurs variables (après un `if` par exemple). Par conséquent, nous allons remplacer cette réunion "physique" $\cup$ par une réunion "symbolique" $\sqcup$: l'union de plusieurs graphes sera tout simplement le sous-ensemble de G contenant ces graphes. Et, dans un deuxième temps, il suffira de prendre l'union physique pour retomber sur un graphe unique.

Posons donc $A_1 = \mathcal{P}(G)$ et $A_2 = A = G$. Un élément générique de A_1 sera noté $\mathcal{E}$. Voici la définition précise de la notation union symbolique :

$$\bigsqcup_{E \in \mathcal{E}} f(E) = \{f(E) \;/\; E \in \mathcal{E}\}$$

Nous avons préféré cette notation à la notation ensembliste directe, en raison des fortes similitudes entre les propriétés algébriques de $\cup$ et $\sqcup$.

On va établir les deux approximations suivantes entre $\mathcal{A}$ et A_2 :

$$\mathcal{A} \;\underset{\alpha_1}{\overset{\gamma_1}{\rightleftarrows}}\; A_1 \;\underset{\alpha_2}{\overset{\gamma_2}{\rightleftarrows}}\; A_2 = A$$

On définit :

$$\alpha_1(\Sigma) = \{\alpha(\sigma) \;/\; \sigma \in \Sigma\} = \alpha(\Sigma)$$

$$\alpha_2(\mathcal{E}) = \bigcup_{E \in \mathcal{E}} E$$

$$\gamma_2(E) = \{F \in G \;/\; F \subseteq E\}$$

$$\gamma_1(\mathcal{E}) = \{\sigma \;/\; \alpha(\sigma) \in \mathcal{E}\} = \alpha^{-1}(\mathcal{E})$$

Bien sûr, par composition, on retrouve les fonctions précédentes :

$$\begin{array}{rcl}
\alpha_2(\alpha_1(\Sigma)) & = & \bigcup \{\alpha(\sigma) \ / \ \sigma \in \Sigma\} \\
\alpha_2 \circ \alpha_1 & = & \alpha \\
\gamma_1(\gamma_2(E)) & = & \{\sigma \ / \ \alpha(\sigma) \in \gamma_2(E)\} \\
 & = & \{\sigma \ / \ \alpha(\sigma) \subseteq E\} \\
\gamma_1 \circ \gamma_2 & = & \gamma
\end{array}$$

Par ailleurs, on a :

$$\begin{array}{rcl}
\Sigma \subseteq \gamma_1(\mathcal{E}) & \Leftrightarrow & \forall \sigma \in \Sigma, \ \sigma \in \gamma_1(\mathcal{E}) \\
 & \Leftrightarrow & \forall \sigma \in \Sigma, \ \alpha(\sigma) \in \mathcal{E} \\
 & \Leftrightarrow & \alpha_1(\Sigma) \subseteq \mathcal{E}
\end{array}$$

Et :

$$\begin{array}{rcl}
\mathcal{E} \subseteq \gamma_2(E) & \Leftrightarrow & \forall F \in \mathcal{E}, \ F \subseteq E \\
 & \Leftrightarrow & \bigcup_{F \in \mathcal{E}} F \subseteq E \\
 & \Leftrightarrow & \alpha_2(\mathcal{E}) \subseteq E
\end{array}$$

Et, comme α_1, α_2, γ_1 et γ_2 sont visiblement croissantes, $< \alpha_1, \gamma_1 >$ et $< \alpha_2, \gamma_2 >$ sont des paires de fonctions adjointes, respectivement entre $\mathcal{A}$ et A_1, et entre A_1 et A_2.

4.4.1 L'algorithme

Le schéma d'algorithme reste le même ; seules les règles de transformation changent. Ces règles sont données à la figure 4.6.

FIG. 4.6 – *Règles de transformation dans A_1*

$$\begin{array}{rcl}
t_1(\texttt{x := e})(\mathcal{E}) & = & \bigsqcup_{E \in \mathcal{E}} t(\texttt{x := e})(E) \\
t_1(\texttt{*x := e})(\mathcal{E}) & = & \bigsqcup_{E \in \mathcal{E}} \ \bigsqcup_{(\texttt{x},\texttt{y}) \in E} t(\texttt{y := e})(E) \\
t_1(\texttt{b})(\mathcal{E}) & = & \mathcal{E} \ \text{ pour tout test booléen } \texttt{b} \\
t_1(\texttt{skip})(\mathcal{E}) & = & \mathcal{E}
\end{array}$$

Ensuite, pour obtenir le résultat dans A_2, il suffit de prendre la fonction α_2, qui est tout à fait utilisable en pratique.

4.4.2 Etude de l'approximation

Nous avons cette fois-ci deux approximations à étudier. Commençons par la deuxième : nous allons montrer que t est la meilleure approximation supérieure correcte de t_1.

$$\begin{aligned}
\alpha_2(t_1(\texttt{x := e})(\gamma_2(E))) &= \bigcup \bigsqcup_{F\subseteq E} t(\texttt{x := e})(F) \\
&= \bigcup_{F\subseteq E} t(\texttt{x := e})(F) \\
&= t(\texttt{x := e})(E) \quad \text{car } t(\texttt{x := e}) \text{ est croissante}
\end{aligned}$$

$$\begin{aligned}
\alpha_2(t_1(\texttt{*x := e})(\gamma_2(E))) &= \bigcup \bigsqcup_{F\subseteq E} \bigsqcup_{(\texttt{x},\texttt{y})\in F} t(\texttt{y := e})(F) \\
&= \bigcup_{F\subseteq E} \bigcup_{(\texttt{x},\texttt{y})\in F} t(\texttt{y := e})(F) \\
&= \bigcup_{F\subseteq E} t(\texttt{*x := e})(F) \\
&= t(\texttt{*x := e})(E)
\end{aligned}$$

$\alpha_2(t_1(\texttt{b})(\gamma_2(E))) = \alpha_2(\gamma_2(E)) = E = t(\texttt{b})(E)$ pour tout test booléen `b` et de même pour `skip`

Ceci prouve qu'on ne peut pas faire mieux dans le passage de A_1 à A_2.

Pour la première approximation, nous allons montrer la condition plus forte que nous avions donnée plus haut :

$$\begin{aligned}
t_1(\texttt{x := e})(\alpha_1(\Sigma)) &= \bigsqcup_{E\in\alpha_1(\Sigma)} t(\texttt{x := e})(E) \\
&= \bigsqcup_{\sigma\in\Sigma} t(\texttt{x := e})(\alpha(\sigma)) \\
&= \bigsqcup_{\sigma\in\Sigma} \alpha(\llbracket \texttt{x := e} \rrbracket (\sigma)) \\
&= \alpha_1(\tau(\texttt{x := e})(\Sigma))
\end{aligned}$$

Et maintenant, le cas qui nous avait posé problème :

$$\begin{aligned}
t_1(\texttt{*x := e})(\alpha_1(\Sigma)) &= \bigsqcup_{E\in\alpha_1(\Sigma)} \bigsqcup_{(\texttt{x},\texttt{y})\in E} t(\texttt{y := e})(E) \\
&= \bigsqcup_{\sigma\in\Sigma} \bigsqcup_{(\texttt{x},\texttt{y})\in\alpha(\sigma)} t(\texttt{y := e})(\alpha(\sigma)) \\
&= \bigsqcup_{\sigma\in\Sigma} \bigsqcup_{(\texttt{x},\texttt{y})\in\alpha(\sigma)} \alpha(\llbracket \texttt{y := e} \rrbracket (\sigma)) \\
&= \bigsqcup_{\sigma\in\Sigma} \bigsqcup_{(\texttt{x},\texttt{y})\in\alpha(\sigma)} \alpha(\sigma[\texttt{y} \leftarrow \llbracket \texttt{e} \rrbracket (\sigma)]) \\
&= \bigsqcup_{\sigma\in\Sigma \text{ et } \sigma(\texttt{x})\in V} \alpha(\sigma[\sigma(\texttt{x}) \leftarrow \llbracket \texttt{e} \rrbracket (\sigma)]) \\
&= \bigsqcup_{\sigma\in\Sigma} \alpha(\llbracket \texttt{*x := e} \rrbracket (\sigma)) \\
&= \alpha_1(\tau(\texttt{*x := e})(\Sigma))
\end{aligned}$$

Cette fois-ci, on obtient l'égalité. Par contre, pour les tests booléens, on continue évidemment à n'avoir qu'une inclusion :

$$\alpha_1(\tau(\mathtt{b})(\Sigma)) \subseteq \alpha_1(\Sigma) = t_1(\mathtt{b})(\alpha_1(\Sigma))$$

t_1 est donc une approximation supérieure correcte de τ dans A_1. Mais on peut dire qu'elle s'approche plus de la meilleure approximation que t puisqu'on a obtenu l'égalité pour toutes les commandes sauf les tests, alors que ce n'était pas le cas pour t.

Exemple 4.3 *Reprenons l'exemple 4.2.*

$$\gamma_2(E) = \{\emptyset, \{(\mathtt{x},\mathtt{x})\}, \{(\mathtt{x},\mathtt{y})\}, E\}$$

$$\begin{array}{rcl} t_1(C)(\gamma_2(E)) & = & \emptyset \sqcup \emptyset \sqcup \{(\mathtt{x},\mathtt{y})\} \sqcup (\emptyset \sqcup \{(\mathtt{x},\mathtt{y})\}) \\ & = & \{\emptyset, \{(\mathtt{x},\mathtt{y})\}\} \end{array}$$

Et donc,

$$\alpha_2(t_1(C)(\gamma_2(E)) = \{(\mathtt{x},\mathtt{y})\} = \alpha(\tau(C)(\gamma(E)))$$

Cet exemple prouve qu'il existe des cas pour lesquels le deuxième algorithme donne des résultats plus précis que le premier. Il faut cependant apporter un bémol à cette avancée et considérer la complexité des deux algorithmes. Notons n le cardinal de V et p le nombre de sommets dans le graphe de flot. Le premier algorithme manipule des éléments strictement croissants de G^p, soit une complexité au pire en $\mathcal{O}(n^{2p})$. Le second algorithme manipule quant à lui des éléments de $(\mathcal{P}(G))^p$, soit une complexité au pire en $2^{n^2 p}$. Ce deuxième algorithme est donc bien plus lent, et il n'est pas difficile de concevoir un exemple de seulement quelques lignes pour lequel il devient totalement inutilisable. C'est pourquoi nous avons également présenté le premier algorithme, puisque c'est le seul utilisable en pratique.

Exemple 4.4 *Considérons le programme suivant (un peu surréaliste, j'en conviens) :*

```
(while 1=1 do
 (if 1=1 then (a := &a) else (a := b);
  if 1=1 then (b := &b) else (b := c);
  if 1=1 then (c := &c) else (c := d);
  if 1=1 then (d := &d) else (d := e);
  if 1=1 then (e := &e) else (e := a)
 )
)
```

Son analyse par le premier algorithme met moins d'une seconde, tandis que le deuxième algorithme prend presque une minute. Il suffit d'écrire un programme similaire avec 8 ou 10 variables au lieu de 5 pour être sûr que le deuxième algorithme ne terminera "jamais" (au sens informatique).

En fait, ce programme a été conçu de telle sorte que chacune des variables peut ou non pointer sur une autre. Tous les graphes de G sont donc des représentations correctes d'une exécution possible. Par conséquent, le deuxième algorithme doit générer tous les éléments de $\mathcal{P}(G)$, tandis que le premier ne génère que ceux de G.

4.5 Une extension : l'instruction `new`

Notre langage est plutôt pauvre pour l'instant. Nous allons lui ajouter une nouvelle possibilité : l'instruction `new`. L'effet de cette instruction est d'allouer un nouvel emplacement non encore utilisé dans la mémoire, de l'initialiser avec la valeur donnée en paramètre et de retourner un pointeur sur cet emplacement. On supposera donc en théorie que V est infini et qu'il est toujours possible de trouver un $\mathtt{z} \in V$ tel que $\sigma(\mathtt{z}) = \perp$ (i.e. `z` n'a jamais été utilisé) pour n'importe quel $\sigma \in Mem$.

Nous rajoutons donc la construction suivante dans la syntaxe BNF de la figure 4.1 :

$$Instr \quad ::= \quad \ldots \mid Loc \ \mathtt{:= new}\ Simple$$

Et la sémantique suivante dans la figure 4.3 :

$$\begin{array}{rcl} [\![\mathtt{x\ :=\ new\ s}]\!]\,(\sigma) & = & \sigma[\mathtt{x} \leftarrow \mathtt{z}, \mathtt{z} \leftarrow [\![\mathtt{s}]\!]\,(\sigma)] \ \text{ avec } \mathtt{z} \in V \ / \ \sigma(\mathtt{z}) = \perp \\ [\![\mathtt{*x\ :=\ new\ s}]\!]\,(\sigma) & = & \left\{ \begin{array}{ll} \sigma[\sigma(\mathtt{x}) \leftarrow \mathtt{z}, \mathtt{z} \leftarrow [\![\mathtt{s}]\!]\,(\sigma)] \ \text{ avec } \mathtt{z} \in V \ / \ \sigma(\mathtt{z}) = \perp & \text{si } \sigma(\mathtt{x}) \in V \\ \mathtt{err} & \text{sinon} \end{array} \right. \end{array}$$

Voyons maintenant comment effectuer l'analyse d'alias avec cette nouvelle instruction. La première solution qui vient à l'esprit est d'introduire un nouveau nom de variable à chaque fois qu'on traite une instruction `new`, mais ceci n'est pas possible dès que cette instruction est dans une boucle `while` car V grossirait indéfiniment et l'algorithme ne s'arrêterait jamais.

Nous allons donc appliquer une autre technique d'approximation, appelée la *k-limitation.* Elle consiste à limiter la hauteur des graphes manipulés à une valeur $k \in \mathbb{N}$. En fait, nous allons effectuer un prétraitement syntaxique du programme à analyser en "dépliant" k fois les boucles `while`. Plus précisément, on effectue k fois la transformation suivante :

$$\mathtt{while\ b\ do\ (P)} \longrightarrow \mathtt{if\ b\ then\ (P';\ while\ b\ do\ (P))\ else\ (skip)}$$

où `P'` est la copie de `P` dans laquelle on a remplacé chaque instruction de la forme `(x := new e)` par $(\mathtt{x\ :=\ \&\_new}_i\mathtt{;\ *\_new}_i\mathtt{\ :=\ e})$ avec $\mathtt{\_new}_i$ un nouveau nom de variable (ces noms sont réservés et ne doivent pas être utilisés ailleurs dans le programme ; l'indice i signifie simplement qu'on utilise à chaque fois un nouveau nom de variable, i.e. que toutes les variables $\mathtt{\_new}_i$ doivent être différentes). Ces variables seront traitées comme les autres variables du programme lors de l'analyse de flot.

En revanche, dans le `while` de la $k^{ème}$ branche, on utilise un autre nom de variable réservée (par exemple $\mathtt{\_newk}_i$), ce qui nous donne : $(\mathtt{x\ :=\ \&\_newk}_i\mathtt{;\ *\_newk}_i\mathtt{\ :=\ e})$. En effet, cette variable doit être traitée différemment : il faut la voir comme si elle représentait une infinité de nouvelles variables. Intuitivement, on peut considérer qu'un graphe produit par k-limitation est le résultat d'une analyse avec création d'un nombre infini de nouvelles variables $\mathtt{x}_1$, $\mathtt{x}_2$, ... (en théorie), suivi d'un quotientage du graphe en posant $\mathtt{x}_k = \mathtt{x}_{k+1} = \ldots = \mathtt{x}_n$ pour $n \geq k$. Par conséquent, puisque $\mathtt{x}_k$ ne représente pas seulement $\mathtt{x}_k$ mais aussi d'autres variables, il ne faut pas supprimer les anciennes liaisons lors d'une instruction du type $\mathtt{x}_k$ `:=` .... C'est pourquoi on remplace les règles d'affectation à `x` de la figure 4.5 par les règles suivantes dans le cas particulier où `x` est une variable du type $\mathtt{\_newk}_i$:

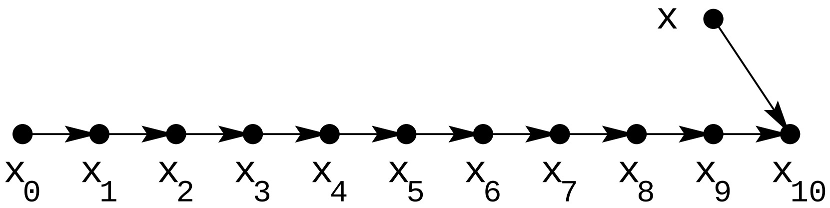
$$\begin{array}{rcl} t(\mathtt{\_newk}_i \mathtt{\ :=\ y})(E) & = & E \cup \{(\mathtt{\_newk}_i, \mathtt{z}) \ / \ (\mathtt{y}, \mathtt{z}) \in E\} \\ t(\mathtt{\_newk}_i \mathtt{\ :=\ v})(E) & = & E \quad \text{pour } \mathtt{v} \in Value \\ t(\mathtt{\_newk}_i \mathtt{\ :=\ s}_1 \mathtt{\ +\ s}_2)(E) & = & E \quad \text{et de même pour les autres opérations binaires} \\ t(\mathtt{\_newk}_i \mathtt{\ :=\ \&y})(E) & = & E \cup \{(\mathtt{\_newk}_i, \mathtt{y})\} \\ t(\mathtt{\_newk}_i \mathtt{\ :=\ *y})(E) & = & \left\{ \begin{array}{ll} \mathtt{err} & \text{si } succ(\mathtt{y}) = \emptyset \\ E \cup \{(\mathtt{\_newk}_i, \mathtt{z'}) \ / \ (\mathtt{y}, \mathtt{z}) \in E \text{ et } (\mathtt{z}, \mathtt{z'}) \in E\} & \text{sinon} \end{array} \right. \end{array}$$

Les autres règles de t et la définition de t_1 sont ensuite les mêmes, puis il suffit d'appliquer les algorithmes précédents pour obtenir les résultats souhaités.

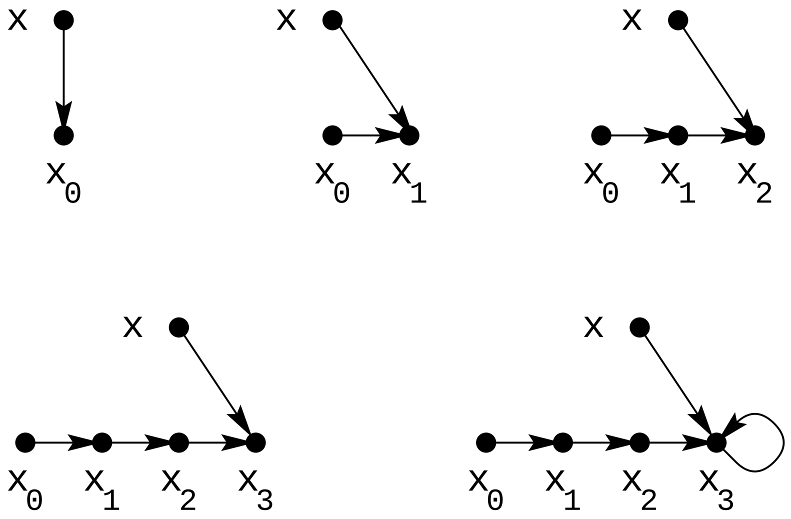
Exemple 4.5 *Considérons le programme C suivant (réaliste cette fois-ci) :*

```
(n := 10;
 x := new 0;
 while n > 0 do
 (n := n - 1;
  *x := new 0;
  x := *x
 );
 *x := 1
)
```

Son rôle est de créer une chaîne de 11 nouvelles variables qui pointent les unes sur les autres en chaîne. A la fin de l'exécution réelle, le graphe des relations entre variables aura l'aspect suivant :



Mais, comme il est impossible pour l'analyse de flot de prévoir à l'avance le nombre de passages dans la boucle `while` *et donc le nombre de nouvelles variables créées (qui pourrait aussi être infini), on est obligé de faire appel à la k-limitation. Avec $k = 3$, le résultat donné par $\widetilde{t_1}(C)(\emptyset)$ est un ensemble de 5 graphes correspondant aux différents cas de figures possibles : 0, 1, 2, 3 ou plus de 3 passages dans la boucle* `while`.



On remarquera la boucle sur $\mathtt{x}_3$ *dans le dernier graphe. Elle traduit le fait que ce $5^{\grave{e}me}$ graphe est la réunion de toutes les graphes correspondant à 3 passages ou plus dans la boucle* `while`.

Chapitre 5

Extensions possibles

Le reproche qu'on puisse faire à cette étude est que le langage utilisé est trop restreint. Cependant, les techniques utilisées ici restent suffisamment générales et sont adaptables à d'autres constructions :

- les tableaux ou, plus généralement, l'arithmétique sur les pointeurs : en analyse de flot, on considère le plus souvent que l'affectation à une case d'un tableau dont on ne connaît pas explicitement l'indice doit se comporter comme l'affectation à toutes les cases du tableau. Par exemple, après l'instruction `tab[i] := &x`, puisqu'on ne peut connaître la valeur de i a priori, on doit considérer que n'importe quelle case de `tab` peut pointer sur `x`.
- les procédures : le graphe de flot de contrôle est formé de l'union des graphes des procédures, et d'arcs supplémentaires correspondant à l'appel et au retour d'une procédure. Le principe est de considérer le *MOP* sur tous les chemins réalisables, correspondant aux appels et aux retours idoines. Pour gérer le mécanisme d'appel, on ajoute à l'état mémoire σ l'état de la pile des appels. Puis on considère alors une abstraction de la pile sous la forme des alias induits par le passage d'arguments à la procédure (pour plus de précisions sur cette technique, cf. [4]).
- les structures de données comme les listes ou les paires : ce cas se résout facilement en étiquetant les arcs du graphe des relations entre variables par les destructeurs à appliquer. Jusqu'à présent, on n'étiquetait pas ces graphes, ou plutôt on les étiquetait implicitement avec le destructeur `*`. Il suffit donc de généraliser aux autres destructeurs pour les autres structures de données. Par exemple, après l'instruction `x := (y,0)`, on doit construire un nouvel arc $\mathtt{x} \overset{\mathtt{fst}}{\longmapsto} \mathtt{y}$, signifiant que `fst(x)` et `y` sont dorénavant synonymes.

Chapitre 6

Conclusion

Nous voici arrivés au bout de notre étude sur l'analyse d'alias. Elle a permis de mettre au point un algorithme détectant automatiquement les chemins d'accès synonymes pour un même emplacement mémoire. Bien que ne couvrant qu'un aspect de la sécurité à garantir lors de l'exécution de codes mobiles, elle permet déjà de s'assurer que les variables de ce code resteront bien dans l'environnement d'exécution qu'on leur a attribué. Pour aboutir à un tel résultat, cette étude a fait la synthèse de deux techniques :

- l'analyse de flot qui, par ses algorithmes généraux issus d'une longue expérience dans le domaine de l'optimisation de code, a permis de mettre facilement au point un algorithme spécialisé pour l'analyse d'alias

- l'interprétation abstraite qui, par son formalisme, a permis à la fois de prouver la sûreté de nos algorithmes mais aussi que ces algorithmes donnaient les meilleures approximations qu'on pouvait espérer (un algorithme d'analyse trivial aurait été de dire que tous les chemins d'accès existant sont synonymes ; on aurait ainsi bien obtenu une approximation supérieure correcte, mais elle aurait été particulièrement mauvaise)

Chapitre 7

Annexes

7.1 Présentation des résultats par les langages rationnels

Lors des tests, il nous est apparu que les graphes sur V étaient commodes pour les manipuler par l'algorithme, mais pas quand il s'agissait de lire les résultats. En revanche, il se trouve que l'ensemble des chemins d'accès à une variable (comme par exemple `x`, `*y` ou `***z`) forme un langage rationnel : il suffit pour s'en convaincre de voir le graphe sur V comme un automate dont toutes les flèches seraient étiquetées par le symbole `*`. Il ne reste alors plus qu'à mettre en oeuvre un algorithme permettant de construire le langage rationnel à partir de son automate. Un tel algorithme est décrit dans [3].

Cet algorithme relève de la programmation dynamique : on calcule un ensemble de langages $R^k_{i,j}$, $1 \leq i, j \leq n$ pour k allant de 1 à n (n est le nombre d'états de l'automate). On veut que :

$$R^k_{i,j} = \left\{ x \in \Sigma^\star \ / \ \overline{\delta}(q_i, x) = q_j \ \begin{array}{l} \text{en ne passant que par des états de numéro} \leq k \\ \text{(sauf } i \text{ et } j \text{ au début et à la fin)} \end{array} \right\}$$

où $\delta : Q \times \Sigma \to Q$ désigne la fonction de transition de l'automate et $\overline{\delta} : Q \times \Sigma^\star \to Q$ son extension aux mots de $\Sigma^\star$.

Pour cela, on calcule successivement pour k de 1 à n :

$$\begin{array}{rcl} R^0_{i,j} & = & \left\{ \begin{array}{ll} \{a \in \Sigma \ / \ \delta(q_i, a) = q_j\} & \text{pour } i \neq j \\ \{a \in \Sigma \ / \ \delta(q_i, a) = q_i\} \cup \{\epsilon\} & \text{pour } i = j \end{array} \right. \\ R^k_{i,j} & = & R^{k-1}_{i,k} (R^{k-1}_{k,k})^\star R^{k-1}_{k,j} \cup R^{k-1}_{i,j} \end{array}$$

Puis, le langage rationnel est donné par (on suppose que l'état 1 est l'entrée de l'automate) :

$$L = \bigcup_{j \ / \ q_j final} R^n_{1,j}$$

puisqu'un chemin passe forcément par des états de numéro $\leq n$.

Cet algorithme a *a priori* une complexité en $\mathcal{O}(n^3)$. Son seul défaut est qu'il introduit de nombreuses constructions inutiles (comme $\epsilon^\star$). Il faut donc utiliser en même temps un algorithme de simplification des formules pour essayer de limiter ce phénomène.

7.2 Le code source

L'intégralité du code source implémentant le petit langage utilisé et l'analyse d'alias est donnée dans les pages suivantes. Le programme est écrit en *Objective Caml* et il a été testé avec la version 1.07. Voici la liste des fichiers, ainsi que leur contenu :

- `loadall.ml` : charge tous les fichiers dans le bon ordre
- `types.ml` : définit les types de données manipulés dans tout le programme
- `set.ml` : implémente une petite bibliothèque de fonctions polymorphes afin de pouvoir manipuler des ensembles (type `set`)
- `utils.ml` : différentes fonctions utilitaires sur les listes ou la syntaxe du langage
- `parser.ml` : le parseur, la fonction essentielle est `read`
- `new.ml` : contient les fonctions de dépliage nécessaires pour l'approximation de l'instruction `new`
- `graph.ml` : la fonction `graph_of_prog` contruit le graphe de flot associé à un programme donné sous la forme d'un arbre syntaxique
- `analyse.ml` : contient les règles de transformation (i.e. les fonctions t, $t_1 \ldots$, un algorithme générique d'analyse de flot (la fonction `analyse`) ainsi que ses versions spécialiées:
  - `analyse12`: applique le premier algorithme d'approximation ; les calculs et le résultat sont dans A
  - `analyse1`: applique le deuxième algorithme d'approximation ; les calculs et le résultat sont dans $\overline{A}$
  - `analyse1then2`: applique d'abord `analyse1` puis α_2 pour obtenir le résultat dans A, ce qui permet de le comparer avec celui de `analyse12`
  - `nanalyse12`, `nanalyse1` et `nanalyse1then2` sont respectivement les versions à utiliser à la place de `analyse12`, `analyse1` et `analyse1then2` pour les programmes contenant des instructions `new`
- `classes.ml` : implémente quelques fonctions sur les langages rationnels et met en oeuvre un algorithme permettant de les construire (cf. plus haut) ; la fonction `print_classes` écrit les langages rationnels correspondant aux chemins d'accès pour chacune des variables de V

La version du programme présentée ici est celle dont le langage contient l'instruction `new`. Les fonctions d'analyse d'alias sont fournies pour les programmes avec et sans `new`. C'est à l'utilisateur de veiller à utiliser la bonne fonction. Il est possible d'utiliser `analyse12` sur un programme avec `new`, mais les résultats ne sont pas garantis. Et si on utilise `nanalyse12` sur un programme ne contenant pas de `new`, le résultat sera correct, mais on aura fait des calculs supplémentaires inutiles. Il aurait bien sûr été possible de donner deux versions du programme, mais cela aurait obligé à fournir deux codes sources peu différents l'un de l'autre...

Bibliographie

[1] A. Aho, R. Sethi, and J. Ullman. *Compilateurs: Principes, techniques et outils*, chapter 10. Addison-Wesley Europe – InterEditions, 1989.

[2] P. Cousot and R. Cousot. Systematic Design of Program Analysis Frameworks. In *6th POPL, San Antonio, Texas*, pages 269–282, January 1979.

[3] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to automata theory, languages and computation*, pages 33–35. Addison-Wesley, 1979.

[4] T. Reps, M. Sagiv, and S. Horwitz. Interprocedural dataflow analysis via graph reachability. Technical Report 94/14, DIKU, University of Copenhagen, Denmark, 1994.