IRISA

INSTITUT DE RECHERCHE
EN INFORMATIQUE
ET SYSTÈMES ALÉATOIRES
Campus Universitaire de Beaulieu
35042 RENNES CEDEX FRANCE
Tél: 02 99 84 71 00 - Télex : UNIRISA 950473 F
Télécopie : 02 99 84 71 71

IFSIC
INSTITUT DE FORMATION SUPÉRIEURE en INFORMATIQUE et COMMUNICATION

Résolution de systèmes d'équations et d'inéquations pour l'analyse de programmes.

Rapport de stage de D.E.A.
Marc Éluard

Projet LANDE

Encadreurs :
Olivier RIDOUX
Thomas JENSEN

Septembre 1998

Remerciements :

Je remercie Olivier Ridoux et Thomas Jensen de m’avoir encadré durant mon stage, de leur disponibilité ainsi que de leur patience.

Je tiens à remercier Daniel Le Métayer pour m’avoir accueilli au sein de l’équipe LANDE.

Je remercie également toute l’équipe LANDE pour son accueil.

Table des matières

Introduction

Ce stage s'inscrit dans le cadre d'un moteur générique pour l'analyse de programmes. Cet axe de recherche du projet LANDE se décompose en deux phases, la première consiste à analyser un programme et à générer un système complexe de relations sur des variables. La deuxième phase est la résolution du système généré. Le stage s'inscrit dans la deuxième phase.

Le but de ce stage est de trouver un algorithme capable de résoudre des systèmes d'équations et d'inéquations. Pour ce faire, nous nous baserons sur des algorithmes de recherche de la plus petite solution par recherche du plus petit point fixe.

Dans le chapitre 1 nous ferons un survol des méthodes d'analyse de programmes, nous présenterons des méthodes de résolution et nous ferons des rappels sur les notions de treillis et sur les fonctions.

Dans le chapitre 2 nous nous intéresserons à l'existence de solutions dans des systèmes composés d'équations et d'inéquations et aussi à leur accessibilité.

Nous présenterons dans le chapitre 3 deux algorithmes de recherche de la plus petite solution d'un système. Le premier est basé sur l'algorithme itératif de Gary Kildall [Kildall 73] et le second utilise les composantes fortement connexes.

Chapitre 1

Bibliographie

Dans ce chapitre, nous ferons un survol sur les méthodes d'analyse de programmes dans la section 1.1, sur les méthodes de résolution de système dans la section 1.3, sur l'utilisation qui en est faite en analyse de programme dans la section 1.4. Nous en profiterons pour faire des rappels sur les treillis et les fonctions dans la section 1.2.

1.1 Introduction

1.1.1 L'origine du problème

1.1.1.1 Les différentes analyses de programmes

L'analyse statique a pour but l'évaluation du comportement des programmes à l'exécution sans les exécuter : il s'agit en fait, de simuler l'exécution du programme, ce qui permet d'obtenir des informations sur, par exemple, le mode d'utilisation du programme, son déterminisme, sa terminaison, sa correction ou l'inférence de type [Aiken et Wimmers 93, Aiken et al. 94].

Les propriétés caractérisant le comportement du programme à l'exécution, obtenues grâce à cette analyse, ont de nombreuses applications : preuve de la correction du programme, optimisation du programme (évaluation partielle) ou aide à la compilation par exemple. Dans le cadre de l'évaluation partielle, de très nombreuses analyses sont utilisées [Hornof 97].

En analyse de programmes, la notion d'approximation de programmes est fondamentale car il est souvent impossible d'obtenir des informations exactes sur le programme. Par exemple, les problèmes les plus intéressants sont très souvent indécidables (déterminer si tel ou tel programme termine ou alors déterminer quelle valeur exacte peut avoir une variable après l'exécution d'un programme...). En faisant une approximation sur cette analyse, il se peut que le problème devienne décidable. Bien qu'il soit impossible de savoir exactement quelle valeur aura une variable après l'exécution d'un programme, il est possible de faire une analyse qui ne donne pas la valeur exacte mais un ensemble de valeurs que la variable peut prendre. C'est ce qui se passe, par exemple, dans le cadre de l'analyse de pointeurs.

Ces notions d'analyse statique et d'approximation ont été formalisées et ont donné naissance à l'interprétation abstraite.

1.1.1.2 L'interprétation abstraite

L'interprétation abstraite consiste à construire une sémantique abstraite à partir d'une sémantique concrète. Ceci est obtenu en faisant d'abord une abstraction sur les données (elles sont remplacées par leurs descriptions), puis une abstraction des opérations sur ces données en des opérations sur leurs abstractions.

De nombreuses interprétations abstraites sont envisageables, chacune se faisant sur un domaine abstrait particulier. Le choix de ce domaine est primordial pour la qualité de l'analyse et se fait en fonction du type de propriété à étudier : le domaine choisi doit pouvoir caractériser correctement la propriété en permettant de ne garder que les aspects des données intéressants à cet égard. Par exemple, l'interprétation abstraite peut être utilisée pour faire une analyse de nécessité [Mycroft 81]. Cette analyse sert à déterminer les expressions et sous-expressions qui seront forcément évaluées dans une sémantique qui modélise l'évaluation paresseuse. Elle permet de gérer au mieux l'appel par nom (pour les variables qui ne sont pas forcément utilisées) et l'appel par valeur (pour les variables qui sont forcément évaluées). Un autre exemple, si nous voulons savoir quel est le signe du résultat d'une addition. Nous abstrayons les entiers par leurs signes. Le domaine abstrait est l'ensemble $\{\bot, zero, neg, pos, \top\}$, il est représenté par le treillis (la définition se trouve page 7) figure 1.1 ($\bot$ nous indique l'absence de valeur alors que $\top$ nous indique que nous ne pouvons pas déterminer le signe de l'expression). La relation de treillis est une relation d'approximation. Plus nous montons dans le treillis moins nous avons d'information.

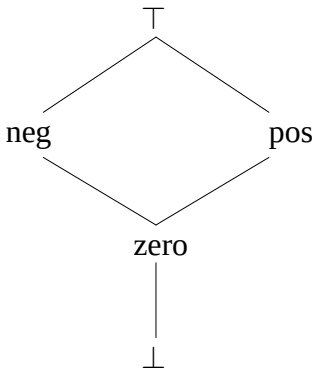

FIG. 1.1 – *Abstraction sur les entiers*

Nous pouvons donner l'abstraction de l'addition par le tableau suivant.

	$\bot$	$zero$	neg	pos	$\top$
$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$
$zero$	$\bot$	$zero$	neg	pos	$\top$
neg	$\bot$	neg	neg	$\top$	$\top$
pos	$\bot$	pos	$\top$	pos	$\top$
$\top$	$\bot$	$\top$	$\top$	$\top$	$\top$

L'abstraction doit être correcte : l'abstraction du résultat doit être incluse ou égale au résultat de l'abstraction. Soit Ξ la fonction d'abstraction pour l'addition, $X + Y = Z$ et $\Xi(X + Y) = Z_a$. Pour cette fonction, il faut que $\Xi(Z) \leq Z_a$. Prenons l'expression $(-6) + (+12)$, nous avons $\Xi((-6) + (+12)) = neg +_a pos = \top$ et $\Xi(+6) = pos$, le critère de correction est vérifié car $pos \leq \top$. Nous remarquons, au passage, le phénomène d'approximation. Nous avons trouvé $\top$ comme résultat alors que la solution était pos.

L'un des avantages de l'interprétation abstraite est de fournir une méthode de spécification des propriétés d'un programme indépendante du langage [Cousot et Cousot 95]. De plus, la méthode de résolution utilisée ne dépend pas de l'interprétation abstraite, il est possible d'utiliser plusieurs méthodes de résolution pour une interprétation abstraite. Il y a plusieurs utilisations possibles de l'interprétation abstraite, par exemple pour l'étude des systèmes de types, la vérification et l'inférence de types. Il est aussi possible de faire de l'interprétation abstraite de structures de contrôle des programmes (fonctions d'ordre supérieur dans les

langages fonctionnels, retour-arrière dans les programmes logiques, mécanismes de communication et de synchronisation dans les programmes parallèles, etc.).

1.1.2 La résolution

Le résultat de ces analyses est un système représentant les relations que doivent satisfaire les variables. Il peut être constitué d'équations [Heintze 92], d'inéquations [Rehof et Mogensen 98] ou alors un système mixte [O'Keefe 87]. Le problème est de résoudre le système ainsi obtenu. Il existe plusieurs méthodes de résolution comme la recherche de point fixe et des méthodes symboliques.

L'analyse statique utilise des ensembles ordonnés, des treillis, des fonctions... Nous ferons quelques rappels sur les notions de treillis, de point fixe, de fonctions... Nous en profiterons pour donner quelques notations dans la section 1.2. Ensuite, nous verrons quelques méthodes pour résoudre un système dans la section 1.3. Nous verrons, sur des exemples, l'utilisation des analyses de programmes ainsi que la résolution des systèmes résultant dans la section 1.4. Nous conclurons dans la section 1.5.

1.2 Rappels et notations

Dans cette section, nous rappelons quelques définitions sur les ensembles, les relations, les treillis, des propriétés sur les fonctions ainsi que les fonctions d'ordre supérieur et nous adoptons des notations que nous garderons tout au long de cette étude [Ross et Wright 88, Abramsky et Hankin 87, Kaufmann et Pichat 77].

1.2.1 Les treillis

Les treillis sont utilisés comme domaine abstrait, nous allons donner quelques définitions ainsi que des notations.

Définition 1.1 (EPO,*POSet*) *On appelle un EPO (Ensemble Partiellement Ordonné), le doublet $(S, \preceq)$ où S est un ensemble et $\preceq$ une relation vérifiant :*

$s \preceq s \; \forall s \in S$	*Réflexive*
$s \preceq t$ *et* $t \preceq s$ *implique* $s = t$	*Anti-symétrique*
$s \preceq t$ *et* $t \preceq u$ *implique* $s \preceq u$	*Transitive*

Exemple 1.1 : Soit $S = \{1, 2, 3, 4, 5, 6, 7, 8\}$, soit $\preceq$ la relation division : $m/n \equiv m$ divise n. Il est facile de montrer les trois propriétés pour la relation division. Il est évident que tous les éléments ne sont pas en relation entre eux (ex. 3 et 5 ne sont pas en relation car 3 ne divise pas 5). Nous obtenons la configuration représentée sur la figure 1.2. •

Définition 1.2 (Sous-EPO) *Soit $(S, \preceq)$ un EPO, $(T, \preceq)$ est un sous-EPO si $T \subseteq S$.*

Exemple 1.2 : Soit $(S, divise)$ l'EPO de la figure 1.2 alors $(T, divise)$ avec $T = \{2, 3, 4, 5, 6\}$ est un sous-EPO. •

Définition 1.3 (Plus grand élément) *Un élément* Max *de S est appelé plus grand élément de S si pour tout s dans S, $s \preceq$* Max.

Définition 1.4 (Plus petit élément) *Un élément* Min *de S est appelé plus petit élément de S si pour tout s dans S,* Min $\preceq s$.

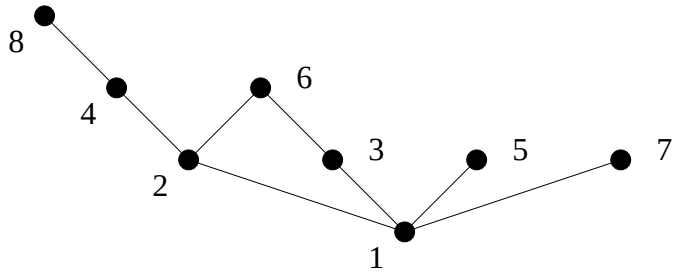

FIG. 1.2 – *Exemple d'EPO*

Exemple 1.3 : Sur la figure 1.2, l'ensemble des *Max* est vide et l'ensemble des *Min* se résume à un seul élément 1. •

Définition 1.5 (Borne supérieure) *Soit P un sous-EPO de $(S, \preceq)$. Il est possible qu'il y ait un élément* M *dans S tel que $p \preceq$* M *pour tout p dans P. Cet élément* M *est appelé borne supérieure de P.*

Définition 1.6 (Borne inférieure) *Soit P un sous-EPO de $(S, \preceq)$ Il est possible qu'il y ait un élément* m *dans S tel que* m $\preceq p$ *pour tout p dans P. Cet élément* m *est appelé borne inférieure de P.*

Exemple 1.4 : Sur la figure 1.2, l'ensemble $\{2,3\}$ n'a pas de plus grand élément mais il possède une borne supérieure qui est 6. De même, il n'a pas de plus petit élément mais il possède une borne inférieure qui est 1. •

Définition 1.7 (Glb (*Greatest Lower Bound*)) *Soit un EPO $(S, \preceq)$, on dit que T, $T \subseteq S$, possède une plus grande borne inférieure, noté glb (T), si quelque soit w une borne inférieure de T on a :*

$$w \preceq \textit{glb}(T).$$

«glb» se prononce «gleub» comme dans «seul».

Définition 1.8 (lub (*Least Upper Bound*)) *Soit un EPO $(S, \preceq)$, on dit que T, $T \subseteq S$, possède une plus petite borne supérieure, noté lub (T), si quelque soit w une borne supérieure de T on ait :*

$$\textit{lub}(T) \preceq w.$$

«lub» se prononce «leub» comme dans «seul».

Notation :

$$\begin{array}{lcccl} \textit{glb}(\{x,y\}) & = & x \sqcap y & \equiv & x \text{ meet } y \quad \forall x, y \in S \\ \textit{lub}(\{x,y\}) & = & x \sqcup y & \equiv & x \text{ join } y \quad \forall x, y \in S \end{array}$$

Exemple 1.5 : Sur la figure 1.3, l'ensemble $\{\{a\},\{c\}\}$ possède deux bornes supérieures qui sont $\{a,c\}$ et $\{a,b,c\}$ mais le *lub*$(\{\{a\},\{c\}\})$ existe et vaut $\{a,c\}$. Alors que sur la figure 1.4, l'ensemble $\{a,b\}$ possède deux bornes supérieures qui sont c et d, mais le *lub* n'existe pas. •

Définition 1.9 (Treillis) *Un EPO $(S, \preceq)$ est un treillis si et seulement si glb$(\{x, y\})$ et lub$(\{x, y\})$ existent quelque soit x et y appartenant à S.*

Exemple 1.6 : Avec la figure 1.2, *lub*$(\{3,5\})$ n'existe pas, donc ce n'est pas un treillis. Alors que la figure 1.3 représente un treillis. •

Définition 1.10 (Sous-treillis) *Le treillis $(S, \preceq)$ est sous-treillis d'un treillis $(T, \preceq)$ si $S \subseteq T$ et quelque soit x et y appartenant à S alors le lub$(\{x, y\})$ et le glb$(\{x, y\})$ dans T, sont aussi dans S.*

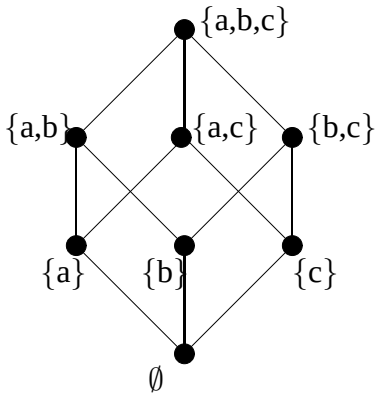

FIG. 1.3 – *Exemple d'ordre partiel qui est un treillis*

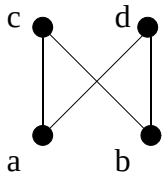

FIG. 1.4 – *Exemple d'ordre partiel qui n'est pas un treillis*

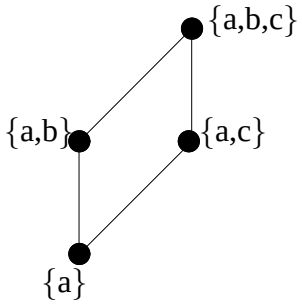

FIG. 1.5 – *Sous-treillis du treillis 1.3*

Exemple 1.7 : Le treillis de la figure 1.5 est un sous-treillis du treillis de la figure 1.3. •

Définition 1.11 (Chaîne) *Soit un EPO $(S, \preceq)$, une chaîne de S est un sous-ensemble C totalement ordonné (c-à-d. $\forall x, y \in C, x \preceq y$ ou $y \preceq x$).*

Exemple 1.8 : Sur la figure 1.2, {2,4,8} est une chaîne (2/4, 4/8 et 2/8 par transitivité), alors que {2,3,6} n'en est pas une (il n'y a pas de relation entre 2 et 3). •

Définition 1.12 (EPOC,*CPOSet*) *Un EPO $(S, \preceq)$ est un EPOC (Ensemble Partiellement Ordonné Complet) si toute chaîne de S possède une plus petite borne supérieure.*

Exemple 1.9 : Dans l'exemple des entiers, il est aisé de voir que tout l'ensemble est une chaîne et qu'il n'existe pas de plus petite borne supérieure. •

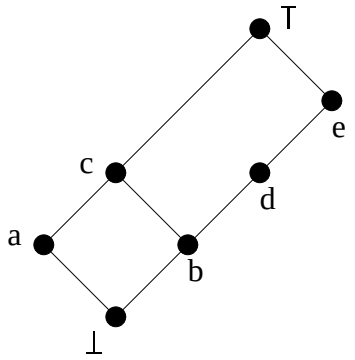

FIG. 1.6 – *Exemple d'EPOC*

Définition 1.13 (Treillis distributif) *Un treillis T est dit distributif si les deux opérations binaires $\sqcup$ et $\sqcap$ sont distributives.*

$$\begin{array}{rcll} x \sqcup (y \sqcap z) & = & (x \sqcup y) \sqcap (x \sqcup z) & \forall x, y, z \in T \\ x \sqcap (y \sqcup z) & = & (x \sqcap y) \sqcup (x \sqcap z) & \forall x, y, z \in T \end{array}$$

Exemple 1.10 : Le treillis $(\mathcal{P}(\{a, b, c\}), \subseteq)$ [1] avec l'union pour *join* et l'intersection pour *meet* est un treillis distributif. •

Définition 1.14 (Borne supérieure universelle) *Soit un treillis T, T possède une borne supérieure universelle $x \in T$ si $\forall y \in T\ y \preceq x$.*

Définition 1.15 (Borne inférieure universelle) *Soit un treillis T, T possède une borne inférieure universelle $z \in T$ si $\forall y \in T\ z \preceq y$.*

Notation : La borne supérieure universelle est appelée *Top* et est notée $\top$. La borne inférieure universelle est appelée *Bottom* et est notée $\bot$.

Exemple 1.11 : L'EPO, représenté figure 1.3, possède les deux bornes universelles, $\{a, b, c\}$ pour $\top$ et $\emptyset$ pour $\bot$. Alors que celui représenté figure 1.2 ne possède pas de Top mais possède un Bottom qui est 1. •

Définition 1.16 (Complémentaire) *Soit un treillis T, soient $x, y \in T$, on dit que x et y sont complémentaires si*

$$\begin{array}{rl} & x \sqcup y = \top \\ \textit{et} & x \sqcap y = \bot \end{array}$$

1. $\mathcal{P}(A)$ représente l'ensemble des parties de A. Le treillis figure 1.3 est le treillis des parties de $\{a, b, c\}$ ($\mathcal{P}(\{a, b, c\})$).

Définition 1.17 (Treillis complémenté) *Un treillis T est dit complémenté si tout élément de T possède au moins un complément.*

Définition 1.18 (Complément unique) *Quand dans un treillis T tout élément x a un complément unique, on note ce complément $\overline{x}$.*

Exemple 1.12 : Le treillis, figure 1.3, avec l'union $\cup$ et l'intersection $\cap$ est un treillis complémenté (ex. $\overline{\{a,b\}} = \{c\}$ car $\{a,b\} \cup \{c\} = \{a,b,c\} = \top$ et $\{a,b\} \cap \{c\} = \emptyset = \bot$). Pour celui, figure 1.6, *a* possède deux compléments *d* et *e*, *e* n'a qu'un seul complément qui est *a* alors que *b* et *c* n'ont aucun complément. •

Définition 1.19 (Treillis booléen) *Un treillis distributif et complémenté est appelé treillis booléen.*

Exemple 1.13 : L'exemple de treillis, figure 1.3, est un treillis booléen. •

Définition 1.20 (Sous-treillis booléen) *Le treillis $(S, \preceq)$ est sous-treillis booléen d'un treillis booléen $(T, \preceq)$ si $(S, \preceq)$ est sous-treillis de $(T, \preceq)$ et si $\bot$ et $\top$ sont dans S.*

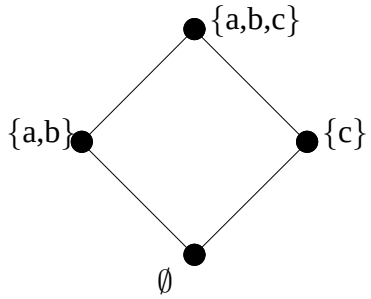

FIG. 1.7 – *Sous-treillis booléen du treillis 1.3*

Exemple 1.14 : Le treillis de la figure 1.7 est un sous-treillis booléen du treillis booléen de la figure 1.3, mais le treillis de la figure 1.5 n'en est pas un. •

Définition 1.21 (Demi-treillis) *Un demi-treillis est un treillis avec une seule des opérations binaires (c-à-d. qui ne possède que le* meet *ou le* join*).*

1.2.2 Les fonctions

Les fonctions sont utilisées pour les opérations abstraites. Voici quelques définitions et notations sur les fonctions.

Définition 1.22 (Monotone) *Soit un EPO $(S, \preceq)$, soit f une fonction de $S \rightarrow S$, f est dite monotone si $\forall a, b \in S, a \preceq b \rightarrow f(a) \preceq f(b)$.*

Exemple 1.15 : Prenons l'exemple de la fonction $f(x) = x^2$ et de l'EPO $(S, \leq)$, avec $S = [0, 1]$ sur les réels. Nous avons bien la monotonie ($\forall a, b \in S, a \leq b \rightarrow f(a) \leq f(b)$). •

Définition 1.23 (Continue) *Soit un EPO $(S, \preceq)$, soit f une fonction de $S \rightarrow S$, f est dite continue (c-à-d. préserve les limites) si pour toute chaîne C de S $f(lub(C)) = lub(f(C))$.*

Exemple 1.16 : Quelque soit l'intervalle $[a, b] \subseteq [0, 1]$, $lub([a, b]) = b$ et $f(b) = b^2$ de même, $f([a, b]) = [a^2, b^2]$ et $lub([a^2, b^2]) = b^2$. Donc $f(x) = x^2$ est une fonction continue sur l'intervalle $[0, 1]$. •

Définition 1.24 (Point fixe) *Soit un EPO $(S, \preceq)$, soit f une fonction de $S \to S$, on appelle point fixe de f un élément $a \in S$ tel que $f(a) = a$.*

Définition 1.25 (Plus petit point fixe) *Soit un EPO $(S, \preceq)$, soit f une fonction de $S \to S$, on appelle plus petit point fixe d de f un élément $d \in S$ tel que $f(d) = d$ et $\forall e \in S, f(e) = e \to d \preceq e$.*

Exemple 1.17 : La fonction $f(x) = x^2$ possède deux points fixes qui sont 0 et 1 ($f(0) = 0$ et $f(1) = 1$). Il en existe un plus petit qui est 0 ($0 \leq 1$). •

Définition 1.26 (Les fonctions d'ordre supérieur) *On appelle fonction d'ordre supérieur une fonction qui admet d'autres fonctions en paramètres ou qui produit une fonction en résultat. Dans le cadre de fonctions typées, il est possible de décrire l'ordre supérieur avec la formulation suivante :*

$$Ord(\tau) = \begin{cases} 0 & \textit{si } \tau \textit{ est un type sans flèche} \\ max(Ord(\alpha) + 1, Ord(\beta)) & \textit{si } \tau = \alpha \to \beta \end{cases}$$

Exemple 1.18 : Utilisation de la notation de types simples (α est un type simple).

Type	Ordre
α	ordre 0
$\alpha \to \alpha \to \alpha$	ordre 1
$\alpha \to (\alpha \to \alpha) \to \alpha$	ordre 2
$((\alpha \to \alpha) \to \alpha) \to \alpha$	ordre 3

Une fonction d'ordre 0 est une simple constante. •

Théorème 1.27 (Tarski) *Soit un treillis complet T $(S, \preceq)$, soit f une fonction de T dans T monotone, il existe un point fixe de f, $pf(f)$ donné par :*

$$pf(f) = \sqcap\{x \in T \mid f(x) \preceq x\}$$

Théorème 1.28 *Soit f une fonction continue, il est possible de caractériser le plus petit point fixe de f par :*

$$pf(f) = \sqcup_i f^i(\bot)$$

Définition 1.29 (Post-Point-Fixe) *Soient $(S, \preceq)$ un ensemble ordonné et $f \in S \to S$, alors l'ensemble des post-points-fixes de f est*

$$postpf(f) = \{x \in L : x \succeq f(x)\}.$$

Définition 1.30 (Pré-Point-Fixe) *Soient $(S, \preceq)$ un ensemble ordonné et $f \in S \to S$, alors l'ensemble des pré-points-fixes de f est*

$$prepf(f) = \{x \in L : x \preceq f(x)\}.$$

1.3 Des méthodes de résolution de systèmes

Nous avons un ensemble de variables $\{f_1, ..., f_n\}$ qui prennent leurs valeurs dans des treillis. Nous avons un ensemble d'expressions $\{e_1, ..., e_n\}$ continues et monotones. Nous allons voir comment résoudre ces systèmes à l'aide du point fixe (section 1.3.1) ou avec des méthodes symboliques (section 1.3.2).

1.3.1 Point fixe

Dans cette section, nous allons voir comment résoudre le problème de la recherche de solution pour un système. Cette résolution se fait par recherche d'un point fixe du système. Nous verrons qu'il est possible d'optimiser cette recherche grâce à différentes techniques. Cette technique est très utilisée par de nombreuses personnes dans de nombreux domaines ([Cousot et Cousot 77, O'Keefe 87, Le Charlier et al. 93]...). Nous nous baserons sur le genre de système suivant :

$$\begin{array}{c} f_1 \Re\, e_1(X_1, ..., X_n) \\ \vdots \\ f_n \Re\, e_n(X_1, ..., X_n) \end{array}$$

Dans la suite, nous verrons la relation $\Re$ comme étant soit la relation $=$, soit la relation $\geq$. Nous utilisons un calcul itératif : les expressions (les e_i) sont évaluées avec les valeurs calculées à l'itération précédente ce qui donne les nouvelles valeurs. Le calcul continue jusqu'a obtention d'un point fixe. Lors d'un calcul ascendant (l'itération part de $\bot$ pour aller vers $\top$) nous écrirons les valeurs prises par les variables $\hat{f}$. Le pas d'itération sera indiqué en exposant sur la variable. $\hat{f}_5^2$ signifie que la cinquième variable est calculée de manière ascendante et que nous sommes au deuxième pas de l'itération.

Jusqu'à maintenant, nous avons sous-entendu que le point fixe est toujours solution du système et surtout qu'il existe toujours un point fixe. Dans le cadre du système décrit précédement, il existe toujours un point fixe et ce point fixe est la plus petite solution du système. Pour des systèmes plus complexes, il n'existe pas toujours de points fixes et de solutions. Dans ce cas, il est possible de rajouter une solution au système [Palsberg et Schwartzbach 91] ou alors de forcer la convergence et d'invalider la solution trouvée [Cousot et Cousot 77]. Nous introduisons les notions de «solution acceptable» et de «solution inacceptable». Un point fixe peut ne pas être une «solution acceptable» du système, soit parce qu'il n'est pas solution, soit parce que cette solution ne nous intéresse pas. Nous donnons une signification d'échec à ces points fixes «inacceptables». Si tous les points fixes atteignables sont des points fixes inacceptables, alors nous concluons qu'il n'y a pas de solution au système.

1.3.1.1 Calcul classique

Pour résoudre ce genre de système, il existe une méthode simple qui est constituée de deux phases. La première phase consiste à initialiser toutes les variables $\hat{f}_i^0$ avec la valeur $\bot$. La deuxième phase consiste à itérer le calcul de toutes les expressions simultanément. Nous appelons ce calcul, méthode des approximations successives. En effet, nous faisons une sous-approximation du plus petit point fixe jusqu'à obtenir ce point fixe. Soit $\hat{\phi}^j$ le vecteur contenant la valeur des $\hat{f}_i^j$, après la première phase nous avons $\hat{\phi}^0 = (\bot, ..., \bot)$ et de façon itérative jusqu'à stabilisation du système, nous calculons $\hat{\phi}^{i+1} = (e_1(\hat{\phi}^i), ..., e_n(\hat{\phi}^i))$. Ce calcul est très simple mais relativement peu efficace (il est possible que nous calculions plusieurs fois une expression alors qu'aucune variable n'a variée). Nous appelons cela un calcul parallèle. Il est plus efficace de faire ce calcul par «itération chaotique» ([Cousot et Cousot 77]). Une première optimisation est triviale, il suffit de ne plus faire le calcul simultané de toutes les expressions, mais de le faire de façon séquentielle. En effet, lors du calcul séquentiel, nous utilisons les valeurs qui viennent d'être recalculées. Par exemple, les équations sont calculées dans l'ordre des indices ($\hat{f}_1^i$ puis $\hat{f}_2^i$ puis...). Dans l'équation $\hat{f}_3 = \hat{f}_1 + \hat{f}_2$, avec un calcul parallèle nous aurions au premier pas d'itération $\hat{f}_3^1 = \hat{f}_1^0 + \hat{f}_2^0 = \bot + \bot$, alors qu'avec un calcul séquentiel nous aurions le calcul de $\hat{f}_3^1$ avec les nouvelles valeurs de $\hat{f}_1^1$ et de $\hat{f}_2^1$.

Tout ceci marche aussi dans le cadre d'un calcul descendant. Il suffit d'initialiser les variables $\check{f}_i^0$ avec la valeur $\top$ et d'itérer en faisant une sur-estimation du plus grand point fixe. Le résultat n'est plus le plus petit point fixe mais le plus grand point fixe.

1.3.1.2 Itération chaotique

L'hypothèse de monotonie permet d'utiliser n'importe quelle stratégie chaotique [Cousot et Cousot 95] : dans l'itération, à condition de respecter le principe d'équité, c-à-d. de ne jamais oublier une expression définitivement, on peut arbitrairement déterminer à chaque pas quelles sont les composantes à calculer en fonction de l'itéré précédent. L'itération chaotique permet d'obtenir la convergence vers le même point fixe qu'avec la méthode des approximations successives. Il existe plusieurs manières de déterminer quelles sont les expressions à recalculer, nous allons en voir quelques-unes.

Nous pouvons utiliser un graphe de dépendances sur les variables. Ce graphe permet de savoir pour n'importe quelle variable f_i, quelles sont les variables qui peuvent influer sur sa valeur. Étant donné la forme du système, les variables intéressantes sont, par exemple, celles qui ont une chance de changer de valeur (ex. si nous avons $f_4 \Re e_4(f_2, f_3)$, il n'est pas intéressant de recalculer f_4 après une variation de f_1, qui n'aura aucun impact). Nous pouvons donc en déduire un premier raffinement qui consiste à ne recalculer que les expressions qui ont vu une de leurs variables subir une variation (ex. on recalcule f_4 à l'itération i seulement si f_2 ou f_3 ont varié depuis la dernière évaluation de f_4). Si une variable n'est pas recalculée au pas i, nous avons $\phi_j^i = \phi_j^{i-1}$ En fait, il suffit de prendre les équations dans l'ordre et de ne recalculer que celles qui en ont besoin. Une question peut quand même se poser. Dans l'exemple, f_4 dépend de f_2 et de f_3, si nous avons en plus $f_3 \Re e_3(..., f_2, ...)$, il serait intéressant de ne recalculer f_4 qu'après avoir recalculé f_2 et f_3. En effet, si nous recalculons f_4 tout de suite après une variation de f_2 et qu'ensuite f_3 varie à son tour il nous faudra recalculer f_4 une deuxième fois, ce qui est une perte de temps. L'idéal, c'est de recalculer une expression seulement lorsque nous sommes sûrs que toutes les variables dont elle dépend ont subi leurs variations.

Un autre problème se pose quand une variable apparaît des deux côtés de l'équation (ex. $f_1 \Re e_1(f_1, f_2, f_6)$). Il est clair qu'une variation de f_1 au pas i risque d'entraîner une variation de f_1 au pas $i+1$; plusieurs choix sont alors possibles. Le premier est de ne pas recalculer tout de suite f_1 mais d'attendre les variations de ses autres composantes (f_2 ou f_6). Il faut tout de même faire attention si aucune de ses dépendances ne varie. Un autre choix consiste à réitérer sur f_1 et de chercher un point fixe partiel, c-à-d. de rechercher un point fixe pour $f_i \Re e_i(f_i, K)$ avec K un contexte constant. Dans un treillis fini, il n'y a aucun problème, ce point fixe existera toujours. Dans le cas d'un treillis infini, il est possible que cette recherche de point fixe local ne se termine jamais, ce qui violerait le principe d'équité. Il est possible de mixer ces deux méthodes. Pour ce faire, il suffit de commencer à rechercher un point fixe local et de s'arrêter au bout de i itérations si nous ne l'avons toujours pas trouvé.

Dans ce cadre de l'itération chaotique, un algorithme a été décrit par Jacob Rehof et Torben Æ. Mogensen [Rehof et Mogensen 98] qui, eux-même, l'attribuent à Gary Kildall [Kildall 73].

ALGORITHME

CS est l'ensemble de toutes les contraintes.
WS est l'ensemble de toutes les contraintes non résolues.
RS est l'ensemble de toutes les contraintes résolues.

Début
  $WS := CS$;
  $RS := \emptyset$;
  Toutes les variables du système ont la valeur $\perp$;
  TantQue $WS \neq \emptyset$ **Faire**
    $(\alpha \Re \beta) :=$ ENLEVER (WS) ;
    VALEUR $(\alpha) :=$ EVALUER (β) ;
    Pour $(\gamma \Re \delta) \in RS$ **Faire**

Si (Non VERIFIER $(\gamma \Re \delta)$**)**
Alors
Début
OTER $(RS, (\gamma \Re \delta))$;
AJOUTER $(WS, (\gamma \Re \delta))$;
Fin
FinSi
FinPour
AJOUTER $(RS, (\alpha \Re \beta))$;
FinTantQue
Fin

Avec :

- ENLEVER (S) : Enlève une relation de l'ensemble S,
- VALEUR (v) : Permet d'avoir la valeur de la variable v,
- EVALUER (E) : Évalue l'expression E,
- VERIFIER (I) : Permet de savoir si la relation I est vérifiée,
- OTER (S, I) : Enlève la relation I de l'ensemble S,
- AJOUTER (S, I) : Ajoute la relation I à l'ensemble S.

FIN

1.3.1.3 L'élargissement et le rétrécissement

L'élargissement (en anglais *widening*) est une méthode permettant d'accélérer la convergence de l'algorithme vers un point fixe en faisant une approximation sur la valeur de l'opérateur *lub* [Cousot et Cousot 92b]. En fait, l'élargissement et le rétrécissement (en anglais *narrowing*) permettent d'accélérer la convergence dans le cadre d'un treillis de hauteur finie, alors que dans le cadre d'un treillis infini, ils permettent d'avoir une terminaison là où l'algorithme naïf ne termine pas. L'élargissement est utilisé avec une recherche ascendante (il fait une approximation supérieure du point fixe, recherche d'un Post-Point-Fixe) alors que le rétrécissement est utilisé avec une recherche descendante (il fait une approximation du point fixe en restant au-dessus de ce point fixe). Le point fixe trouvé n'est pas forcément le plus petit point fixe du système. Pour essayer d'obtenir le plus petit point fixe il faut combiner les deux méthodes : utilisation de l'élargissement pour rechercher un Post-Point-Fixe et ensuite utilisation du rétrécissement pour approximer au mieux le plus petit point fixe.

Comme le montre le schéma 1.8, l'élargissement permet de faire des sauts en avant alors que le rétrécissement permet de faire des sauts en arrière (généralement de plus petits sauts que l'élargissement). De manière plus formelle, l'élargissement peut être défini comme étant un opérateur $\bigtriangledown \in L \times L \rightarrow L$ (L étant un treillis) tel que :

- $\forall x, y \in L : x \preceq x \bigtriangledown y$
- $\forall x, y \in L : y \preceq x \bigtriangledown y$
- Pour toute chaîne croissante $x^0 \preceq x^1 \preceq$ ... la chaîne croissante définie par $y^0 = x^0, ..., y^{i+1} = y^i \bigtriangledown x^{i+1}$ ... est non strictement croissante.

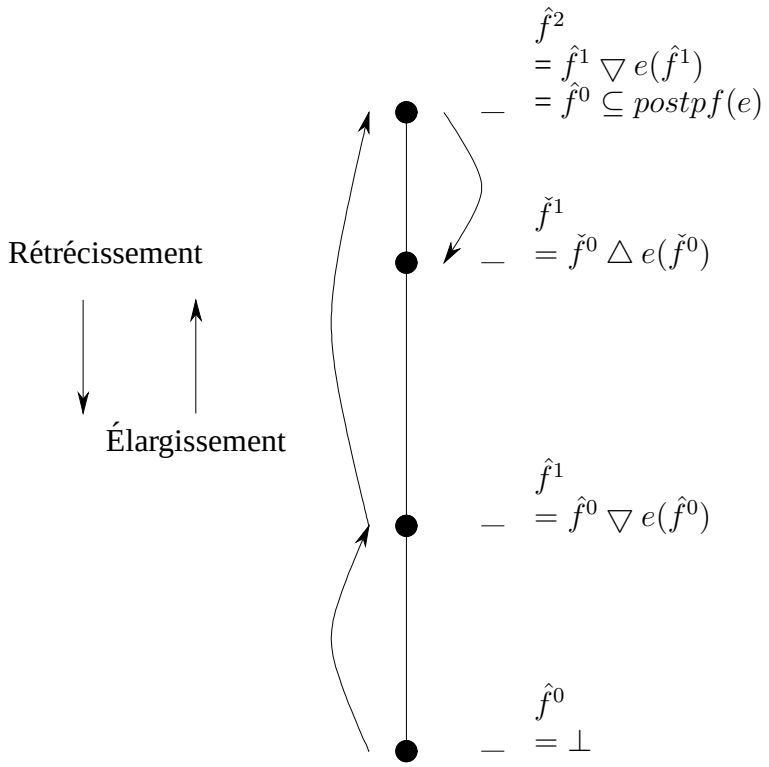

FIG. 1.8 – *Élargissement/Rétrécissement*

Ensuite il suffit de l'intégrer au système en faisant :

$$\begin{array}{rcll} f^0 & = & \bot & \\ f^{i+1} & = & f^i & \text{Si } e(f^i) \preceq f^i \\ & = & f^i \bigtriangledown e(f^i) & \text{Sinon} \end{array}$$

Pour le rétrécissement, il suffit de définir un opérateur $\bigtriangleup \in L \times L \rightarrow L$ (L étant un treillis) tel que :
- $\forall x, y \in L : (y \preceq x) \rightarrow (y \preceq (x \bigtriangleup y) \preceq x)$
- Pour toute chaîne décroissante $x^0 \succeq x^1 \succeq ...$ la chaîne décroissante définie par $y^0 = x^0, ..., y^{i+1} = y^i \bigtriangleup x^{i+1}$ ... est non strictement décroissante.

Ensuite il suffit de l'intégrer au système en faisant :

$$\begin{array}{rcl} f^0 & = & \widehat{A} \\ f^{i+1} & = & f^i \bigtriangleup e(f^i) \end{array}$$

$\widehat{A}$ est une limite supérieure de la chaîne. En fait, on utilise le Post-Point-Fixe trouvé par l'élargissement. Dans le cas d'un treillis de hauteur finie, il est possible d'utiliser directement le rétrécissement sans passer par l'élargissement en donnant la valeur $\top$ à f^0. Un exemple d'élargissement se trouve page 19.

1.3.2 Les méthodes symboliques

Les méthodes symboliques peuvent être vues comme des transformations permettant, à partir d'un système quelconque, d'arriver à un système résolu. Cette transformation peut se faire en transformant incrémentalement le système de contraintes jusqu'à ce que le système ainsi trouvé soit sous une forme

finale ou bien que nous découvrions que le système est non résoluble. Ces transformations se font sur la syntaxe du système, il n'y a pas d'évaluation des différentes expressions. Cette méthode est très dépendante du système à résoudre ainsi que de la forme voulue du résultat.

Pour utiliser cette méthode, nous définissons la forme du système de départ. Ensuite, il faut définir ce qu'est un système final. Un système sera dit final s'il est complètement résolu ou s'il présente les caractéristiques voulues. Il est possible que le but de la résolution d'un problème ne soit pas de donner une valeur exacte à une variable mais une expression dont nous sommes capable de dire quelque chose. Par exemple dans le cadre de la résolution de systèmes algébriques dans lesquels il y a plus de variables que d'équations. Il est impossible de le résoudre entièrement (donner une valeur exacte à chaque variable) et le résultat est alors un système dit résolu dans lequel les variables peuvent dépendre d'autres variables. Il faut faire attention à deux choses : il est possible d'avoir du non-déterminisme (plusieurs règles peuvent être utilisées, laquelle s'applique?) et il est possible de laisser tomber volontairement des solutions pour augmenter la rapidité de la résolution. Il existe une relation d'implication entre les solutions du premier (S) et du second (S') système. Les solutions de S' impliquent les solutions de S.

Prenons un exemple très simple, nous travaillons sur les ensembles avec les seules opérations l'union $\cup$, l'intersection $\cap$ et le complément $\neg$. Une expression de la forme $X \subseteq Y \cap Z$ peut se transformer en $X \subseteq Y$ et $X \subseteq Z$, de même $X \cup Y \subseteq Z$ se transforme en $X \subseteq Z$ et $Y \subseteq Z$. L'expression $X \subseteq Y \cup Z$ pose déjà plus de problèmes, il est possible de la transformer en $X \cap \neg Y \subseteq Z$. $X \cap Y \subseteq Z$ se transforme en $Z \subseteq X$ et $Z \subseteq Y$. Cette dernière règle est un bon exemple de perte d'information. Dans l'équation initiale, Z peut être plus gros que l'intersection de X et Y alors que sa transformée dit que Z est compris dans l'intersection de X et Y. Mais nous avons bien la relation d'implication sur les solutions. Il est bien évident que cet exemple est très simple ; il faut bien voir que nous utilisons une théorie des ensembles très simple. En particulier, le complément peut ne pas exister ou ne pas être représentable (dans un univers infini, le complément d'un ensemble fini est infini).

L'élimination de Gauss peut être vue comme une résolution de système à l'aide d'une règle de transformation. En effet, il s'agit de remplacer l'occurrence d'une variable par l'expression qui lui est associée et ainsi de suite. Cela donne une triangularisation du système ; il suffit ensuite de remonter dans les expressions une par une en les évaluant.

Nous avons vu dans cette section qu'il existait plusieurs méthodes de résolution. Bien qu'elles ne se ressemblent pas, elles ont le même but, résoudre un système. Il est très difficile de dire si une méthode est meilleure qu'une autre, que ce soit du point de vue efficacité ou du point de vue mise en oeuvre. De plus, il n'est pas évident qu'elles soient toutes si différentes. En effet, Thomas Jensen [Jensen 95] a montré que dans le cadre de l'inférence de type de programmes fonctionnels d'ordre supérieur, l'interprétation abstraite et une méthode symbolique avaient la même puissance.

1.4 Les différences selon les langages étudiés

Chaque langage de programmation a ses particularités, une analyse peut n'avoir d'intérêt que pour un seul langage alors que d'autres sont plus «universelles», elles peuvent être utilisées pour plusieurs langages. Nous allons faire un survol des langages de programmation en donnant quelques analyses avec des références significatives et nous allons donner pour chaque type de langage une analyse et la méthode de résolution qui seront plus détaillées. Nous commencerons par les langages impératifs dans le paragraphe 1.4.1, dans le paragraphe 1.4.2 nous parlerons des langages orientés objets, suivront les langages fonctionnels dans le paragraphe 1.4.3 et pour terminer nous parlerons des langages logiques dans le paragraphe 1.4.4.

1.4.1 Les langages impératifs

Il existe beaucoup d'analyses intéressantes pour les langages impératifs. Par exemple l'analyse d'alias, l'analyse de temps de liaison, l'analyse USE-DEF, l'analyse des expressions disponibles, l'analyse des variables actives, la vérification de type, etc. ([Hornof 97, Gouranton 97]...).

Pour faire ces différentes analyses, Patrick Cousot et Radhia Cousot ont proposé une méthode basée sur l'interprétation abstraite [Cousot et Cousot 77]. Une grande partie de ces analyses produit des systèmes d'équations. Selon cette méthode, un programme peut être décomposé en un ensemble de noeuds et chaque noeud possède des prédécesseurs et des successeurs. Cet ensemble de noeuds se décompose en cinq sous-ensembles qui sont : les «entrées», les «affectations», les «tests», les «jonctions» et les «sorties». Les «entrées» ont un successeur et pas de prédécesseur, les «affectations» ont un successeur et un prédécesseur, les «tests» ont deux successeurs et un prédécesseur, les «jonctions» ont un successeur et plus d'un prédécesseur et les «sorties» ont un prédécesseur et pas de successeur. La figure 1.9 donne un exemple d'une décomposition d'un programme.

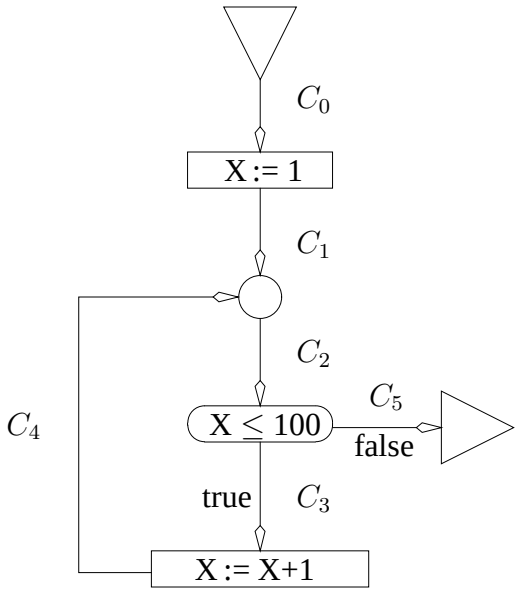

FIG. 1.9 – *Décomposition en noeuds*

Nous introduisons la notion de contexte ; un contexte C_i est l'ensemble des valeurs des variables entre deux noeuds du programme. Le i représente le numéro du lien entre les deux noeuds du programme (à chaque lien est associé un contexte). Le problème se pose en terme de C_i, il faut donc déterminer la valeur de tous les contextes présents dans le programme quelque soit le chemin d'exécution (figure 1.9). Un contexte C_i qui suit un noeud N_j est fonction des contextes arrivant à N_j, ainsi que de l'opération effectuée dans N_j. Une fois toutes les contraintes établies, il suffit de les résoudre en utilisant la recherche du plus petit point fixe. Par exemple, dans le cadre d'un langage comme PASCAL, quand on indexe un tableau à l'aide d'une variable X, il faut s'assurer que toutes les valeurs que peut prendre cette variable sont dans les bornes du tableau. Nous prendrons comme exemple la variable X qui apparaît dans le schéma 1.9. Nous allons déterminer quelles sont les valeurs que peut prendre cette variable (pour vérifier l'indiçage d'un tableau par exemple). Pour ce faire, nous utiliserons le domaine intervalle ; son treillis peut être représenté par la figure 1.10. Il est clair que ce treillis est infini.

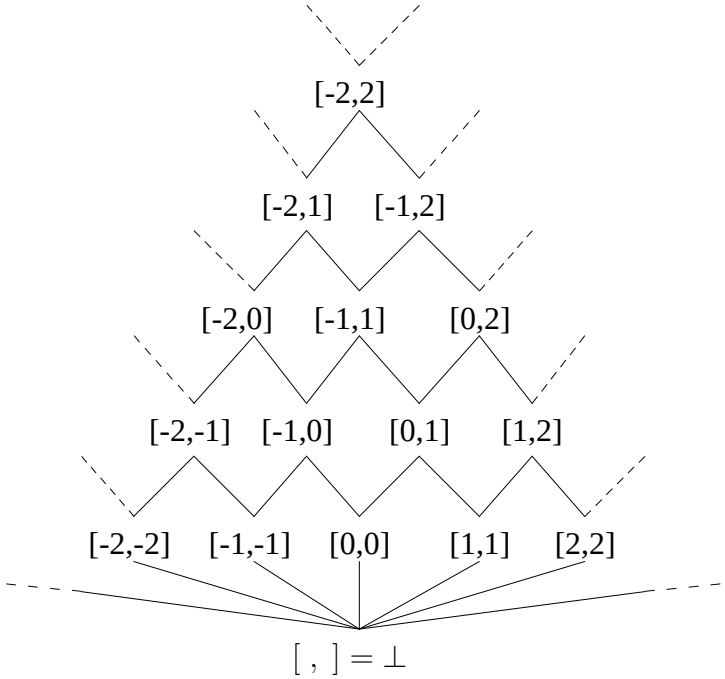

FIG. 1.10 – *Treillis des intervalles sur les entiers*

Soit la notation $[a, b]$ avec $a \leq b$ pour le prédicat $a \leq x \leq b$, le système résultant de l'exemple est :

$$\begin{array}{lcl} C_0 & = & [\ ,\ ] \\ C_1 & = & [1,1] \\ C_2 & = & C_1 \cup C_4 \\ C_3 & = & C_2 \cap [-\infty, 100] \\ C_4 & = & C_3 + [1,1] \\ C_5 & = & C_2 \cap [101, +\infty] \end{array}$$

avec la convention $[a, b]+[c, d] = [a+c, b+d]$. Dans un treillis infini, il existe une infinité de chaînes infinies. Nous savons qu'en cas de chaîne infinie l'itération classique peut ne pas terminer. Il est donc intéressant d'introduire l'opérateur d'élargissement $\bigtriangledown$ (voir définition section 1.3.1.3) sur les intervalles par :

$$\begin{array}{l} [\ ,\ ] \text{ est l'élément nul pour } \bigtriangledown \\ [i, j] \bigtriangledown [k, l] \quad = \quad [\underline{si}\ k < i\ \underline{alors}\ -\infty\ \underline{sinon}\ i\ \underline{fsi}, \\ \qquad\qquad\quad \underline{si}\ l > j\ \underline{alors}\ +\infty\ \underline{sinon}\ j\ \underline{fsi}] \end{array}$$

Il suffit de modifier le système en changeant l'équation de C_2 par $C_2 = C_2 \bigtriangledown (C_1 \cup C_4)$. C_2 est modifié car il dépend de l'expression $X := X + 1$ qui peut être infini. Cette modification permet de garantir la convergence vers un point fixe et donc de garantir la terminaison de l'algorithme. Nous appliquons le calcul du plus petit point fixe en commençant par initialiser les C_i avec $[\ ,\ ]$ puis en itérant. Nous obtenons le résultat suivant :

$$\begin{array}{lcl} C_0 & = & [\ ,\ ] \\ C_1 & = & [1,1] \\ C_2 & = & [1,+\infty] \\ C_3 & = & [1,100] \\ C_4 & = & [2,101] \\ C_5 & = & [101,+\infty] \end{array}$$

Dans le contexte C_3, la variable X ne sort jamais de l'intervalle $[1,100]$. Si dans ce contexte, la variable X sert à indicer un tableau $[a..b]$ avec $a \leq 1$ et $b \geq 100$ nous pouvons garantir qu'il n'y aura pas d'accès hors bornes. Dans le cas contraire, nous pouvons indiquer un risque de problème. Voici le tableau récapitulatif des itérations :

0	${}^*C_0$	$=$	$[\ ,\ ]$
	C_i	$=$	$[\ ,\ ]$ *pour* $i \in [1,5]$
1	${}^*C_1$	$=$	$[1,1]$
2	C_2	$=$	$C_2 \bigtriangledown (C_1 \cup C_4)$
		$=$	$[\ ,\ ] \bigtriangledown ([1,1] \cup [\ ,\ ])$
		$=$	$[\ ,\ ] \bigtriangledown [1,1]$
		$=$	$[1,1]$
3	C_3	$=$	$C_2 \cap [-\infty, 100]$
		$=$	$[1,1] \cap [-\infty, 100]$
		$=$	$[1,1]$
4	C_4	$=$	$C_3 + [1,1]$
		$=$	$[1,1] + [1,1]$
		$=$	$[2,2]$
5	C_2	$=$	$C_2 \bigtriangledown (C_1 \cup C_4)$
		$=$	$[1,1] \bigtriangledown ([1,1] \cup [2,2])$
		$=$	$[1,1] \bigtriangledown [1,2]$
	*	$=$	$[1,+\infty]$
6	C_3	$=$	$C_2 \cap [-\infty, 100]$
		$=$	$[1,+\infty] \cap [-\infty, 100]$
	*	$=$	$[1,100]$
7	C_4	$=$	$C_3 + [1,1]$
		$=$	$[1,100] + [1,1]$
	*	$=$	$[2,101]$
8	C_5	$=$	$C_2 \cap [101, +\infty]$
		$=$	$[1,+\infty] \cap [101, +\infty]$
	*	$=$	$[101,+\infty]$

La valeur finale de chaque contexte est notée par une étoile *.

1.4.2 Les langages orientés objets

Nous nous plaçons dans le cadre des langages orientés objets non typés avec des affectations et la liaison différée. L'utilisation de la liaison différée peut amener les programmes à être peu fiables (appel de méthodes non implémentées), illisibles (quelle méthode est effectivement appelée) et inefficaces (recherche dynamique de la méthode). En fait, étant donné un appel de méthode, nous aimerions savoir quelles sont toutes les classes qui implémentent cette méthode. L'inférence de type peut aider à résoudre ce problème

mais, jusque là, aucun algorithme d'inférence n'a été capable de contrôler complètement les programmes non typés les plus communs.

Dans ce cadre, Jens Palsberg et Michael I. Schwartzbach [Palsberg et Schwartzbach 91] proposent un algorithme qui garantit que tous les appels sont corrects (pas d'appel à des méthodes non implémentées ou incorrectes), annote le programme avec l'information de type, permet l'utilisation de méthodes polymorphes et peut être employé comme la base d'un compilateur optimisant. Selon cette proposition, les types forment des ensembles finis de classes et le sous-typage est représenté par l'inclusion d'ensemble. Étant donné un programme concret, l'algorithme construit un système de contraintes de type où les variables représentent les classes possibles d'une expression qui apparaît dans l'implémentation d'une méthode. L'algorithme est similaire aux travaux précédents sur l'inférence de type (se référer à [Milner 78, Borning et Ingalls 82, Cardelli 84] etc.), en utilisant des contraintes de type, mais il diffère en maniant la liaison différée par des contraintes conditionnelles. Toutes les équations, ainsi construites, sont de la forme :

$$C_1, C_2, ..., C_k \rightarrow Q$$

avec :

- un ensemble fini $\mathcal{A}$ de classes,
- chaque C_i de la forme $a \in X_i$ avec X_i une variable et $a \in \mathcal{A}$ une constante,
- une inégalité Q d'une de ces trois formes (avec $A \subseteq \mathcal{A}$) :

$$A \subseteq X_i$$
$$X_i \subseteq A$$
$$X_i \subseteq X_j$$

.

Le programme est typable si ces contraintes sont solvables. L'algorithme utilise une méthode itérative de calcul de point fixe pour trouver la plus petite solution (voir section 1.3.1), en temps exponentiel dans le pire des cas. En fait c'est la construction du problème qui peut se faire en temps exponentiel, l'algorithme en lui-même travaille en temps quadratique. Si ce point fixe n'est pas acceptable (c-à-d. que le programme n'est pas typable) alors l'algorithme retourne une erreur. Ici, l'absence de solution est gérée par ajout d'une solution, puis invalidation si cette solution est trouvée. Le résultat est un vecteur de n composantes (n est le nombre de variables), qui indique pour chaque variable quelles sont les valeurs qui satisfont le problème ou la valeur «erreur» pour indiquer qu'il y a un problème.

1.4.3 Les langages fonctionnels

En programmation fonctionnelle certaines analyses sont très utilisées comme l'inférence de type, l'analyse de nécessité [Mishra 88, Kuo et Mishra 89], l'analyse de gestion de mémoire [Aiken et al. 95], l'analyse de sécurité [Palsberg et Schwartzbach 95]... Il existe différentes manières d'analyser ces programmes ainsi que différentes façons de résoudre ces analyses.

Un des problèmes les plus étudiés est l'inférence de type [Palsberg et O'Keefe 95]. Étant donné un λ-terme E, le problème d'inférence peut être redéfini comme étant un problème de résolution d'un système de contraintes de type. L'ensemble des types peut être représenté par la grammaire suivante :

$$t ::= \ t_1 \rightarrow t_2 \mid Int \mid v \mid \mu v.t \mid \top \mid \bot$$

Avec une relation binaire de construction de type $\rightarrow$, un type constant Int, la possibilité de créer des types récursifs et deux constantes de types supplémentaires, $\top$ et $\bot$. Nous avons en plus une relation de sous-typage noté $\leq$. Nous utilisons comme exemple un langage dérivé du λ-calcul, généré par la grammaire suivante :

$$E ::= \ x \mid \lambda x.E \mid (E\ E) \mid 0 \mid succ\ E$$

Soit $\sigma = \{®, Int, \bot, \top\}$ l'alphabet où $\rightarrow$ est binaire et $Int, \bot, \top$ sont 0-aires. Soit T_σ l'ensemble des types défini par :

$$T_\sigma ::= Int \mid \bot \mid \top \mid T_\sigma \rightarrow T_\sigma$$

Nous supposons que toutes les variables liées de E sont distinctes (il suffit de faire une α-conversion). Soit X_E, l'ensemble des λ-variables apparaissant dans E et soit Y_E l'ensemble disjoint de X_E, constitué d'une variable $[\![F]\!]$ pour chaque occurrence d'un sous-terme F de E. Pour résoudre le problème du typage, nous générons le système suivant sur $X_E \cup Y_E$:

- pour toutes occurrences dans E d'un sous-terme de la forme 0, l'inégalité
$$Int \leq [\![0]\!]$$
- pour toutes occurrences dans E d'un sous-terme de la forme $succ\ F$, les deux inégalités
$$Int \leq [\![succ\ F]\!]$$
$$[\![F]\!] \leq Int$$
- pour toutes occurrences dans E d'un sous-terme de la forme $\lambda x.F$, l'inégalité
$$(x \rightarrow [\![F]\!])_{\lambda x.F} \leq [\![\lambda x.F]\!]$$
- pour toutes occurrences dans E d'un sous-terme de la forme GH, l'inégalité
$$[\![G]\!] \leq ([\![H]\!] \rightarrow [\![GH]\!])_{GH}$$
- pour toutes occurrences dans E d'une λ-variable x, l'inégalité
$$x \leq [\![x]\!]$$

Soit $T(E)$ le système de contraintes généré sur E. Pour tous les λ-termes E, soit $Tmap(E)$ l'ensemble des fonctions totales de $X_E \cup Y_E$ dans T_σ. La fonction $\psi \in Tmap(E)$ est solution de $T(E)$ si c'est une solution pour toutes les contraintes de $T(E)$. Spécialement pour $V, V', V'' \in X_E \cup Y_E$ et pour les occurrences des sous-termes $\lambda x.F$ et GH dans E, ce qui donne :

La contrainte	a une solution ψ si
$Int \leq V$	$Int \leq \psi(V)$
$V \leq Int$	$\psi(V) \leq Int$
$(V \rightarrow V')_{\lambda x.F} \leq V''$	$\psi(V) \rightarrow \psi(V') \leq \psi(V'')$
$V \leq (V' \rightarrow V'')_{GH}$	$\psi(V) \leq \psi(V') \rightarrow \psi(V'')$
$V \leq V'$	$\psi(V) \leq \psi(V'')$

Les solutions de $T(E)$ correspondent aux annotations de types possibles pour E.

1.4.4 Les langages logiques

En programmation logique, certaines analyses sont fréquemment utilisées comme l'analyse de mode [Warren 77], l'analyse de partage [Debray et Warren 88], etc.

Une autre analyse est l'analyse de déterminisme [Mellish 85], elle permet de déterminer dans un programme Prolog si les prédicats sont déterministes. Pour ce faire, la propriété de déterminisme pour le programme Prolog est modélisée par un système d'équations. Ce passage d'un programme à un système est très simple, nous allons le regarder sur deux exemples.

Notation : nous noterons <X> la propriété de déterminisme du prédicat X.

Exemple 1.19 : Considerons la version naïve de *reverse* (renverser une liste) :

```
nrev ([X|Y],Z) :- !, nrev (Y,Z1), append (Z1,[X],Z).
nrev ([],[]).

append ([X|Y],Z,[X|Y1]) :- !, append (Y,Z,Y1).
```

```
append ([],X,X).
```

`nrev` dépend de `nrev` et de `append` donc `nrev` est déterministe si `nrev` et `append` le sont. De même pour `append`, il est déterministe si `append` l'est. La propriété de déterminisme pour ce programme peut se modéliser par le système suivant :

<nrev> = <nrev> and <append>
<append> = <append>

- Nous allons voir sur un exemple plus complet comment se passe la résolution. C'est en fait une recherche de point fixe de manière itérative. Le treillis est le suivant :

FALSE
|
TRUE

L'initialisation des variables se fait avec la valeur TRUE.

Le résultat pour le premier exemple donne la valeur TRUE aux deux prédicats et donc tous les prédicats sont déterministes.

Exemple 1.20 :

```
human (X) :- mother (X,Mother), human (Mother).

animal (X) :- human (X).

mother (fred, jane).
mother (abel,eve).
```

Nous en déduisons le système d'équations :

<human> = <mother> and <human>
<animal> = <human>
<mother> = FALSE

Voici le tableau récapitulatif des itérations :

0	<human>	-	TRUE
	<animal>	-	TRUE
	<mother>	-	TRUE
1	<human>	-	TRUE
	<animal>	-	TRUE
	<mother>	-	FALSE
2	<human>	-	FALSE
	<animal>	-	TRUE
	<mother>	-	FALSE
3	<human>	-	FALSE
	<animal>	-	FALSE
	<mother>	-	FALSE

Donc, aucun des prédicats n'est déterministe.

- Il existe aussi l'analyse de partage. Cette analyse permet de propager des informations à travers les liaisons des variables. Christopher S. Mellish a travaillé sur cette analyse de partage [Mellish 85]. Le résultat de cette analyse peut être intéressant pour optimiser la gestion mémoire du code généré. Son travail comportait

quelques erreurs qui furent corrigées par Saumya K. Debray et Raghu Ramakrishnan [Debray et Ramakrishnan 94]. Ils se sont servi d'une transformation de programmes logiques appelée «magic» [Bancilhon et al. 86]. Cette transformation permet de passer du programme Prolog dans lequel le contrôle est invisible à un programme Prolog dans lequel le contrôle est visible. Il a été montré que l'analyse par chaînage arrière [2] du programme original est équivalente à l'analyse par chaînage avant [3] du programme résultant de la tranformation. Ces travaux, avec ceux de David S. Warren (OLDT [Warren 92], [Chen et al. 95]), Pascal Van Hentenryck, Agostino Cortesi et Baudouin Le Charlier [Van Hentenryck et al. 95], Michael Codish et Bart Demoen [Codish et Demoen 93] ont amené à faire de la compilation abstraite (λ-Prolog [Malésieux et al. 98]). D'autres auteurs comme Patrick et Radhia Cousot [Cousot et Cousot 92a] ont essayé de ramener l'utilisation de l'interprétation abstraite dans un cadre plus conventionnel. Le problème, c'est qu'il est très difficile d'avoir une sémantique concrète de type opérationnelle, qui nous montre certains traits de l'exécution du programme Prolog pour pouvoir l'abstraire. Dans cette optique, des sémantiques «instrumentables» ont été développées ([Le Charlier et Musumbu 92, Le Charlier et al. 94]).

Toutes ces analyses sont restées à l'état de prototype. Mais il existe un langage de programmation logique qui s'appelle Mercury (un quasi Prolog) développé à l'université de Melbourne par Zoltan Somogyi, Fergus Henderson et Thomas C. Conway ([Conway et al. 95, Conway et al. 96]) qui utilise de nombreuses analyses statiques d'une manière essentielle. Tout le processus de compilation est guidé par de telles analyses.

1.5 Conclusion

Bien que chaque langage nécessite des analyses particulières (l'analyse de nécessité pour les langages fonctionnels ou l'analyse de mode pour les langages logiques), certaines analyses sont plus générales et existent pour différents langages (par exemple l'inférence de types).

Il faut aussi constater que dans le cadre des analyses statiques, il existe une séparation entre la partie analyse propement dite qui produit un système et la partie résolution qui résoud ce système. Beaucoup d'analyses peuvent se résumer à une production d'un système plus ou moins complexe (il suffit de regarder les différents exemples exposés dans la section 1.4).

L'intérêt de cette remarque est qu'il devient envisageable d'avoir d'une part un analyseur générique pouvant produire un système (voir [Gouranton 97]) et d'autre part un moteur générique capable de résoudre une grande variété de systèmes. Cette approche est celle du projet LANDE à l'Irisa : être capable de fournir un outil générique qui remplacerait les différents outils utilisés dans le cadre d'analyses statiques de programmes.

2. Il suffit de partir de la tête de la clause. La tête est vérifiée si le corps de la clause est vérifié. Pour la clause $p : -q, r, s.$ et la question p, la réponse est p est vrai si q, r et s sont vrais et ainsi de suite. Le calcul ne se fait que sur ce qui est intéressant pour la question.

3. Le calcul est fait sur tout le programme avec une recherche de point fixe. On obtient la sémantique du programme.

Chapitre 2

Existence de solution?

Dans ce chapitre, nous allons présenter les systèmes que nous voulons résoudre et surtout nous allons nous intéresser à l'existence de solutions dans de tels systèmes et à la manière de les trouver si elles existent.

2.1 Définition du problème

Nous voulons trouver la plus petite solution d'un système. Cette recherche se fait à l'aide d'un algorithme itératif de recherche du plus petit point fixe. Le système est composé de variables et d'expressions constituées d'opérateurs monotones. Il peut exister plusieurs relations portant sur la même variable et ces relations peuvent être de type égalité ou de type inégalité (supérieur ou égal). Un système est représenté tableau 2.1. Les variables sont notées f_i et ce sont des variables d'ordre supérieur.

$$
P \ : \ \left\{ \begin{array}{lll}
 & = & e_1^1 \\
 & \vdots & \\
f_1(\overrightarrow{x_1}) & = & e_1^{t_1} \\
 & \geq & e_1^{t_1+1} \\
 & \vdots & \\
 & \geq & e_1^{s_1} \\
 & \vdots & \\
 & = & e_n^1 \\
 & \vdots & \\
f_n(\overrightarrow{x_n}) & = & e_n^{t_n} \\
 & \geq & e_n^{t_n+1} \\
 & \vdots & \\
 & \geq & e_n^{s_n}
\end{array} \right.
$$

TAB. 2.1 – *Le système original*

Dans un premier temps nous allons regarder ce qui se passe pour un système qui est constitué exclusivement d'inégalités dans la section 2.2. Ensuite, nous regarderons comment intégrer les équations au système d'inéquations dans la section 2.3.

2.2 Système avec des inégalités

Dans un premier temps, nous allons nous occuper des systèmes ne comportant que des relations de type supérieur ou égal. Il est possible d'avoir plusieurs relations portant sur la même variable et ces variables sont d'ordre supérieur.

Le système :

$$A = \begin{cases} & \geq & e_1^1 \\ f_1(\overrightarrow{x_1}) & \vdots & \\ & \geq & e_1^{s_1} \\ & \vdots & \\ & \geq & e_n^1 \\ f_n(\overrightarrow{x_n}) & \vdots & \\ & \geq & e_n^{s_n} \end{cases}$$

Existe-t-il toujours une solution pour un tel système? Une solution de ce système est une valeur pour chaque f_i telle que le système soit vérifié (si nous donnons à chaque $f_i(\overrightarrow{x_i})$ la valeur $\top$ alors le système A est vérifié, car $\forall e, \top \geq e$). Nous allons réécrire le système de la façon suivante.

$$S = \begin{cases} f_i(\overrightarrow{x_i}) & \geq & e_i^1 \\ & \vdots & \\ f_i(\overrightarrow{x_i}) & \geq & e_i^{s_i} \end{cases}$$

$$S' = f_i(\overrightarrow{x_i}) \geq e_i^1 \sqcup ... \sqcup e_i^{s_i}$$

Théorème 2.1 *S et S' sont équivalents, une solution pour S' est une solution pour S.*

PREUVE [équivalence des deux systèmes] : Si ϕ est une solution pour S, $\phi \geq e_i^1, ..., \phi \geq e_i^{s_i}$. Donc $\phi \geq e_i^1 \sqcup ... \sqcup e_i^{s_i}$ et donc ϕ est une solution pour S'. De même si ϕ est une solution pour S', $\phi \geq e_i^1 \sqcup ... \sqcup e_i^{s_i}$. Donc $\phi \geq e_i^1, ..., \phi \geq e_i^{s_i}$ et donc ϕ est une solution pour S. □

Donc A peut se réécrire en A' tel que

$$A' = \begin{cases} f_1(\overrightarrow{x_1}) & \geq & (e_1^1 \sqcup ... \sqcup e_1^{s_1}) \\ & \vdots & \\ f_n(\overrightarrow{x_n}) & \geq & (e_n^1 \sqcup ... \sqcup e_n^{s_n}) \end{cases}$$

Prouver qu'il existe au moins une solution pour A revient à montrer qu'il existe au moins une solution pour A'. Nous abrègerons les expressions $e_i^1 \sqcup ... \sqcup e_i^{s_i}$ par e_i, ce qui nous donne pour le système A' :

$$A' = \begin{cases} f_1(\overrightarrow{x_1}) & \geq & e_1 \\ & \vdots & \\ f_n(\overrightarrow{x_n}) & \geq & e_n \end{cases}$$

Théorème 2.2 *Les solutions du système A' peuvent être caractérisées par les points fixes de la fonction*

$$g : (D_1 \to E_1 \times ... \times D_n \to E_n) \to (D_1 \to E_1 \times ... \times D_n \to E_n)$$

induite par les e_i et définie par

$$g\phi = (d_1 \mapsto e_1\phi[\overrightarrow{x_1} \mapsto d_1], ..., d_n \mapsto e_n\phi[\overrightarrow{x_n} \mapsto d_n]).$$

Le plus petit point fixe de g est aussi la plus petite solution pour le système A'.

PREUVE [équivalence des deux solutions] : g est une fonction monotone sur des treillis finis donc, d'après Tarski, cette fonction admet un plus petit point fixe qui est caractérisé par $\sqcap\{f \mid f \geq g(f)\}$, ce qui est l'ensemble des solutions du système A'. Soit ϕ un point fixe de g, alors ϕ vérifie la relation $\phi \geq g(\phi)$ et donc ϕ est un solution du système A'. D'après la définition du glb, si $t = glb(T)$ alors t est plus petit ou égal à tous les éléments de T. Donc le plus petit point fixe de g est bien la plus petite solution du système A'. □

Maintenant que nous savons que le plus petit point fixe de g est la plus petite solution du système, il reste à le calculer.

Théorème 2.3 *Le plus petit point fixe de g peut être calculé avec l'algorithme de Work-Set (voir page 40).*

PREUVE [calculabilité] : Comme g est une fonction monotone sur des treillis finis, elle est continue et donc il est possible de caractériser un point fixe de g par $\bigsqcup_{i=0}^{\infty}\{g^i(\bot)\}$ et il est donc possible d'utiliser l'algorithme du Work-Set pour le calculer. Nous avons $g^0 = \bot \sqsubseteq g^1 \sqsubseteq g^2 \sqsubseteq ...$, mais l'algorithme de Work-Set ne calcule pas cette suite mais une approximation de cette suite car il ne calcule que ce qui a changé dans g. Nous avons donc la suite $\phi^0 = \bot \sqsubseteq \phi^1 \sqsubseteq \phi^2 \sqsubseteq ...$, avec quelque soit i, $\phi^i \sqsubseteq g^i$ [1]. Soit $\alpha = \sqcap\{X \geq g(X)\}$ le plus petit point fixe de g. L'algorithme s'arrête pour une solution ϕ^k. On a donc $\phi^k \geq g(\phi^k)$. Comme α est la plus petite solution de $X \geq g(X)$, on a $\phi^k \sqsupseteq \alpha$ (1). Nous avons $\forall i, \phi^i \sqsubseteq g^i$, en particulier pour $k, \phi^k \sqsubseteq g^k$ et $g^k \sqsubseteq \bigsqcup_0^{\infty}\{g^i\}$. Donc nous avons $\phi^k \sqsubseteq \bigsqcup_0^{\infty}\{g^i\}$. Comme $\alpha = \bigsqcup_0^{\infty}\{g^i\}$ nous pouvons déduire que $\phi^k \sqsubseteq \alpha$ (2). Par (1) et (2) nous pouvons déduire que $\phi^k = \bigsqcup\{g^i\}$ et donc que ϕ^k est le plus petit point fixe de g. Par conséquent, l'algorithme du Work-Set calcule bien la plus petite solution du système A'. □

2.3 Prise en compte des égalités

Nous allons maintenant intégrer les équations aux systèmes d'inéquations. Plusieurs problèmes se posent lorsque nous voulons traiter un système mixte. Nous regarderons ce qui se passe en ce qui concerne l'existence des solutions et si nous pouvons les trouver avec un algorithme de recherche du plus petit point fixe.

2.3.1 Existence de solution

2.3.1.1 Cas où il y a plusieurs égalités portant sur la même variable

Le sous-système peut s'écrire :

$$\begin{cases} f_i(\overrightarrow{x_i}) &= e_i^1 \\ f_i(\overrightarrow{x_i}) &= e_i^2 \end{cases}$$

Si nous nous plaçons dans un treillis à deux éléments, il est facile de voir qu'il n'existe pas toujours de solution à ce problème. Par exemple pour le problème suivant :

Le système :

$$X = \top$$
$$X = \bot$$

1. Soit I l'ensemble des indices des variables à réévaluer. Démonstration par récurrence, $(\phi^0 = \bot) \sqsubseteq (g^0 = \bot)$. Supposons $\phi^{i-1} \sqsubseteq g^{i-1}$. $\forall j \in I$, nous avons $\phi_j^i = e_j\phi^{i-1}$. Comme $\phi^{i-1} \sqsubseteq g^{i-1}$, nous avons $\phi_j^i \sqsubseteq e_j g^{i-1}$. Mais $e_j g^{i-1} = g_j^i$, donc $\forall j \in I$, nous avons $\phi_j^i \sqsubseteq g_j^i$. $\forall k \notin I$ nous avons $\phi_k^i = \phi_k^{i-1}$. Comme $\phi^{i-1} \sqsubseteq g^{i-1}$, nous avons $\phi_k^i \sqsubseteq g_k^{i-1}$. Mais $g^{i-1} \sqsubseteq g^i$, donc nous avons $\forall k \notin I, \phi_k^i \sqsubseteq g_k^i$. Donc $\forall j$, nous avons $\phi_j^i \sqsubseteq g_j^i$ et donc $\phi^i \sqsubseteq g^i$.

Avec le treillis :

$$\begin{array}{c} \top \\ | \\ \bot \end{array}$$

Le système n'a pas de solution.

2.3.1.2 Cas où il y a une égalité et plusieurs inégalités portant sur la même variable

Le sous-système peut s'écrire :

$$\left\{ \begin{array}{lcl} f_i(\overrightarrow{x_i}) & = & e_i^1 \\ f_i(\overrightarrow{x_i}) & \geq & e_i^2 \end{array} \right.$$

Si nous nous plaçons dans un treillis à deux éléments, il est facile de voir qu'il n'existe pas toujours de solution à ce problème. Par exemple pour le problème suivant :

Le système :

$$\begin{array}{l} X \geq \top \\ X = \bot \end{array}$$

Avec le treillis :

$$\begin{array}{c} \top \\ | \\ \bot \end{array}$$

Le système n'a pas de solution.

Il est donc impossible de garantir qu'il existe une solution pour des systèmes dans lesquels il existe plusieurs relations sur la même variable, sauf si ces relations sont des inégalités. Nous allons regarder s'il est possible de forcer l'existence de solution en transformant le système de départ.

2.3.2 Transformation du système de départ

Nous sommes capable de résoudre un système composé exclusivement d'inéquations de la forme $variable \geq expression$. Cette résolution se fait avec un algorithme itératif de recherche du plus petit point fixe (celui présenté par Kildall par exemple [Kildall 73]). Donc, pour résoudre le système de départ (tableau 2.1) qui est composé d'équations et d'inéquations, nous allons le transformer en deux systèmes, un système à résoudre (tableau 2.2) et un système de contraintes (tableau 2.3). La règle de transformation est la suivante : toute relation d'égalité est transformée en deux relations d'inégalité. Nous allons voir si l'algorithme peut toujours être utilisé.

$$S \ : \ \left\{ \begin{array}{lcl} f_1(\overrightarrow{x_1}) & \geq & \bigsqcup_{i=1}^{s_1} e_1^i \\ & \vdots & \\ f_n(\overrightarrow{x_n}) & \geq & \bigsqcup_{i=1}^{s_n} e_n^i \end{array} \right.$$

TAB. 2.2 – *Le système à résoudre*

Définition 2.4 *Une solution $\widehat{\phi}$ de P est une solution pour S vérifiant C.*

$$C \ : \ \left\{ \begin{array}{lcl} f_1(\overrightarrow{x_1}) & \leq & \sqcap_{i=1}^{t_1} e_1^i \\ & \vdots & \\ f_n(\overrightarrow{x_n}) & \leq & \sqcap_{i=1}^{t_n} e_n^i \end{array} \right.$$

TAB. 2.3 – *Le système de contraintes*

2.3.2.1 Résolution du système P

Nous aimerions que la plus petite solution de P soit aussi la plus petite solution de S. Hélas, ce n'est pas le cas.

Théorème 2.5 *Soit $\widehat{\widehat{\phi}}$ une solution de S vérifiant C, il peut exister $\widehat{\phi}$ une solution de S ne vérifiant pas C plus petite que $\widehat{\widehat{\phi}}$.*

PREUVE [théorème 2.5] :

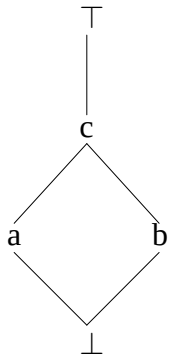

FIG. 2.1 – *le treillis de la preuve du théorème 2.5*

Le système (basé sur le treillis figure 2.1):

$$P : \left\{ \begin{array}{lcl} X & = & if\ X \sqsubseteq c\ then\ a\ else\ \top \\ X & = & if\ X \sqsubseteq c\ then\ b\ else\ \top \end{array} \right.$$

Ce qui donne :

$$\begin{array}{lcll} S & : & X & \geq & if\ X \sqsubseteq c\ then\ a \sqcup b\ (= c)\ else\ \top \sqcup \top\ (= \top) \\ C & : & X & \leq & if\ X \sqsubseteq c\ then\ a \sqcap b\ (= \bot)\ else\ \top \sqcap \top\ (= \top) \end{array}$$

La plus petite solution de S est $X = c$ et elle n'est pas valide pour C. Mais l'autre point fixe de S qui est $X = \top$ est solution du système de départ. □

Il n'est donc pas possible de résoudre de tels problèmes avec la méthode de recherche du plus petit point fixe. Trois choix sont alors possibles :

– Restreindre la syntaxe du système de départ : s'il existe une contrainte d'égalité portant sur une variable alors c'est la seule contrainte. Dans ce cas il n'existe plus de système de contraintes (nous ne sommes pas obligé de faire la transformation des égalités).

– Donner des limitations sur les expressions construites pour les contraintes, afin de garantir que la plus petite solution de P est aussi la plus petite solution de S.
– Changer la méthode de résolution.

Nous ne voulons pas changer de méthode de résolution car nous n'en connaissons pas d'autre. La seule technique qui serait susceptible de nous aider est celle utilisant le *Widening* et le *Narrowing* [Cousot et Cousot 77]. Mais cette technique ne garantit pas que le point fixe trouvé est le plus petit point fixe. De plus, si le point fixe trouvé ne vérifie pas le système de contraintes, nous ne pourrions pas dire s'il existe une autre solution qui vérifierait C ni où elle se trouve par rapport à la solution que nous venons de trouver. Nous allons donc développer les deux autres possibilités. Dans un premier temps nous verrons comment limiter le système de départ dans la section 2.3.2.2 et dans un deuxième temps nous regarderons les propriétés intéressantes pour le système de contraintes dans la section 2.3.2.3.

2.3.2.2 Restriction sur P

Nous définissons les fonctions ρ (sur le système à résoudre S) et ψ (sur le système de contraintes C) de la manière suivante :

$$\begin{array}{lcl} \rho & : & (D_1 \to E_1 \times ... \times D_n \to E_n) \to (D_1 \to E_1 \times ... \times D_n \to E_n) \\ \rho\phi & = & (d_1 \mapsto \bigsqcup_{i=1}^{s_1} e_1^i \phi[\overrightarrow{x_1} \mapsto d_1], ..., d_n \mapsto \bigsqcup_{i=1}^{s_n} e_n^i \phi[\overrightarrow{x_n} \mapsto d_n]) \end{array}$$

$$\begin{array}{lcl} \psi & : & (D_1 \to E_1 \times ... \times D_n \to E_n) \to (D_1 \to E_1 \times ... \times D_n \to E_n) \\ \psi\phi & = & (d_1 \mapsto \sqcap_{i=1}^{t_1} e_1^i \phi[\overrightarrow{x_1} \mapsto d_1], ..., d_n \mapsto \sqcap_{i=1}^{t_n} e_n^i \phi[\overrightarrow{x_n} \mapsto d_n]) \end{array}$$

Les fonctions ρ et ψ résument les système S et C.

Remarque 2.6 *ρ et ψ sont deux fonctions monotones par construction.*

Nous limitons le système de la manière suivante, soit $t_i = 1$ et $s_i = t_i$ soit $t_i = 0$ et $s_i \geq 1$. Dans ce cas la fonction ψ n'existe plus, nous ne travaillons qu'avec la fonction ρ.

Théorème 2.7 *Les solutions du système P peuvent être caractérisée par les points fixes de la fonction ρ. Le plus petit point fixe de ρ est aussi la plus petite solution pour le système P.*

PREUVE [équivalence des deux solutions] : ρ est une fonction monotone sur des treillis finis donc, d'après Tarski, cette fonction admet un plus petit point fixe qui est caractérisé par $\sqcap\{f \mid f = \rho(f)\}$, ce qui est l'ensemble des solutions du système P. Soit ϕ un point fixe de ρ, alors ϕ vérifie la relation $\phi = \rho(\phi)$ et donc ϕ est un solution du système P. D'après la définition du glb, si $t = glb(T)$ alors t est plus petit ou égal à tous les éléments de T. Donc le plus petit point fixe de ρ est bien la plus petite solution du système P. □

Théorème 2.8 *Le plus petit point fixe de ρ peut être calculé avec l'algorithme itératif de recherche du plus petit point fixe.*

PREUVE [validité de l'algorithme] : Comme ρ est une fonction monotone sur des treillis finis, elle est continue et donc il est possible de caractériser un point fixe de ρ par $\bigsqcup_{i=0}^{\infty}\{\rho^i(\bot)\}$ et il est donc possible d'utiliser l'algorithme itératif pour le calculer. Nous avons $\rho^0 = \bot \sqsubseteq \rho^1 \sqsubseteq \rho^2 \sqsubseteq ...$, mais l'algorithme itératif ne calcule pas cette suite mais une approximation de cette suite car il ne calcule que ce qui a changé dans ρ. Nous avons donc la suite $\phi^0 = \bot \sqsubseteq \phi^1 \sqsubseteq \phi^2 \sqsubseteq ...$, avec quelque soit i, $\phi^i \sqsubseteq \rho^i$ [2]. Soit $\alpha = \sqcap\{X =$

2. Soit I l'ensemble des indices des variables à réévaluer. Démonstration par récurrence, $(\phi^0 = \bot) \sqsubseteq (\rho^0 = \bot)$. Supposons $\phi^{i-1} \sqsubseteq \rho^{i-1}$. $\forall j \in I$, nous avons $\phi_j^i = e_j\phi^{i-1}$. Comme $\phi^{i-1} \sqsubseteq \rho^{i-1}$, nous avons $\phi_j^i \sqsubseteq e_j\rho^{i-1}$. Mais $e_j\rho^{i-1} = \rho_j^i$, donc $\forall j \in I$, nous avons $\phi_j^i \sqsubseteq \rho_j^i$. $\forall k \notin I$ nous avons $\phi_k^i = \phi_k^{i-1}$. Comme $\phi^{i-1} \sqsubseteq \rho^{i-1}$, nous avons $\phi_k^i \sqsubseteq \rho_k^{i-1}$. Mais $\rho^{i-1} \sqsubseteq \rho^i$, donc nous avons $\forall k \notin I, \phi_k^i \sqsubseteq \rho_k^i$. Donc $\forall j$, nous avons $\phi_j^i \sqsubseteq \rho_j^i$ et donc $\phi^i \sqsubseteq \rho^i$.

$\rho(X)\}$ le plus petit point fixe de ρ. L'algorithme s'arrête pour une solution ϕ^k. On a donc $\phi^k = \rho(\phi^k)$. Comme α est la plus petite solution de $X = \rho(X)$, on a $\phi^k \sqsupseteq \alpha$ (1). Nous avons $\forall i, \phi^i \sqsubseteq \rho^i$, en particulier pour $k, \phi^k \sqsubseteq \rho^k$ et $\rho^k \sqsubseteq \bigsqcup_0^\infty\{\rho^i\}$. Donc nous avons $\phi^k \sqsubseteq \bigsqcup_0^\infty\{\rho^i\}$. Comme $\alpha = \bigsqcup_0^\infty\{\rho^i\}$ nous pouvons déduire que $\phi^k \sqsubseteq \alpha$ (2). Par (1) et (2) nous pouvons déduire que $\phi^k = \bigsqcup\{\rho^i\}$ et donc que ϕ^k est le plus petit point fixe de ρ. Par conséquent, l'algorithme itératif calcule bien la plus petite solution du système P. □

2.3.2.3 Limitation de ψ

Nous avons montré qu'il était possible de restreindre le système et de le résoudre. Nous allons voir comment restreindre les opérateurs pour pouvoir résoudre un système plus complexe (un système tel que $t_i \geq 0$, $s_i \geq 0$ et $t_i + s_i \geq 1$).

Pour ce faire, nous introduisons une notion de distance. La distance $\mathcal{D}(X, Y)$ est la longueur du plus grand chemin orienté allant de X à Y. Nous travaillons avec un ordre partiel, nous obtenons donc une distance partiellement définie. Tous les points ne sont pas forcement en relation ; si deux points ne sont pas comparables alors il n'existe pas de distance entre ces deux points. Pour obtenir une distance totalement définie, il suffit de définir la distance entre deux points non comparables comme étant l'infini.

Définition 2.9 (Fonction contractante) *Une fonction f est dite contractante si et seulement si quelque soit X et Y comparables, $\mathcal{D}(X, Y) \geq \mathcal{D}(f(X), f(Y))$.*

La fonction ψ est monotone par construction et nous faisons l'hypothèse qu'elle est aussi contractante.

Théorème 2.10 *Si la fonction ψ est contractante alors la plus petite solution de P est la plus petite solution de S.*

PREUVE [théorème 2.10] : Soit $\widehat{\phi}$ la plus petite solution de S, $\widehat{\phi}$ ne vérifie pas C ($\widehat{\phi} \geq \rho\widehat{\phi}$ et $\neg(\widehat{\phi} \leq \psi\widehat{\phi})$). Soit $\widehat{\widehat{\phi}}$ une solution de S qui vérifie C, plus grande que $\widehat{\phi}$ ($\widehat{\widehat{\phi}} \geq \rho\widehat{\phi}$, $\widehat{\widehat{\phi}} \leq \psi\widehat{\widehat{\phi}}$ et $\widehat{\widehat{\phi}} > \widehat{\phi}$). Nous allons montrer que c'est impossible.

- $\widehat{\phi} \geq \rho\widehat{\phi}$, car $\widehat{\phi}$ est solution de S et $\rho\widehat{\phi} \geq \psi\widehat{\phi}$, car $\rho \geq \psi$ par définition. Donc $\widehat{\phi}$ et $\psi\widehat{\phi}$ sont comparables ($\widehat{\phi} \geq \psi\widehat{\phi}$), donc $\neg(\widehat{\phi} \leq \psi\widehat{\phi})$ implique $\widehat{\phi} > \psi\widehat{\phi}$. Donc (**)$\mathcal{D}(\widehat{\phi}, \psi\widehat{\phi}) = v$ existe et $v > 0$.
- $\widehat{\phi} < \widehat{\widehat{\phi}}$ donc (**)$\mathcal{D}(\widehat{\phi}, \widehat{\widehat{\phi}}) = d$ et $d > 0$.
- $\psi\widehat{\phi} \leq \psi\widehat{\widehat{\phi}}$ par monotonie de ψ donc (*)$\mathcal{D}(\psi\widehat{\phi}, \psi\widehat{\widehat{\phi}}) = d^-$ et $d^- \leq d$ par limitation de ψ.
- $\widehat{\widehat{\phi}} \leq \psi\widehat{\widehat{\phi}}$ car $\widehat{\widehat{\phi}}$ vérifie C, (**)$\mathcal{D}(\widehat{\widehat{\phi}}, \psi\widehat{\widehat{\phi}}) = v'$.
- Quels sont les chemins allant de $\psi\widehat{\phi}$ à $\psi\widehat{\widehat{\phi}}$.
  - par (*), $\mathcal{D}(\psi\widehat{\phi}, \psi\widehat{\widehat{\phi}}) = d^-$.
  - par (**), $\mathcal{D}(\psi\widehat{\phi}, \psi\widehat{\widehat{\phi}}) = v + d + v'$.

  d^- est la longueur du plus grand chemin, par définition, mais $d^- \leq d$ donc nous avons $d \geq d^- \geq v + d + v' > 0$. Donc $d^- = d$, $v' = 0$ et $v = 0$. $v = 0$ implique que $\widehat{\widehat{\phi}} = \psi\widehat{\phi}$ ce qui contredit $\widehat{\widehat{\phi}} > \psi\widehat{\phi}$.

□

Remarque 2.11 *Au cours de la preuve, nous avons montré des égalités entre certaines valeurs. Cela se résume sur la figure 2.3.*

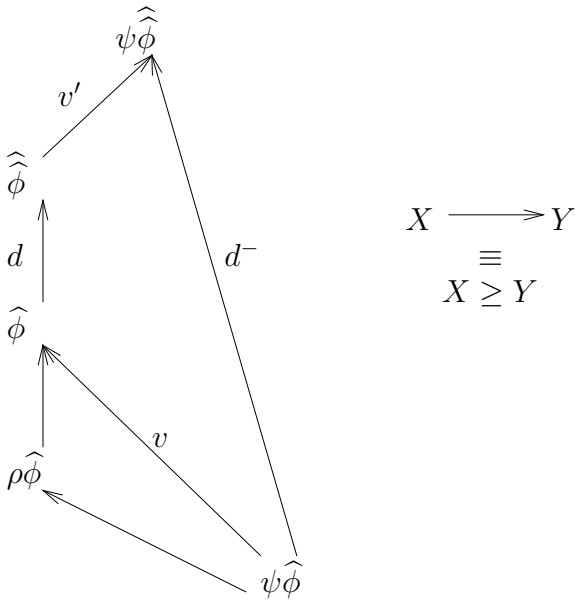

FIG. 2.2 – *Les relation entre les variables*

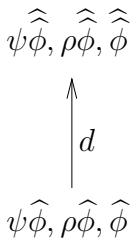

FIG. 2.3 – *Les relation entre les variables*

En conclusion de cette démonstration, nous pouvons dire que la propriété de contractance est une condition suffisante. Mais cette condition n'est pas nécessaire, en effet il est possible de trouver une fonction ψ telle qu'elle ne soit pas contractante et qu'il soit pourtant possible de trouver la plus petite solution.

Exemple 2.1 [Accessibilité de la solution sans la contractance] **:**

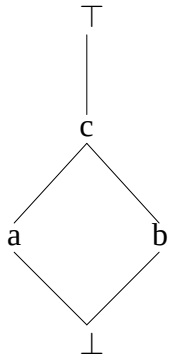

FIG. 2.4 – *le treillis de l'exemple 2.3.1*

Le système (basé sur le treillis figure 2.4):

$$P : \begin{cases} X & = & if\ X \sqsubset c\ then\ a\ else\ \top \\ X & = & if\ X \sqsubset c\ then\ b\ else\ \top \end{cases}$$

Ce qui donne :

$$\begin{array}{rcrcl} S & : & X & \geq & if\ X \sqsubset c\ then\ a \sqcup b\ (= c)\ else\ \top \sqcup \top\ (= \top) \\ C & : & X & \leq & if\ X \sqsubset c\ then\ a \sqcap b\ (= \bot)\ else\ \top \sqcap \top\ (= \top) \end{array}$$

La plus petite solution de S est $X = \top$ et elle est valide pour C, mais la fonction ψ, construite sur C n'est pas contractante ($\mathcal{D}(b, c) = 1$ et $\mathcal{D}(\psi(b), \psi(c)) = \mathcal{D}(\bot, \top) = 3$). La condition de contractance est bien une condition suffisante mais pas nécessaire. •

De même, dans le cas où le système de départ n'a pas de solution, nous pouvons conclure malgré l'absence de la propriété de contractance.

Exemple 2.2 [Conclure sans la contractance] **:**

Le système (basé sur le treillis figure 2.5):

$$P : \begin{cases} X & = & if\ X \sqsubseteq c\ then\ a\ else\ d \\ X & = & if\ X \sqsubseteq c\ then\ b\ else\ e \end{cases}$$

Ce qui donne :

$$\begin{array}{rcrcl} S & : & X & \geq & if\ X \sqsubseteq c\ then\ a \sqcup b\ (= c)\ else\ d \sqcup e\ (= \top) \\ C & : & X & \leq & if\ X \sqsubseteq c\ then\ a \sqcap b\ (= \bot)\ else\ d \sqcap e\ (= c) \end{array}$$

La plus petite solution de S est $X = c$ et elle n'est pas valide pour C. Nous concluons qu'il n'y a pas de solution pour P et c'est le cas. Pourtant la fonction ψ, construite sur C n'est pas contractante ($\mathcal{D}(c, d) = 1$ et $\mathcal{D}(\psi(c), \psi(d)) = \mathcal{D}(\bot, c) = 2$). La condition de contractance est bien une condition suffisante mais pas nécessaire. •

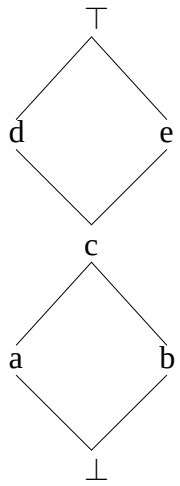

FIG. 2.5 – *le treillis de l'exemple 2.3.2*

2.3.2.4 Construction d'opérations contractantes

Nous venons de montrer qu'avec la propriété de contractance, nous étions capable de prouver que nous pouvions trouver la plus petite solution d'un système avec l'algorithme itératif de recherche du plus petit point fixe. Il reste à savoir si nous sommes capable de construire des opérateurs contractants. En particulier, pour les deux opérateurs *lub* et *glb*.

Théorème 2.12 *Soient f et g deux fonctions contractantes, alors $f \sqcup g$ n'est pas forcément une fonction contractante.*

PREUVE :

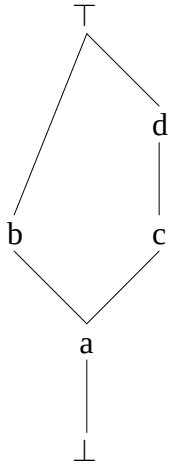

FIG. 2.6 – *Contre-exemple pour le lub*

Soit $f(X) = X$ et $g(X) = c$, deux fonctions sur le treillis figure 2.6, alors $h(X) = f(X) \sqcup g(X)$ n'est pas contractante. f, g et h sont des fonctions monotones. f et g sont des fonctions contractantes.

$\mathcal{D}(a,b) = 1$, $h(a) = c$ et $h(b) = \top$ d'où $\mathcal{D}(h(a), h(b)) = 2$ donc h n'est pas contractante. □

Théorème 2.13 *Soient f et g deux fonctions contractantes, alors $f \sqcap g$ n'est pas forcément une fonction contractante.*

PREUVE :

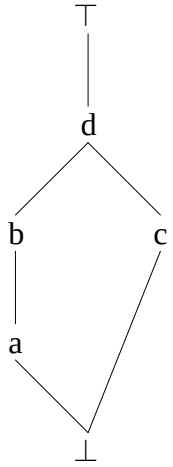

FIG. 2.7 – *Contre-exemple pour le glb*

Soit $f(X) = X$ et $g(X) = b$, deux fonctions sur le treillis figure 2.7, alors $h(X) = f(X) \sqcap g(X)$ n'est pas contractante. f, g et h sont des fonctions monotones. f et g sont des fonctions contractantes. $\mathcal{D}(d,c) = 1$, $h(d) = b$ et $h(c) = \bot$ d'où $\mathcal{D}(h(d), h(c)) = 2$ donc h n'est pas contractante. □

Nous avons cherché d'autres propriétés permettant d'obtenir le même résultat que la contractance, sans succès. Les propriétés comme celle que tout chemin allant de X à Y sont de même longueur ou bien que les treillis soient des sous-treillis non booléens d'un treillis booléen n'ont rien donné. Par contre, celle utilisant les treillis booléens, même si elle n'a pas permis de faire la preuve, n'a pas encore été infirmée par un contre-exemple.

2.3.2.5 Conclusion

Il n'y a pas de raison particulière pour que le *lub* et le *glb* soient contractants. Les restrictions à poser pour qu'ils le deviennent seraient trop contraignantes et cela n'aurait plus aucun intérêt. De plus la propriété de contractance est une condition suffisante mais pas nécessaire. Il faut donc revenir en arrière et revoir dans un premier temps la démonstration pour essayer de trouver un autre argument que la contractance et peut être revenir jusqu'à la transformation qui permettait de passer d'un système mixte à un système ne contenant que des inéquations.

Dans le cadre de la programmation logique, il existe un problème un peu similaire, c'est l'utilisation de la négation par l'échec [Apt et Bol 94]. Pour résoudre le problème de la négation, les programmes peuvent être stratifiés. Étant donné un programme stratifié en n couches (la première étant la couche 0), la négation peut être utilisée à la couche i si et seulement si elle n'utilise que des éléments des couches précédentes $(0, .., i-1)$. Dans notre cas, il faudrait que les expressions utilisées dans les égalités soient des strates

inférieures.

Dans un cadre général, il est difficile de garantir l'existence et l'accessibilité des solutions en présence d'équations. L'inexistence de solutions nous empêche d'utiliser un algorithme à la Kildall car il ne se terminera jamais. L'inaccessibilité empêche de conclure quand nous trouvons une solution invalide. Il reste donc un grand travail à faire en ce qui concerne ces deux problèmes.

2.4 Retour à un système d'égalité

Pour des raisons de commodité, nous allons regarder s'il est possible de retrouver un système ne contenant que des égalités à partir d'un système ne contenant que des inégalités.

Théorème 2.14 *Il est possible de retrouver un système ne contenant que des égalités en partant du système sur les inégalités.*

Pour cela il suffit d'utiliser une règle de transformation.

$$S = \left\{ \begin{array}{lcl} & \geq & e_i^1 \\ f_i(\overrightarrow{x_i}) & \vdots & \\ & \geq & e_i^{s_i} \end{array} \right. \quad \leftrightarrow \quad S' = \left\{ \begin{array}{lcl} f_i^1(\overrightarrow{x_i}) & = & e_i^1 \\ \vdots & & \\ f_i^{s_i}(\overrightarrow{x_i}) & = & e_i^{s_i} \\ f_i(\overrightarrow{x_i}) & = & f_i^1(\overrightarrow{x_i}) \sqcup ... \sqcup f_i^{s_i}(\overrightarrow{x_i}) \end{array} \right.$$

PREUVE [équivalence des systèmes S et S'] : Par définition du lub, nous avons que $f_i(\overrightarrow{x_i}) = f_i^1(\overrightarrow{x_i}) \sqcup ... \sqcup f_i^{s_i}(\overrightarrow{x_i})$ implique $f_i(\overrightarrow{x_i}) \geq f_i^1(\overrightarrow{x_i}), ..., f_i(\overrightarrow{x_i}) \geq f_i^{s_i}(\overrightarrow{x_i})$. Étant donnée la définition des $f_i^j(\overrightarrow{x_i})$, nous avons $f_i(\overrightarrow{x_i}) \geq e_i^1, ..., f_i(\overrightarrow{x_i}) \geq e_i^{s_i}$. Donc nous avons $S' \leftarrow S$. Nous avons $f_i(\overrightarrow{x_i}) \geq e_i^1, ..., f_i(\overrightarrow{x_i}) \geq e_i^{s_i}$ si et seulement si $f_i(\overrightarrow{x_i}) \geq e_i^1 \sqcup ... \sqcup e_i^{s_i}$. La plus petite solution est pour $f_i(\overrightarrow{x_i}) = e_i^1 \sqcup ... \sqcup e_i^{s_i}$ et elle existe toujours par définition du lub. Nous avons donc $f_i(\overrightarrow{x_i}) = f_i^1(\overrightarrow{x_i}) \sqcup ... \sqcup f_i^{s_i}(\overrightarrow{x_i})$ après renommage des e_i^j. Donc nous avons $S' \rightarrow S$. Les deux systèmes sont donc bien équivalents. □

Ce qui nous donne sur le système de départ :

$$A = \left\{ \begin{array}{lcl} & \geq & e_1^1 \\ f_1(\overrightarrow{x_1}) & \vdots & \\ & \geq & e_1^{s_1} \\ & \vdots & \\ & \geq & e_n^1 \\ f_n(\overrightarrow{x_n}) & \vdots & \\ & \geq & e_n^{s_n} \end{array} \right. \quad \leftrightarrow \quad A' = \left\{ \begin{array}{lcl} f_1^1(\overrightarrow{x_1}) & = & e_1^1 \\ & \vdots & \\ f_1^{s_1}(\overrightarrow{x_1}) & = & e_1^{s_1} \\ f_1(\overrightarrow{x_1}) & = & f_1^1(\overrightarrow{x_1}) \sqcup ... \sqcup f_1^{s_1}(\overrightarrow{x_1}) \\ \vdots & & \\ f_n^1(\overrightarrow{x_n}) & = & e_n^1 \\ & \vdots & \\ f_n^{s_n}(\overrightarrow{x_n}) & = & e_n^{s_n} \\ f_n(\overrightarrow{x_n}) & = & f_n^1(\overrightarrow{x_n}) \sqcup ... \sqcup f_n^{s_n}(\overrightarrow{x_n}) \end{array} \right.$$

PREUVE [équivalence des systèmes A et A'] : La démonstration découle de la précédente qui prouve l'équivalence des sous-systèmes. □

Remarque 2.15 (Cas particulier) *Si une variable ne possède qu'une seule contrainte et que celle-ci est une contrainte d'égalité alors il est possible de résoudre ce système sans passer par les transformations.*

En conclusion, nous venons de montrer que le système de la page 25 pouvait être résolu en utilisant l'algorithme du Work-Set, après quelques modifications possibles du système à résoudre. Ces deux transformations ont un coût. Un coût en nombre de variables, dans le système de départ nous avions n variables et dans le système résultant nous en avons $n + \sum_i (s_i - 1)$. Un coût en nombre de relations car il nous faut en sauvegarder $\sum_i p_i$ pour la vérification des équations (ces coûts ne tiennent pas compte du cas particulier). Et enfin un coût en calcul car il faut effectuer ces transformations (linéaire sur le nombre de relations) et cette vérification (linéaire sur le nombre d'équations). Ces différents coûts peuvent être considérés comme négligeables par rapport au coût de la résolution proprement dite.

Chapitre 3

La résolution

Dans ce chapitre, nous regarderons comment trouver la plus petite solution d'un système de la forme de celui présenté figure 2.1 page 25. Il s'agit d'un système dont les variables sont d'ordre supérieur, les relations sont des égalités ou des inégalités (de type supérieur ou égal) et il peut y avoir plusieurs relations sur la même variable. Nous supposons qu'il n'y a pas de problèmes avec l'existence et l'accessibilité de la plus petite solution (voir chapitre 2). Nous ne faisons aucune supposition sur la manière dont les problèmes d'existence et d'accéssibilité ont été résolus.

Nous introduisons la notion de forme normale.

Définition 3.1 (Forme normale) *Un système est sous forme normale si et seulement si les parties gauches sont des variables d'ordre supérieur et si les expressions en partie droite possèdent au plus un opérateur.*

Remarque 3.2 (Forme normale) *L'application n'est pas un opérateur.*

La mise en forme normale des systèmes présente plusieurs avantages. Elle permet de simplifier les expressions et de pouvoir facilement parler des opérandes. Il est possible de lier des traitements particuliers pour un opérateur car il est facile de le retrouver. Un dernier avantage, qui n'est pas le moindre, elle permet de limiter naturellement les réévaluations. En effet, une expression qui contient beaucoup de variables et plusieurs opérateurs risque d'être réévaluée souvent à cause du nombre de ses dépendances. Avec la forme normale, la propagation se sera peut-être arrêtée avant car les variations n'engendraient plus rien. Prenons la relation $X_i = e_1 \sqcup e_2$ [1] avec e_1 dépendant d'une variable X_j qui n'apparaît pas dans e_2. Dans ce cas, une variation de X_j entraînera une réévaluation de e_1 mais aussi de e_2 car il faut recalculer X_i et ce quelque soit la nouvelle valeur de e_1. Avec la forme normale, nous aurions attendu le résultat de e_1 pour recalculer X_i s'il y avait eu une variation. Dans ce chapitre, nous supposons que les systèmes à résoudre sont toujours sous forme normale.

Dans ce chapitre nous regarderons comment résoudre les systèmes, pour cela nous allons introduire les graphes de dépendances dans la section 3.1 et nous nous en servirons dans la cadre de l'algorithme du *Work-Set* section 3.2 et dans le cadre d'un algorithme basé sur les composantes fortement connexes section 3.3.

3.1 Définition d'un graphe de dépendances

Nous parlons de dépendances purement syntaxiques. Une variable f_i est dépendante d'une variable f_j, ce que nous noterons $f_j \rightarrow f_i$, si f_j apparaît dans une des expressions e_i^k. Par exemple sur l'équation $X = Y \sqcup Z$, nous pouvons dire que X dépend de Y et de Z. En d'autres termes, une modification de

1. e_1 et e_2 sont des expressions avec un ou plusieurs opérateurs.

la valeur de Y ou de Z risque d'entrainer une modification de la valeur de X. Avant d'évaluer X, il est préférable d'évaluer Y et Z.

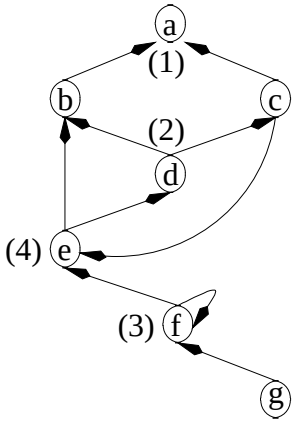

FIG. 3.1 – *Un exemple de graphe de dépendance*

(1) : Une variable peut dépendre de plusieurs variables différentes ($a = \alpha(b, c)$ [2]).

(2) : Plusieurs variables différentes peuvent avoir des dépendances communes ($b = \alpha(d, e)$ et $c = \beta(d)$).

(3) : Une variable peut dépendre d'elle-même de façon directe ($f = \alpha(f, g)$).

(4) : Une variable peut dépendre d'elle même de façon indirecte ($e = \alpha(c, f)$, $d = \beta(e)$ et $c = \gamma(d)$. Ce qui peut se réécrire en $e = \delta(c, d, e, f)$.).

La construction d'un graphe de dépendance se fait très simplement en regardant le système à résoudre et en notant pour chaque relation, la variable qui apparaît à droite ainsi que toutes celles qui apparaissent à gauche. Cette dépendance est purement syntaxique, le graphe de dépendance se calcule sur les variables libres [3] de l'expression considérée. En effet pour une relation telle que $f_i(X_1, X_2) = X_1 \sqcup X_2$ nous ne pouvons pas exhiber de dépendances car elles dépendent des paramètres qui seront passés à f_i lors de l'appel dans une expression. Prenons la notation du λ-calcul, cette fonction s'écrit $f_i = \lambda X_1.\lambda X_2.(\sqcup\ X_1\ X_2)$. Ici, il est clair que $FV\ (f_i) = \emptyset$ d'où aucun calcul de dépendance.

Une idée serait de dire qu'il suffit de regarder tous les appels à f_i et de mettre tous les paramètres en relation avec f_i. Mais ces paramètres peuvent, eux aussi, être des paramètres de la fonction que l'on est en train de définir. Il peut donc être très difficile de gérer ces dépendances.

Bien que cette dépendance ne soit pas marquée explicitement le graphe en tient compte à cause des appels dans les expressions. Par exemple, dans une relation $f_i = \lambda X_1.(X_1\ f_k)$ [4] avec $f_k = ...$ et un appel $f_l = ...(f_i\ f_j)...$, nous notons une dépendance entre f_k et f_i, f_i et f_l et enfin entre f_j et f_l. Donc une variation de f_k sera répercutée sur f_i, bien que f_i ne puisse pas être réévaluée à cause de son paramètre. Il est vrai qu'il est impossible de faire une réévaluation de f_i mais nous pouvons faire, comme en λ-calcul, une β-réduction [Ridoux 98, page 125] sur l'expression $(f_i\ f_j)$ de f_l. Donc f_i a quand même variée ce qui signifie une réévaluation de f_l et donc une réévaluation de $(f_j\ f_k)$, ce qui correspond à f_i. Bien que la dépendance entre f_j et f_k n'ait pas été notée, elle est respectée.

2. Les identificateurs de fonctions α, β, γ et δ n'ont d'intérêt que pour montrer qu'il existe une relation entre les variables.

3. Une variable Y est dite libre dans une expression du λ-calcul s'il n'existe pas de λY qui l'englobe. Autrement cette variable est appelée une variable liée. L'ensemble des variables libres d'une expression e sera noté $FV\ (e)$ [Ridoux 98, pages 65,101].

4. La syntaxe (MN) signifie que nous appliquons M à N. N est le paramètre de M [Ridoux 98, page 67].

Nous allons voir concrètement comment cela marche sur un exemple avec un système (tableau 3.1) et le graphe de dépendance qui en découle (figure 3.2). Nous utiliserons la syntaxe du λ-calcul. Pour le graphe, nous utiliserons les indices des variables comme noeuds, pour ne pas le surcharger (nous noterons i la variable f_i). Il ne faut pas chercher à savoir ce que peut signifier ce système, il n'est là que pour montrer comment les variables peuvent interagir entre elles et ce que donne un graphe de dépendance. En se basant sur cet exemple, nous pouvons donner une version plus formelle du graphe. En effet si nous regardons ce qui se passe, par exemple, pour f_4, nous avons $FV\ (f_4) = \{f_1, f_2, f_3\}$ et sur le graphe $Pred\ (4) = \{1, 2, 3\}$ [5]. Ce qui peut se traduire pour la construction du graphe par

$$Pred\ (i) = FV\ (i)$$

$$\begin{array}{lcl}
f_1 & = & \lambda X_1.X_1 \\
f_2 & = & \lambda X_1.\lambda X_2.\lambda X_3.(\sqcup\ (X_1\ X_2)\ X_3) \\
f_3 & = & \lambda X_1.\lambda X_2.(f_3\ (f_8\ X_1)\ (f_8\ X_2)) \\
f_4 & = & \lambda X_1.\lambda X_2.(f_2\ f_1\ (\sqcup\ X_1\ X_2)\ (f_3\ X_1\ X_2)) \\
f_5 & = & \lambda X_1.\lambda X_2.(f_4\ X_1\ X_2) \\
f_6 & = & \lambda X_1.(f_4\ (f_8\ X_1)) \\
f_7 & = & \lambda X_1.\lambda X_2.\lambda X_3.(\sqcap\ (f_4\ X_1\ X_2)(f_5\ X_1\ X_3)) \\
f_8 & = & \lambda X_1.(f_5\ X_1\ (f_6\ X_1))
\end{array}$$

TAB. 3.1 – *Un exemple de système*

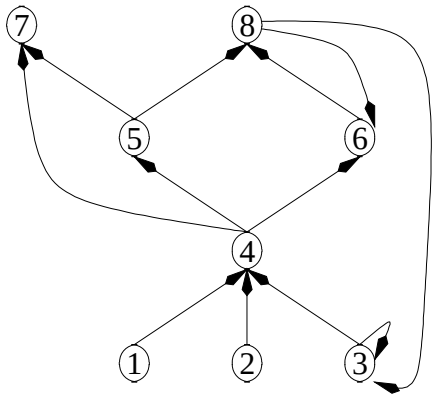

FIG. 3.2 – *Graphe de dépendance du système tableau 3.1*

3.2 Algorithme du *Work-Set*

Nous présentons un algorithme du *Work-Set* basé sur l'algorithme de Gary Kildall ([Kildall 73]) avec en plus, l'utilisation du graphe de dépendance ainsi qu'une gestion plus fine des arcs transitifs.

ALGORITHME — *Work-Set*

—— Partie préliminaire ——

$S', C :=\ transformation_1\ (S)$;

5. Nous notons $Pred\ (i)$ l'ensemble des prédécesseurs du noeud i, c-à-d. l'ensemble $\{j \mid j\ \rightarrow\ i\ \}$.

$simplification$;
$S'' := \ transformation_2 \ (S')$;
$construction$;
Pour Tout $i \in \{1, .., n\}$ **Faire**
  $\phi_i := \ \perp$;
  $\tau_i := \ \perp$;
FinPour
$W := \ init_W \ (S'')$;
—— Partie itérative ——
Répéter
  $I := \ strat\acute{e}gie \ (W, \phi)$;
  Pour Tout $i \in I$ **Faire**
    $\tau_i := \ e_i \phi$;
  FinPour
  $W := \ modifier \ (W, I, \phi, \tau)$;
  $\phi := \ \tau$;
Jusqu'à $I = \emptyset$;
$test \ (C)$;
FIN *Work-Set*

Les ensembles C, S, S' et S'' contiennent l'ensemble des relations du système, avant et après modification. L'ensemble W contient les équations qui sont à traiter (le *Work-Set*). Le vecteur ϕ contient la valeur actuelle de chaque variable alors que le vecteur τ contient les valeurs des variables modifiées à cette itération.

Remarque 3.3 (Type des ensembles C, S, S', S'' et W) *Nous supposerons que chaque variable est de la forme f_i avec i allant de 1 à n (il suffit de faire un renommage des variables). Dans la suite nous nous réfèrerons aux indices pour parler à la fois de la variable et de l'équation, le sens exact sera donné explicitement ou par le contexte. Chacun des ensembles contient des entiers compris entre 1 et n.*

L'algorithme se décompose en deux phases, la première partie (section 3.2.1) est constituée de préliminaires qui servent à préparer la résolution proprement dite et la deuxième partie (section 3.2.2) est l'itération pour rechercher la plus petite solution.

3.2.1 Les préliminaires

Dans cette partie, nous allons regarder les préliminaires, c-à-d. toutes les fonctions qui préparent le système pour la résolution. Nous verrons les transformations, la simplification, la construction du graphe de dépendance ainsi que l'initialisation de l'ensemble des expressions à traiter.

3.2.1.1 $transformation_1$ et $transformation_2$

$transformation_1$ permet de transformer un système S composé d'équations et d'inéquations en deux systèmes. Le premier système S' devient le système à résoudre alors que le deuxième système C est le système de contraintes que la solution devra vérifier pour être valide. Les équations sont transformées en deux inéquations, la relation supérieure ou égale est mise dans S' et l'autre est mise dans C (le système 2.1 est décomposé en un système 2.2 à résoudre et un système 2.3 à vérifier). Les variables qui ne possèdent

qu'une seule relation d'égalité peuvent ne pas être tranformées.

$transformation_2$ permet de transformer un système S composé d'inéquations en un système d'équations S'. Cette fonction transforme tous les systèmes de S de la forme :

$$S = \left\{ \begin{array}{lll} & \geq & e_i^1 \\ f_i(\overrightarrow{x_i}) & \vdots & \\ & \geq & e_i^{s_i} \end{array} \right.$$

en un système de la forme :

$$S' = \left\{ \begin{array}{lll} f_i^1(\overrightarrow{x_i}) & = & e_i^1 \\ \vdots & & \\ f_i^{s_i}(\overrightarrow{x_i}) & = & e_i^{s_i} \\ f_i(\overrightarrow{x_i}) & = & f_i^1(\overrightarrow{x_i}) \sqcup ... \sqcup f_i^{s_i}(\overrightarrow{x_i}) \end{array} \right.$$

Pour les systèmes ne comportant qu'une seule équation de la forme $f_i(\overrightarrow{x_i}) = e_i^j$, ceux-ci sont directement ajoutées à S'. Au renomage des variables près, nous respectons bien l'hypothèse d'avoir n équations à n inconnues (remarque 3.3).

3.2.1.2 *simplification*

Avant d'effectuer l'opération $transformation_2$ il est possible d'ajouter une opération de simplification du système. Cette opération peut se faire sur la composition de deux relations, les constantes...

Composition de deux relations

$$\begin{array}{lll} f_i(\overrightarrow{x_i}) & \geq & e_i^j \\ f_i(\overrightarrow{x_i}) & \geq & e_i^k \\ & \equiv & \\ f_i(\overrightarrow{x_i}) & \geq & e_i^j \sqcup e_i^k \end{array}$$

À quelles conditions cette opération est-elle utile? Cette opération peut être intéressante car $\forall X, X \sqcup X = X \sqcap X = X$. Cela permet de simplifier les expressions, de limiter les recherches de valeur ainsi que de diminuer les calculs. De plus, deux relations sur une même variable ayant les mêmes dépendances serons à recalculer en même temps. Soit F_i^j l'ensemble des indices k des f_k apparaissant dans e_i^j ($F_i^j = FV\ (e_i^j)$). $|F_i^j|$ [6] est le nombre d'évaluations à faire pour pouvoir évaluer e_i^j. Pour que l'opération de simplification soit efficace, il faut que le nombre d'évaluations de l'expression simplifiée soit très inférieur au nombre d'évaluation des deux expressions prises séparément (nous supposons qu'une recherche de valeur pour une variable est négligeable). Il suffit donc que $(|F_i^{jk}| = |F_i^j \cup F_i^k|) \ll (|F_i^j| + |F_i^k|)$. La condition idéale pour que la simplification soit optimale est quand $F_i^j = F_i^k$. Cette opération est d'autant plus intéressante si le treillis est distributif et si nous pouvons faire un travail syntaxique sur les expressions ou si nous avons une recherche de valeur optimale (si une variable apparaît plusieurs fois dans une expression ne faire qu'une seule recherche).

Cas des constantes

6. $|F|$ indique le cardinal de F

Plusieurs opérations sont possibles lorsque les expressions sont des constantes. Quand il existe une seule contrainte d'égalité sur une variable dont l'expression est une constante, il est possible de supprimer cette variable du système en remplaçant toutes ses occurrences par la constante. C'est un phénomène de propagation de constantes classique utilisée dans plusieurs domaines (par exemple dans le domaine de l'évaluation partielle [Hornof 97]). Il n'est pas rare de voir cette propagation accompagnée de calculs sur les constantes. Cette propagation et ces calculs ont un coût et il n'est pas évident que dans notre cadre, nous ayons besoin d'un outil aussi puissant. Ce coût peut être concidéré comme négligeable car ce calcul n'est pas dans l'itération.

S'il y a plusieurs contraintes d'inégalités sur une variable et que l'une d'entre elle a comme expression une constante, il est possible d'utiliser cette constante comme valeur initiale de la variable dans l'algorithme. C'est vrai pour deux raisons, la première est que les fonctions sont monotones (sous-entendu croissantes) donc une fois cette valeur atteinte nous ne pouvons plus trouver une valeur qui lui est inférieure. La deuxième raison est que dans le cas des inégalités nous prenons le *lub*, en effet pour que plusieurs inéquations soient vérifiées, il faut prendre la plus grande valeur.

Sous-ensembles indépendants

Soit S l'ensemble des relations du système. Deux variables f_i et f_j sont indépendantes si quelque soit la modification de l'une d'entre elles, l'autre ne peut pas changer. Dans ce cas, si f_i dépend de f_k (et inversement) alors f_j et f_k sont indépendantes. Il existe donc deux groupes de dépendances, un pour f_i et un pour f_j. Soit S^k l'ensemble des relations du groupe de dépendance de f_k alors $S = S^i \cup S^j$ avec $S^i \cap S^j = \emptyset$. Donc la résolution de S se ramène à deux résolutions indépendantes de S^i et S^j. Soit ϕ^k le vecteur solution de S^k alors pour $S^k = S^i \cup S^j$, $\phi^k = (\phi^i, \phi^j)$. Si S se décompose en S^i, pour $i = 1, .., k$, ensembles indépendants alors la solution ϕ de S est $\phi = (\phi^1, ..., \phi^k)$. A l'ordre 0, il est possible de trouver ces groupes de dépendances alors qu'a l'ordre $n, n \geq 1$ cela devient impossible à cause des paramètres des fonctions.

3.2.1.3 $construction$

$construction$ permet de construire le graphe de dépendances présenté dans la section 3.1. Pour ce faire, nous introduisons une structure de donnée nous permettant de travailler sur les équations. Nous pouvons accéder à une équation avec la fonction $extraction$ qui prend en paramètre l'indice de l'équation (voir la remarque 3.3) et l'ensemble dans lequel elle se trouve. Avec une équation nous pouvons avoir sa partie droite avec $partie - droite$ qui correspond à l'expression et sa partie gauche avec $partie - gauche$ qui est l'indice de la variable. Enfin sur une expression, nous pouvons trouver l'ensemble de ses variables libres avec FV. Pour le graphe, nous pouvons créer et supprimer des noeuds ($créer - noeud$ et $supprimer - noeud$) ainsi que des liens ($créer - lien$[7] et $supprimer - lien$). Ce qui nous donne l'algorithme de $construction$ suivant :

7. Les arcs sont orientés, $créer$-$lien$ (i, j) construit un arc allant de i vers j.

ALGORITHME — *construction*

Pour Tout $i \in \{1, .., n\}$ **Faire**
  $cr\acute{e}er\text{-}noeud\ (i)$;
FinPour
Pour Tout $i \in \{1, .., n\}$ **Faire**
  $extraction\ (i, S)$;
  $T :=\ FV\ (partie\text{-}droite\ (i))$;
  Pour Tout $j \in T$ **Faire**
    $cr\acute{e}er\text{-}lien\ (j, i)$;
  FinPour
FinPour

FIN — *construction*

3.2.1.4 $init_W$

Une première solution est de dire qu'il suffit de mettre dans W les premières équations du système et de laisser les dépendances faire leur travail. Nous allons voir sur un exemple que cela ne marche pas. Prenons un système avec deux équations :

$$\begin{array}{rcl} X & = & \bot \\ Y & = & X \sqcup \top \end{array}$$

Le graphe de dépendance est le suivant :

$$\begin{array}{c} Y \\ \uparrow \\ X \end{array}$$

Supposons que nous ne mettions que X dans W, à la première itération X ne varie pas et donc Y n'est pas rajouté à W. La solution donnée par l'algorithme est alors $X = \bot$ et $Y = \bot$. Il est évident que cette solution n'est pas bonne. Nous sommes donc obligés d'évaluer toutes les expressions au moins une fois. Pour ce faire, nous mettons toutes les équations dans W, d'où l'algorithme suivant.

ALGORITHME — $init_W\ ()$

Retourne ($\{1, .., n\}$) ;

FIN — $init_W\ ()$

3.2.2 L'itération

Nous allons nous intéresser ici à la partie itérative de l'algorithme. Nous allons voir comment sélectionner les variables à évaluer ainsi que l'impact sur l'ensemble W. Nous regarderons aussi comment il est possible de gérer le test de validation.

3.2.2.1 $modifier$

Dans un premier temps, nous allons nous occuper de la fonction $modifier$. Cette fonction permet de modifier l'ensemble W, en fonction des variables qui ont été sélectionnées. Comme nous venons d'évaluer les expressions qui avaient été sélectionnées nous pouvons les retirer de W car elles sont maintenant vérifiées. D'où l'expression de la forme $W :=\ (W \setminus I)$[8]. Mais cela ne suffit pas, il faut aussi tenir compte de l'impact des évaluations sur les autres expressions. Pour ce faire, nous utilisons le graphe de dépendance et nous obtenons $W :=\ (W \setminus I) \cup I!$. L'opération $I!$ permet de calculer l'ensemble des successeurs de tous

8. L'expression $(A \setminus B)$ dans la théorie des ensembles est l'ensemble $\{a \in A \mid a \notin B\}$.

les éléments de I dans le graphe de dépendances. Elle peut être définie comme suit :

$$I! = \{j \mid \exists i \in I : i \rightarrow j\}.$$

Cette formulation de la fonction $modifier$ semble satisfaisante. Dans cette formulation, nous rajoutons automatiquement tout ce qui risque de bouger. Mais si la variable que l'on vient de réévaluer n'a pas variée elle ne peut pas entraîner de variation. De plus si nous ne tenons pas compte de cette «non variation» nous risquons d'avoir une non terminaison de l'algorithme, à cause des boucles (si $i \rightarrow j$ et $j \rightarrow i$ alors l'évaluation de i implique l'évaluation de j qui elle-même implique l'évaluation de i etc.). D'où la formulation pour la fonction $modifier$:

$$W := (W \setminus I) \cup \{i \in I : \phi_i \neq \tau_i\}!.$$

Nous pouvons faire un dernier raffinement de la modification. Il est possible que les opérateurs aient un comportement spécial pour certaines valeurs. Par exemple le *lub* rend $\top$ dès qu'un des deux arguments vaut $\top$. Avec la monotonie, nous pouvons garantir que la variable valant $\top$ ne variera plus et que toute modification de la deuxième variable sera sans effet sur l'expression. Soit l'expression $X = Y \sqcup Z$ avec $Y = \top$, toute modification de Z sera sans impact sur X, donc X ne dépend plus de Z. Pour éviter des réévaluations que nous savons inutiles, il suffit de faire $supprimer\text{-}lien\ (Z, X)$ et $supprimer\text{-}lien\ (Y, X)$. Les variations de Z ne seront plus répercutées sur X. De même, nous savons que Y ne variera plus et donc qu'il ne dépend plus de rien. Nous pouvons donc supprimer tous les arcs entrants de Y, c-à-d. tous les arcs tels que $T \rightarrow Y$. Ces traitements sont faciles à intégrer car nous travaillons sur un système sous forme normale. Il est possible de donner un traitement particulier pour chaque type d'opérateurs. Le *glb* a lui aussi une particularité, elle est plus difficile à mettre en oeuvre. Pour une expression $X = Y \sqcap Z$, X vaudra $\bot$ tant que l'une de ces deux opérandes vaudra $\bot$. Il est donc inutile d'évaluer X si Y ou Z vaut $\bot$. Elle est plus délicate à mettre en oeuvre car ici nous ne devons pas supprimer les arcs mais seulement les invalider temporairement. Les deux arcs ne seront valides que lorsque les deux variables Y et Z auront des valeurs différentes de $\bot$. Il existe encore une valeur particulière pour les deux opérateurs. Nous avons aussi $X \sqcup \bot = X \sqcap \top = X$.

3.2.2.2 $stratégie$

La fonction $stratégie$ permet de sélectionner quelles sont les expressions qui sont à évaluer à ce pas de l'itération. L'idée est que toutes les variables dont nous avons calculé toutes les dépendances, sont à évaluer. En termes plus mathématiques, cela donne l'ensemble $\{j \mid \nexists i \in W : i \rightarrow j\}$. Le problème de cette formulation, c'est qu'elle ne gère pas les boucles. En effet, en présence de boucles, il est impossible de trouver une variable qui ne dépend plus de rien. Dans le cas d'une boucle, une variable ne peut être évaluée que si elle ne dépend que transitivement d'elle même. Nous prenons donc l'ensemble suivant (Méthode 1):

$$\{j \in W \mid \text{si } \exists i \in W : i \rightarrow^+ j \text{ alors } j \rightarrow^+ i\}\ ^9\ .$$

Cette formulation garantit qu'il n'y aura pas de propagation des résultats. Dans un schéma où Y dépend de X, il ne faut pas évaluer Y avant d'évaluer X et c'est d'autant plus vrai si X est une boucle. Dans ce cas, Y dépend transitivement de toutes les variables contenues dans la boucle. Donc tant qu'il existe une variable de la boucle dans W la variable Y ne sera pas sélectionnée. Il y aura propagation de la valeur de X quand celle-ci sera trouvée. Les boucles sont traitées de façon globale, nous sélectionnons le maximum d'éléments de la boucle. Pourquoi prendre plusieurs éléments en sachant très bien que cela va provoquer des réévaluations de variables que l'on vient de choisir? Il est possible de n'en choisir qu'une et de laisser les dépendances jouer leurs rôles. Prenons une boucle simple $i \rightarrow j$, $j \rightarrow k$ et $k \rightarrow i$ et supposons que nous choisissons la variable i pour commencer l'itération (Méthode 2). Un exemple est donné dans le tableau

9. La fermeture transitive de la relation $\rightarrow$: $i \rightarrow j$ et $j \rightarrow k$ implique $i \rightarrow^+ k$.

suivant. $X_a = \delta(Y_b)$ signifie que la variable X à l'itération a est calculée avec l'expression δ et la valeur de la variable Y calculée à l'itération b.

$Méthode\ 1$			$Méthode\ 2$		
i_1	$=$	$\alpha(k_0)$	i_1	$=$	$\alpha(k_0)$
j_1	$=$	$\beta(i_0)$	j_1	$=$	$\beta(i_1)$
k_1	$=$	$\gamma(j_0)$	k_1	$=$	$\gamma(j_1)$
i_2	$=$	$\alpha(k_1)$	i_2	$=$	$\alpha(k_1)$
j_2	$=$	$\beta(i_1)$	j_2	$=$	$\beta(i_2)$
k_2	$=$	$\gamma(j_1)$	k_2	$=$	$\gamma(j_2)$

Il est facile de voir qu'il est plus avantageux, en nombre d'évaluations, d'utiliser la Méthode 2. Avec la méthode 2, la première évaluation de k est faite avec j_1 alors qu'avec la première méthode cela ne se produisait qu'à la deuxième évaluation de k. La seule «perte» de la méthode 2 est sur la variable qui a été choisie au départ. Ce qui est normal car nous forçons son évaluation, toutes les variables dont elle dépend n'ont pas pu être calculées. Cela se passe aussi avec la méthode 1, à la différence près que c'est valable pour toutes les variables.

Pour mettre en oeuvre la méthode 2, nous allons modifier l'algorithme du *Work-Set*.

ALGORITHME — *Work-Set* MODIFIÉ

$S', C :=\ transformation_1\ (S)$;
$simplification$;
$S'' :=\ transformation_2\ (S')$;
$construction$;
Pour Tout $i \in \{1, .., n\}$ **Faire**
  $\phi_i :=\ \perp$;
FinPour
$W :=\ init_W\ (S'')$;
Répéter
  $I :=\ stratégie\ (W, \phi)$;
  $\tau :=\ \phi$;
  $J :=\ I$;
  TantQue $J \neq \emptyset$ **Faire**
    $i :=\ prendre\ (J)$;
    $\phi_i :=\ e_i\phi$;
  FinTantQue
  $W :=\ modifier\ (W, I, \phi, \tau)$;
Jusqu'à $I = \emptyset$;
$test\ (C)$;

FIN — *Work-Set* MODIFIÉ

L'exemple est simple, le choix de la variable de départ n'est pas important car tous les choix sont équivalents, à la variable de départ près. Dans un système plus complexe, les choix ne sont pas forcément équivalents.

Il nous faut définir la fonction $prendre$, elle permet de sélectionner la variable qui va être évaluée.

ALGORITHME — $prendre\ (J)$

Si $\exists i \mid \nexists j \in J : j \rightarrow i$
Alors Retourne (i) ;
Sinon Retourne ($Choix\ (J)$) ;

FIN — $prendre\ (J)$

La fonction $Choix$ permet de «casser» une boucle, elle sélectionne la variable dont nous allons forcer l'évaluation. Dans un premier temps, cette fonction retournera la première variable de l'ensemble ($i \in J \mid \nexists j \in J : j < i$) ou une variable tirée arbitrairement ($i \in J$). Cette version n'est pas optimum, mais elle est simple à mettre en oeuvre.

Dans un deuxième temps, nous pouvons regarder s'il n'est pas possible d'utiliser des informations sur les boucles. Les informations que nous avons ne sont pas nombreuses : le nombre de boucles, la taille des boucles et les variables impliquées dans les boucles. Il faut bien voir que la fonction $strat\acute{e}gie$ extrait la plus grande boucle possible et il peut avoir une répétition d'au moins un des noeuds. Prenons l'exemple représenté sur la figure 3.2. À la deuxième itération, la stratégie nous donne l'ensemble $\{3, 4, 5, 6, 8\}$, ce qui correspond à une boucle $\{3, 4, 5, 8, 3, 4, 6, 8, 3\}$ (b0). Cet ensemble représente aussi quatre boucles simples (c-à-d. sans répétition) qui sont $\{3\}$ (b1), $\{3, 4, 5, 8\}$ (b2), $\{3, 4, 6, 8\}$ (b3) et $\{6, 8\}$ (b4).

Dans le cas d'une boucle simple, nous avons dit que tous les choix sont équivalents. En fait tous les choix ne sont pas réellement équivalents, il y a une information supplémentaire, celle de point d'entrée. Un point d'entrée est un noeud qui dépend de variables qui n'apparaissent pas dans la boucle (par exemple le noeud 4 qui dépend de 3 et de $1, 2$). Ces variables sont légèrement différentes car elles ont déjà vu certaines de leurs dépendances varier. Il serait donc intéressant de les utiliser tout de suite.

Dans le cadre de l'exemple nous avons une boucle complexe (b0), il nous faut donc «casser» quatre boucles et non pas une (b1, b2, b3 et b4). En choisissant le 8, nous pouvons casser trois boucles (b2, b3 et b4) car c'est un noeud commun aux trois boucles. Pour «casser» la dernière boucle, le choix est vite fait, nous prenons le 3. Mais ce noeud apparaît dans deux autres boucles et donc deux des quatre boucles sont «cassées» deux fois. Nous forçons donc deux évaluations au lieu d'une. Nous ne sommes pas obligés de choisir ces deux noeuds, un seul suffit. Comme tous les noeuds appartiennent à la même grande boucle la variation d'un élément se répercute sur tous les éléments. Nous pouvons nous contenter de sélectionner un seul élément.

Pour définir cette fonction $Choix$, nous allons nous servir des points d'entrées et des variables communes à plusieurs boucles simples. Il suffit de choisir parmi les variables d'entrées celle qui est commune à un maximum de boucles simples. Si plusieurs variables peuvent être choisies, nous en prenons une arbitrairement.

ALGORITHME — $Choix\ (J)$

$PE := \emptyset$;
Pour Tout $i \in J$ **Faire**
  $PE := PE \cup \{< i, entr\acute{e}e\ (i), boucle\ (i) >\}$;
FinPour
$PE := max - entr\acute{e}e\ (PE)$;
$PE := max - boucle\ (PE)$;
Retourne ($prendre - un\ (PE)$) ;

FIN — $Choix\ (J)$

$entr\acute{e}e\ (i)$ permet de calculer le nombre d'entrée du noeud i sans tenir compte des entrées transitives, c-à-d. le cardinal de l'ensemble $\{j \notin J \mid j \rightarrow i\}$. $boucle\ (i)$ calcule le nombre de circuit hamilto-

nien[10] passant par le noeud i. $max - entrée\ (E)$ donne les éléments de E qui ont le maximum d'entrée et $max - boucle\ (E)$ ceux qui ont le maximum de circuits hamiltoniens. $prendre - un\ (E)$ permet d'extraire un élément de E arbitrairement, sans le supprimer de E.

La recherche de circuit hamiltonien est une opération très coûteuse. Elle l'est d'autant plus qu'ici nous sommes dans une itération : ce calcul sera répété très souvent. Nous pouvons nous contenter d'une approximation de ce nombre de circuits. Plus cette approximation sera bonne plus le résultat sera bon. Inversement, avec une mauvaise approximation, il est possible d'avoir une grande perte en terme d'évaluation.

3.2.2.3 $test$

La fonction $test\ (C)$ permet de vérifier que toutes les relations contenues dans C sont vérifiées. S'il en existe une qui n'est pas satisfaite alors nous concluons que la solution trouvée est invalide. Nous aimerions pouvoir intégrer le test dans la recherche du point fixe pour pouvoir arrêter la recherche plus rapidement. Si nous trouvons une relation $f_i(\overrightarrow{x_i}) \leq e_i^j$ qui ne peut plus être vérifiée, nous pouvons arrêter la recherche car la solution qui sera trouvée ne sera pas valide. Mais le test ne peut pas être effectué n'importe quand. Nous ne pouvons tester que sur des variables qui ont pris leur valeur définitive. Il n'est pas facile de savoir dans quel état sont les variables après une itération. Ce n'est pas parce qu'une variable n'est plus dans le *Work-Set* qu'elle ne variera plus. Pour savoir à quel moment le test peut être fait, il faudrait faire une analyse pour repérer ces différents endroits, soit statique (avant le début de la recherche) soit dynamique (durant la recherche). Quellle que soit l'analyse choisie, elle entraîne un coût qui risque d'être supérieur au gain.

3.3 Une approche par les Composantes Fortement Connexes

L'approche présentée dans la section 3.2 est une approche dynamique. Le choix des variables à évaluer est fait au cours de l'itération. Dans cette section, nous allons montrer une approche statique dans laquelle tous les choix seront fais avant le début de la résolution proprement dite.

L'approche étudiée dans la section précédente pourrait se résumer ainsi : si une variable i change de valeur alors il faut évaluer tous ses successeurs. Nous pourrions voir le problème sous un angle différent en se disant : si la variable i ne change plus de valeur alors il faut propager cette valeur à ses successeurs. Nous allons étudier cette deuxième approche et pour ce faire nous allons nous servir des composantes fortement connexes. François Bourdoncle a utilisé cette approche, pour déterminer à quels endroits il était possible de mettre des opérateurs d'élargissement ([Bourdoncle 93]).

3.3.1 Construction des Composantes Fortement Connexes

Nous allons décrire un algorithme qui nous permettra de construire un graphe en composantes fortement connexes à partir du graphe des dépendances. Nous allons commencer par déterminer ce qu'est une composante fortement connexe et ensuite nous donnerons l'algorithme de construction.

Définition 3.4 (Composantes Fortement Connexes) *Soit un graphe G. On considère la relation binaire sur les noeuds du graphe :*

$$x \mathcal{R} y \equiv (x \rightarrow^+ y) \wedge (y \rightarrow^+ x)$$

C'est une relation d'équivalence dont les classes s'appellent composantes fortement connexes de G.

Notation [CFC] **:** Pour simplifier l'écriture, nous abrégerons «composante fortement connexe» en CFC.

10. Un circuit est hamiltonien si il passe une fois et une seule par le même sommet, il est sans répétition.

Dans la définition des CFC, nous voyons apparaître la notion de circuit. Quelque soit le graphe de départ G, la transformation nous donne un graphe sans circuit. Nous allons nous en servir sur les graphes de dépendances. Pour l'exemple figure 3.3, la transformation nous donne le graphe représenté figure 3.4.

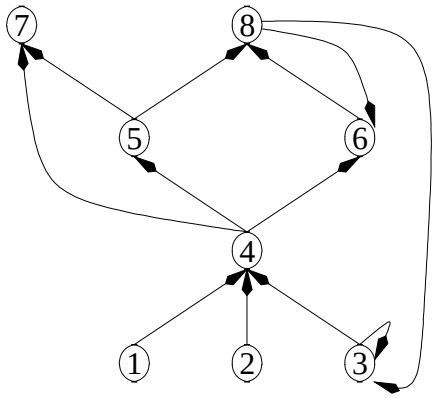

FIG. 3.3 – *Graphe de dépendance du système tableau 3.1*

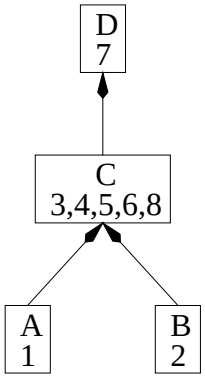

FIG. 3.4 – *Graphe en composantes fortement connexes*

Comme le graphe est sans circuit, il est facile de faire les calculs pour résoudre le système. Il suffit d'évaluer tous les noeuds en respectant l'ordre fournit par le graphe. Un problème subsiste : comment évaluer la composante C (les autres sont de simples expressions)? Pour faire les évaluations, nous disposons d'une procédure, $évaluer - noeud\ (i)$ qui permet d'évaluer le noeud i.

Pour respecter l'ordre d'évaluation du graphe, nous allons donner à chaque composante un attribut de stabilité. Une composante est stable si et seulement si elle vient d'être évaluée. Nous avons une procédure $stabilise - CFC\ (i)$ qui permet de stabiliser une CFC i. Initialement, toutes les CFC sont instables. Il est aussi possible de déstabiliser une CFC ou un noeud avec $déstabilise - CFC\ (i)$. Tant que la CFC n'est pas stable, nous continuons à la traiter.

Une CFC est un circuit qui peut être plus ou moins complexe (un circuit complexe est un circuit qui n'est pas hamiltonien). Nous allons «éclater» ces CFC pour trouver des composantes simples. Pour ce faire nous allons utiliser le point d'entrée que nous avons vu dans la section précédente. Pour «éclater» une CFC, nous lui ôtons son point d'entrée et nous recommençons l'opération de construction des CFC. Il suffit ensuite de rebrancher le point d'entrée avec le graphe que l'on vient de calculer. Pour faire ce branchement, nous avons aussi besoin de définir un point de sortie de la CFC. Pour simplifier le travail, nous définissons comme point

de sortie d'une CFC, le point d'entrée de la CFC. La fonction permettant de faire ce choix est la fonction $Choix$. Il faut relier la CFC avec le graphe résultant de l'éclatement. Cette liaison doit être différenciée des autres liens, nous appelons ce lien, un lien zoom. Ce lien permet d'accéder au graphe lorsque nous voulons évaluer la CFC. Il sera symbolisé par une flèche en zigzag ($\rightsquigarrow$).

Le graphe obtenu est une compilation du graphe de dépendance, c'est un graphe d'ordonnancement. Il nous indique ce qu'il faut que nous traitions. Il faut comprendre une liaison $i \rightarrow j$ par : une fois que la composante i est stable alors il faut traiter la composante j. Attention, cela ne veut pas dire que pour calculer la composante j nous avons besoin de toute la composante i, nous avons sans doute seulement besoin d'une partie des valeurs de la composante i.

G_{init} est le graphe de dépendance qui a été déduit du système. Nous considérons qu'il est utilisable par tous les programmes. En utilisant $construire - graphe$ sur l'exemple 3.2 page 40, nous obtenons la décomposition représentée par la figure 3.5.

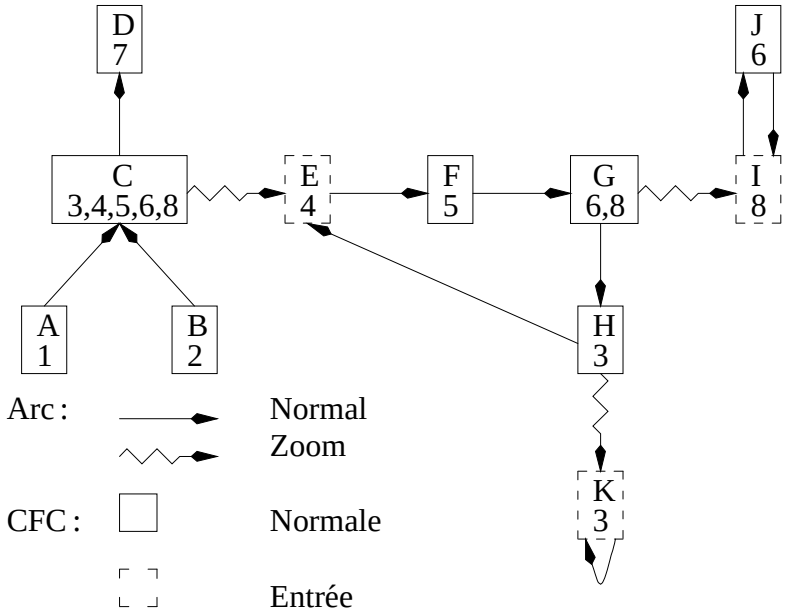

FIG. 3.5 – *Graphe en composantes fortement connexes complet*

Pour construire la décomposition d'un graphe en CFC, nous utiliserons l'algorithme de Robert Tarjan [Tarjan 83]. Le résultat de l'algorithme est un graphe dont les noeuds (les CFC) sont composés d'un nom et de l'ensemble des variables correspondantes à la CFC. Nous allons nous servir de cette construction pour écrire l'algorithme de $construire - graphe\ (G)$.

ALGORITHME $construire - graphe\ (G)$

$G' := Tarjan(G)$;
Pour Tout $c \in composante - CFC\ (G')$ **Faire**
$i := Choix\ (c.ensemble, G_{init})$;
$P := pred(i, G_{init}|_{c.ensemble})$;
$S := succ(i, G_{init}|_{c.ensemble})$;
$G'' := G' \cup_{zoom} < construire - graphe\ (G_{init}|_{c.ensemble \setminus \{i\}}), c, i, P, S >$;
FinPour
Retourne ($minimalise\ (G'')$ **)** ;

FIN $construire - graphe\ (G)$

L'opérateur $\cup_{zoom}$ prend deux opérandes, un graphe et un quintuplet $< G', c, i, P, S >$ (un graphe, une CFC, un noeud, les successeur de i dans c, les prédécesseurs de i dans c). Voici la description du résultat de $G \cup_{zoom} < G', c, i, P, S >$:

- Union des deux graphes G et G'.
- Création d'un noeud N pour le point i.
- Création d'un lien zoom entre c et N.
- Création des liens normaux entre les CFC correspondantes aux éléments de P et le noeud N.
- Création des liens normaux entre le noeud N et les CFC correspondantes aux éléments de S.
- Création d'un nouveau noeud s'il existe $i \rightarrow i$ et mise à jour des liaisons.

$pred(i, c)$ (respectivement $succ(i, c)$) permet de trouver tous les successeurs (respectivement prédécesseurs) du noeud i dans le graphe c. La fonction $composante - CFC\ (G)$ permet de déterminer quels sont les noeuds de G qui correspondent à des CFC. Aucune fonction ne peut traverser un lien zoom. Nous avons à notre disposition la fonction $zoom(c)$ qui nous permet de zoomer sur le noeud c. Elle rend le sous-graphe associé à c par le lien zoom.

ALGORITHME $minimalise\ (G)$

Pour Tout $i \in G$ **Faire**
Si $\exists j, k \in G : i \rightarrow^+ j \wedge j \rightarrow^+ k \wedge i \rightarrow k \wedge k \neq i$
Alors $supprimer\text{-}lien\ (i, k)$;
FinSi
FinPour

FIN $minimalise\ (G)$

La procédure $minimalise$ permet de minimaliser un graphe en lui ôtant tous les arcs transitifs excepté les arcs $i \rightarrow i$.

PREUVE [terminaison de $construire - graphe$] : Le graphe de départ est un graphe fini. L'algorithme de Tarjan termine et nous rend le graphe décomposé en CFC. Ce graphe est aussi fini. $Choix$, $pred$ et $succ$ terminent. $minimalise$ termine car le graphe est fini. $\cup_{zoom}$ termine si $construire - graphe$ termine. Nous allons regarder le nombre d'éléments lors des appels à $construire - graphe$. Nous avons $construire - graphe\ (G)$ avec n éléments pour G. $G_{init}|_{c.ensemble \setminus \{i\}}$ est un graphe avec m éléments et $m < n$. Si nous avons toujours des CFC (c-à-d. que l'algorithme continue), nous avons un nombre d'éléments qui converge vers 0 et à 0 l'algorithme s'arrête. □

3.3.2 Résolution du système

Maintenant que nous avons le graphe en composantes fortement connexes, il nous faut donner un algorithme capable de s'en servir pour résoudre le système. Cet algorithme est basé sur la recherche par approximations successives du plus petit point fixe.

ALGORITHME _____ $\acute{e}valuer - graphe\ (G)$
  $retour :=$ **Faux** ;
  TantQue $\exists i \in G : \forall j \rightarrow i : stable - CFC\ (j) \wedge \mathbf{Non}(stable - CFC\ (i))$ **Faire**
    $retour :=\ retour \vee \acute{e}valuer - noeud\ (i)$;
    $stabilise - CFC\ (i)$;
  FinTantQue
  Retourne ($retour$) ;
FIN _____ $\acute{e}valuer - graphe\ (G)$

La variable $retour$ nous permet de savoir si quelque chose a varié durant l'évaluation. Nous avons vu qu'il y avait deux sortes de noeuds, les noeuds correspondant à des expressions et les noeuds correspondant à des CFC. La procédure $\acute{e}valuer - noeud$ permet de déterminer de quel type de noeud il s'agit et lance la procédure d'évaluation correspondante. Dans le cas d'une CFC, il ne faut pas oublier de zoomer pour extraire le sous-graphe.

ALGORITHME _____ $\acute{e}valuer - noeud\ (i)$
  Si $variable\ (i)$
    Alors Retourne ($\acute{e}valuer - expression\ (i)$) ;
    Sinon Retourne ($\acute{e}valuer - CFC\ (zoom\ (i))$) ;
  FinSi
FIN _____ $\acute{e}valuer - noeud\ (i)$

$\acute{e}valuer - expression$ permet d'évaluer une expression et elle nous indique si la variable a subie une variation.

ALGORITHME _____ $\acute{e}valuer - expression\ (i)$
  Si $\phi_i \neq \acute{e}valuer\ (i)$
    Alors
      $\phi_i :=\ \acute{e}valuer\ (i)$;
      Retourne (**Vrai**) ;
    Sinon Retourne (**Faux**) ;
FIN _____ $\acute{e}valuer - expression\ (i)$

Pour évaluer une CFC, il suffit d'extraire le noeud d'entrée et de l'évaluer. Pour le reste de la CFC, nous stabilisons le point d'entrée et évaluons le graphe. Cette opération continue tant qu'au moins une variable de la CFC a varié.

ALGORITHME ———— $\acute{e}valuer - CFC\ (C)$

$i :=\ s\acute{e}lectionne - entr\acute{e}e(C)$;
$retour :=$ **Faux** ;
Répéter
$varie :=$ **Faux** ;
$stabilise - CFC\ (i)$;
$varie :=\ varie \vee \acute{e}valuer - noeud\ (i)$;
$d\acute{e}stabilise - CFC\ (C \setminus \{i\})$;
$varie :=\ varie \vee \acute{e}valuer - graphe\ (C)$;
$retour :=\ retour \vee varie$;
Jusqu'à $varie =$ **Faux** ;
Retourne ($retour$) ;

FIN ———— $\acute{e}valuer - CFC\ (C)$

La fonction $s\acute{e}lectionne - entr\acute{e}e(C)$ permet d'extraire le point d'entrée de la CFC C.

PREUVE [terminaison de $\acute{e}valuer - graphe$] : Lors de l'appel à $\acute{e}valuer - graphe\ (G)$ avec n éléments à G, l'algorithme termine en n appels à $\acute{e}valuer - noeud$. La fonction $\acute{e}valuer$ termine donc $\acute{e}valuer - noeud$ termine si la fonction $\acute{e}valuer - CFC$ termine. La fonction $\acute{e}valuer - CFC$ boucle tant que quelque chose varie, c-à-d. tant que des variables varient. Nous n'aurons plus de variations quand pour toutes les variables $f_i(\overrightarrow{x_i}) = e_i$, c-à-d. quand nous aurons atteint un point fixe. Or nous savons qu'il existe toujours un point fixe, donc $\acute{e}valuer - CFC$ termine. Par conséquent, la fonction $\acute{e}valuer - graphe$ termine et le résultat est un point fixe pour le système. □

PREUVE [plus petit point fixe] : La preuve est la même que pour l'algorithme du *Work-Set*. L'algorithme de résolution par itérations successives fait une sous-estimation du plus petit point fixe, une fois atteint ce point fixe, il s'arrête. □

Avec les fonctions décrites précédement, il devient facile d'écrire l'algorithme de résolution.

ALGORITHME ———— $R\acute{e}solution$

$S', C :=\ transformation_1\ (S)$;
$simplification$;
$S'' :=\ transformation_2\ (S')$;
$construction$;
Pour Tout $i \in G_{init}$ **Faire**
$\phi_i :=\ \perp$;
FinPour
$G' :=\ construire - graphe\ (G_{init})$;
$\acute{e}valuer - graphe\ (G')$;
$test\ (C)$;

FIN ———— $R\acute{e}solution$

PREUVE [terminaison de $R\acute{e}solution$] : Évident suite aux preuves de terminaisons des algorithmes $construire - graphe$ et $\acute{e}valuer - graphe$. □

Les fonctions $transformation_1$, $simplification$, $transformation_2$ et $construction$ sont les mêmes que

celles présentées dans le cadre du *Work-Set*.

Remarque 3.5 (Place du test) *Nous avons le même problème qu'avec l'algorithme du* Work-Set *(voir section 3.2.2.3). Nous aimerions faire entrer le test dans la résolution. Contrairement à ce qui ce passait, ici nous savons exactement à quel moment une variable a pris sa valeur définitive. Il suffit de regarder le graphe de premier degré (nous ne tenons pas compte des arcs zoom), à la sortie d'un noeud toutes les variables du noeud ont prises leur valeur définitive. Il suffit de tester à ce moment là. À la sortie du noeud contenant une certaine variable i, nous sommes sûrs qu'elle ne variera plus car si elle devait encore varier nous ne serions pas sortis du noeud. L'analyse ne nous a rien coûté puisqu'elle était déjà faite.*

Remarque 3.6 (Passer de récursif à itératif) *Il est possible de passer de cet algorithme d'évaluation récursif à un algorithme itératif à la Kildall. Nous pouvons utiliser deux méthodes, la première est d'utiliser une structure plus riche qu'un simple ensemble pour W et l'autre est d'avoir des éléments plus riches que le simple numéro de l'équation à évaluer.*

La première méthode utilise une pile. Il suffit d'avoir en sommet de pile les expressions qui sont à évaluer. Pour initialiser la pile, il faut parcourir l'arbre à l'envers et empiler tous les noeuds rencontrés. Quand nous avons une CFC à évaluer, il faut l'empiler en partant des prédécesseurs du noeud d'entrée.

La deuxième méthode utilise un ensemble dont les éléments sont les numéros des variables indicées par le nombre d'antécédents non stabilisés. Pour savoir quelles sont les expressions à évaluer il suffit de prendre celles qui ont un indice à 0. S'il n'en existe pas il suffit de prendre le point d'entrée de la boucle. Chaque évaluation entraîne une mise à jour de certains indices.

Remarque 3.7 (Modifications dynamiques) *Nous sommes dans une optique statique (compilation) et il est difficile d'intégrer du dynamisme. Un des changements dynamiques que nous voudrions faire, c'est un changement du noeud d'entrée (car nous avons remarqué que ce choix n'était pas le meilleur). Cela peut poser beaucoup de problèmes. Nous allons en voir trois sur des exemples.*

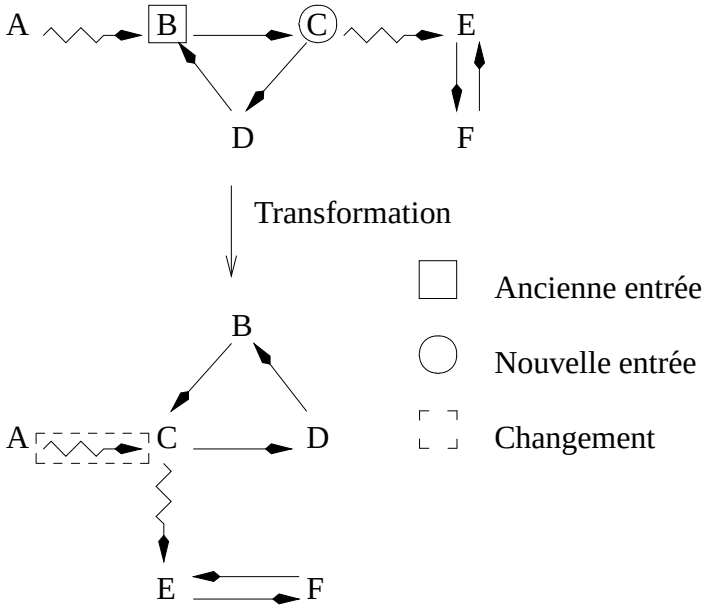

FIG. 3.6 – *Premier exemple de changement dynamique*

Le premier exemple (figure 3.6) nous montre ce qui se passe quand le nouveau noeud d'entrée est un

noeud du même niveau (pas de passage par un arc zoom) qui est une CFC. La modification est simple, il suffit de changer l'arc zoom.

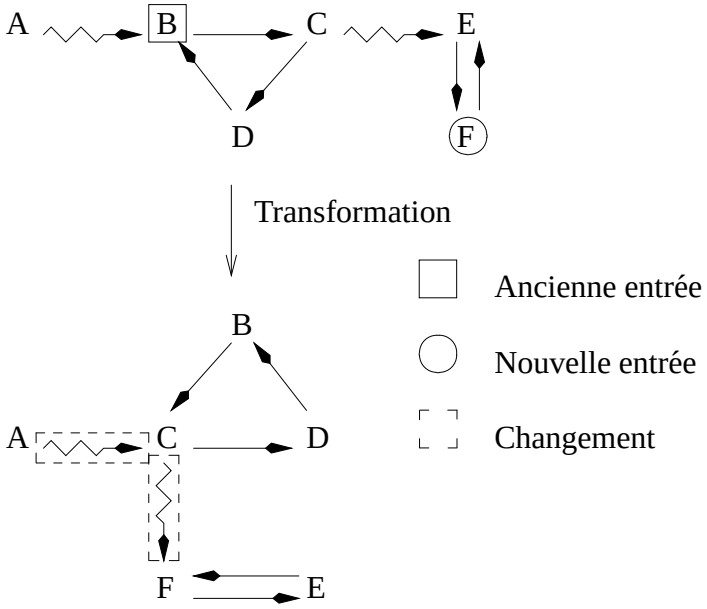

FIG. 3.7 – *Deuxième exemple de changement dynamique*

Dans le deuxième exemple (figure 3.7), le nouveau noeud appartient au niveau suivant. Il faut mettre à jour l'arc zoom en tenant compte de la CFC contenant le nouveau noeud et mettre à jour le point d'entrée de la CFC. Si le saut ne se fait plus sur un niveau mais sur plusieurs, la modification est plus compliquée.

Le troisième exemple (figure 3.8) nous montre bien que le découpage fait pour une CFC dépend fortement du point d'entrée qui avait été choisi. Une modification de ce noeud peut entraîner un redécoupage de la CFC, ce qui n'est pas forcément facile à faire.

Une autre option serait de reconstruire la CFC durant l'évaluation, mais cette solution est très coûteuse. Il vaut mieux éviter de faire entrer la construction dans la partie itérative.

Nous avons décrit deux méthodes qui utilisent la recherche itérative du plus petit point par approximations successives. Elles ne sont pourtant pas identiques, l'approche avec le *Work-Set* est une approche dynamique. Elle recherche dynamiquement les prochaines variables à évaluer. Cela permet de tenir compte des informations que l'on peut collecter au cours de la résolution mais elle entraîne des calculs redondants dans le cas où les informations ne nous apportent rien. L'approche avec les CFC est, au contraire, une approche statique. Tous les choix sont effectués avant le début de l'évaluation, la résolution proprement dite devient détermiste. Mais cette approche ne permet pas de tenir compte des informations que l'on récolte durant l'itération.

Une troisième approche pourait être proposée, elle devra allier une approche statique par compilation d'un graphe d'évaluation tout en permettant de faire facilement des modifications dynamiques. Le graphe pourra ressembler au graphe des composantes fortement connexes, mais il devra être d'une structure plus souple pour pouvoir supporter des modifications dynamiques sans engendrer un coût élevé. Le graphe pourrait donner les grands axes de la résolution et une technique dynamique permettrait de la raffiner. Nous

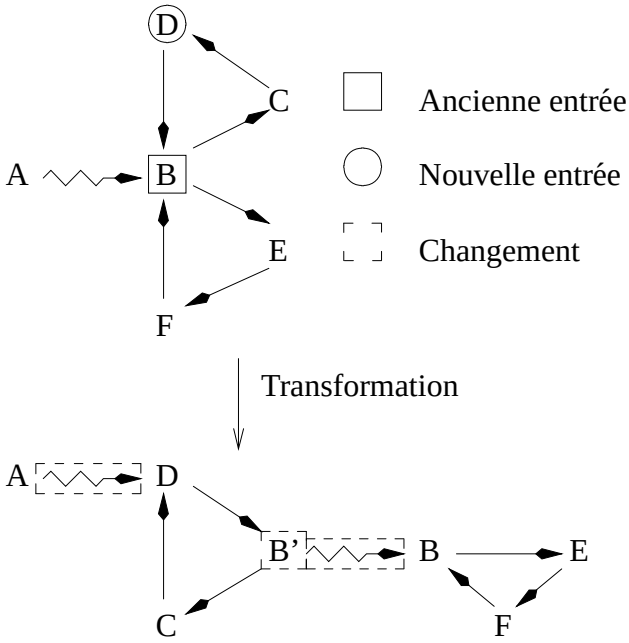

FIG. 3.8 – *Troisième exemple de changement dynamique*

pourrions, par exemple, avoir le graphe en CFC de premier niveau (pas de construction récursive) et faire une gestion dynamique des boucles.

Conclusion

Nous avons présenté deux algorithmes de résolution. Le premier est un algorithme itératif basé sur le *Work-Set* de Gary Kildall [Kildall 73]. Le second est un algorithme récursif qui travaille sur les composantes fortement connexes du graphe de dépendance. La différence fondamentale entre ces deux algorithmes, c'est leur manière d'aborder le problème. Nous avons vu une version dynamique avec l'algorithme à la Kildall et une version statique avec les composantes fortement connexes. Les deux sont bonnes, mais elles ne peuvent pas sortir de leur vision des choses. Il serait intéressant de trouver un algorithme capable de faire du statique et du dynamique ce qui permetrait d'optimiser au mieux l'ordonnancement des évaluations.

D'autre part, la résolution est faite par un seul algorithme, car mis à part la partie préliminaire, nous utilisons la même méthode pour toute la résolution. Nous pourrions avoir une approche moins exclusive, nous pourrions combiner les méthodes de résolution. Commencer par une méthode symbolique, par exemple faire un travail sur les constantes, puis faire tourner un des algorithmes sur un certain nombre d'itérations et regarder les résultats que l'on obtient et s'en servir pour une résolution symbolique et ainsi de suite jusqu'à obtention de la solution. Cette méthode pourrait peut-être permettre de mieux gérer les problèmes d'existence de solution car durant une analyse sur les constantes il est possible de découvrir une inconsistance sur le système (une même variable doit être égale à deux constantes différentes).

Il reste du travail à faire sur ce moteur de résolution, il faudrait étendre les relations acceptées ($=$, $\leq$, $\geq$, $\neq$, $<$, $>$, etc.). Cela reposerait le problème d'existence des solutions ainsi que leur accessibilité dans le cadre d'une recherche du plus petit point fixe du système. Nous avons déjà été confrontés à ce problème, le bon fonctionnement de ces algorithmes repose sur le fait que les systèmes de départ ont les bonnes propriétés en ce qui concerne l'existence et l'accessibilité des solutions. Or nous avons vu qu'il était difficile de garantir ces propriétés. Nous avons montré que si les opérateurs étaient contractants alors nous pouvions garantir ces propriétés, mais qu'il n'était pas possible de savoir statiquement si les opérateurs respectaient cette propriété de contractance. Il faut tout de même se souvenir d'une chose, même si nous ne pouvons garantir les propriétés de façon statique, cela n'entraîne pas forcément des problèmes. Dans de nombreux cas, la résolution du système ne pose pas de problème car étant donné la propriété à vérifier, nous savons qu'il existe forcément une solution, quelle que soit la forme du système.

Bibliographie

[Abramsky et Hankin 87] SAMSON ABRAMSKY et CHRIS HANKIN, *Abstract Interpretation of Declarative Languages*, Ellis Horwood series in computer and their applications, 1987.

[Aiken et al. 94] ALEXANDER S. AIKEN, EDWARD L. WIMMERS et T. K. LAKSHMAN, « Soft Typing with Conditional Types », *Symp. on Principles of Programming Languages (POPL)*, p. 163–173, ACM Press, janvier 1994.

[Aiken et al. 95] ALEXANDER S. AIKEN, MANUEL FÄHNDRICH et RAPH LEVIEN, « Better Static Memory Management: Improving Region-Based Analysis of Higher-Order Languages (Extended Abstract) », *Conf. on Programming Language Design and Implementation (PLDI)*, p. 174–185, juin 1995.

[Aiken et Wimmers 93] ALEXANDER S. AIKEN et EDWARD L. WIMMERS, « Type Inclusion Constraints and Type Inference », *Conf. on Functional Programming and Computer Architecture (FPCA)*, p. 31–41, ACM Press, juin 1993.

[Apt et Bol 94] KRZYSZTOF R. APT et ROLAND N. BOL, « Logic Programming and Negation: A Survey », *Journal of Logic Programming*, vol. 19-20, mai 1994, p. 9–71.

[Bancilhon et al. 86] FRANÇOIS BANCILHON, DAVID MAIER, YEHOSHUA SAGIV et JEFFREY D. ULLMAN, « Magic sets and other strange ways to implement logic programs (extended abstract) », *5th Principles of Database Systems Symp. (PODS)*, p. 1–15, ACM Press, mars 1986.

[Borning et Ingalls 82] ALAN HAMILTON BORNING et DANIEL H. H. INGALLS, « A Type Declaration and Inference System for Smalltalk », *9th Symp. on Principles of Programming Languages (POPL)*, p. 133–141, ACM Press, janvier 1982.

[Bourdoncle 93] FRANÇOIS BOURDONCLE, « Efficient Chaotic Iteration Strategies with Widenings », *Lecture Notes in Computer Science (LNCS)*, vol. 735, juin/juillet 1993, p. 128–141.

[Cardelli 84] LUCA CARDELLI, « A Semantics of Multiple Inheritance », p. 51–66, *Int. Symp. on Semantics of Data Types*, D. B. MacQueen G. Kahn et G. Plotkin ed., Springer-Verlag, juin 1984, Une version revisitée de cet article est paru dans *Information and Computation* [Cardelli 88].

[Cardelli 88] LUCA CARDELLI, « A Semantics of Multiple Inheritance », p. 138–164, *Information and Computation*, Academic Press, février/mars 1988.

[Chen et al. 95] WEIDONG CHEN, TERRANCE SWIFT et DAVID SCOTT WARREN, « Efficient Top-Down Computation of Queries under the Well-Founded Semantics », *Journal of Logic Programming*, vol. 24, n° 3, septembre 1995, p. 161–199.

[Codish et Demoen 93] MICHAEL CODISH et BART DEMOEN, « Analysing Logic Programs using `Prop`-ositional Logic Programs and a Magic Wand », *10th Int. Conf. on Logic Programming (ICLP)*, D. Miller ed., p. 114–129, MIT Press, 1993.

[Conway et al. 95] THOMAS CHARLES CONWAY, FERGUS HENDERSON et ZOLTAN SOMOGYI, « Code Generation for Mercury », *Int. Conf. on Logic Programming (ICLP)*, John Lloyd ed., p. 242–256, MIT Press, décembre 1995.

[Conway et al. 96] THOMAS CHARLES CONWAY, FERGUS HENDERSON et ZOLTAN SOMOGYI, « The Execution Algorithm of Mercury, an Efficient Purely Declarative Logic Programming Language », *Journal of Logic Programming*, vol. 29, n° 1-3, octobre/novembre 1996, p. 17–64.

[Cousot et Cousot 77] PATRICK COUSOT et RADHIA COUSOT, « Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction of Approximation of Fixed Points », *4th Symp. on Principles of Programming Languages (POPL)*, p. 238–252, ACM Press, 1977.

[Cousot et Cousot 92a] PATRICK COUSOT et RADHIA COUSOT, « Abstract Interpretation and applications to logic programs », *Journal of Logic Programming*, vol. 13, n° 2-3, juillet 1992, p. 103–179.

[Cousot et Cousot 92b] PATRICK COUSOT et RADHIA COUSOT, « Invited Lecture: Comparing the Galois Connection and Widening/Narrowing Approaches to Abstract Interpretation », *Lecture Notes in Computer Science (LNCS)*, vol. 631, 1992, p. 269–295.

[Cousot et Cousot 95] PATRICK COUSOT et RADHIA COUSOT, « Formal Language, Grammar and Set-Constraint-Based Program Analysis by Abstract Interpretation », *7th Conf. on Functional Programming and Computer Architecture (FPCA)*, p. 170–181, ACM Press, juin 1995.

[Debray et Ramakrishnan 94] SAUMYA K. DEBRAY et RAGHU RAMAKRISHNAN, « Abstract Interpretation of Logic Programs using Magic Transformations », *Journal of Logic Programming*, vol. 18, n° 2, février 1994, p. 149–176.

[Debray et Warren 88] SAUMYA K. DEBRAY et DAVID SCOTT WARREN, « Automatic Mode Inference for Logic Programs », *Journal of Logic Programming*, vol. 5, n° 3, septembre 1988, p. 207–229.

[Gouranton 97] VALÉRIE GOURANTON, *Dérivation d'analyseurs dynamiques et statiques à partir de spécifications opérationnelles*, thèse, Université de Rennes 1, septembre 1997, N. d'ordre : 1845.

[Heintze 92] NEVIN HEINTZE, « Practical Aspects of Set Based Analysis », *Int. Joint Conf. and Symp. on Logic Programming (IJCSLP)*, Krzysztof Apt ed., p. 765–779, MIT Press, 1992.

[Hornof 97] LUKE HORNOF, *Analyses statiques pour le spécialisation effective de programmes réalistes*, thèse, Université de Rennes 1, juin 1997, N. d'ordre : 1810.

[Jensen 95] THOMAS P. JENSEN, « Conjunctive Type Systems and Abstract Interpretation of Higher-Order Functional Programs », *Journal of Logic and Computation*, vol. 5, n° 4, 1995, p. 397–421.

[Kaufmann et Pichat 77] A. KAUFMANN et E. PICHAT, *Méthodes mathématiques non numériques et leurs algorithmes*, Masson, 1977, Tome 1 : Algorithmes de recherche des éléments maximaux.

[Kildall 73] GARY A. KILDALL, « A Unified Approach to Global Program Optimization », *Symp. on Principles of Programming Languages (POPL)*, p. 194–206, ACM Press, octobre 1973.

[Kuo et Mishra 89] TSUNG-MIN KUO et PRATEEK MISHRA, « Strictness Analysis: A New Perspective Based on Type Inference », *Conf. on Functional Programming and Computer Architecture (FPCA)*, p. 260–272, ACM Press, septembre 1989.

[Le Charlier et al. 93] BAUDOUIN LE CHARLIER, OLIVER DEGIMBE, LAURENT MICHEL et PASCAL VAN HENTENRYCK, « Optimization techniques for general purpose fixpoint algorithms: practical efficiency for the abstract interpretation of Prolog », *Lecture Notes in Computer Science (LNCS)*, vol. 724, septembre 1993, p. 15–26.

[Le Charlier et al. 94] BAUDOUIN LE CHARLIER, SABINA ROSSI et PASCAL VAN HENTENRYCK, « An Abstract Interpretation Framework which Accurately Handles Prolog Search-Rule and the Cut », *Int. Conf. on Logic Programming (ICLP)*, Maurice Bruynooghe ed., p. 157–171, MIT Press, 1994.

[Le Charlier et Musumbu 92] BAUDOUIN LE CHARLIER et KANINDA MUSUMBU, « Une Sémantique Opérationnelle Instrumentale pour Prolog et son application à la Preuve de Consistance d'un modèle d'Interprétation Abstraite », *French Conf. on Logic Programming (JFPL)*, J.-P. Delahaye, P. Devienne, P. Mathieu et P. Yim ed., p. 386–400, 1992.

[Malésieux et al. 98] FRÉDÉRIC MALÉSIEUX, OLIVIER RIDOUX et PATRICE BOIZUMAULT, « Compilation abstraite de λ-Prolog », *Journées Francophones de Programmation Logique et Programmation par Contraintes (JFPLC)*, Hermes ed., p. 287–302, mai 1998.

[Mellish 85] CHRISTOPHER S. MELLISH, « Some Global Optimisations for a Prolog Compiler », *Journal of Logic Programming*, vol. 2, n° 1, 1985, p. 43–66.

[Milner 78] ROBIN MILNER, « A Theory of Type Polymorphism in Programming Languages », *Journal of Computer and System Sciences*, vol. 17, n° 3, 1978, p. 348–375.

[Mishra 88] PRATEEK MISHRA, « Strictness Analysis of the Untyped λ-Calculus », *Information Processing Letters*, vol. 28, n° 3, juillet 1988, p. 121–125.

[Mycroft 81] ALAN MYCROFT, *Abstract Interpretation and Optimizing Transformations for Applicative Programs*, thèse, University of Edinburgh, décembre 1981.

[O'Keefe 87] RICHARD A. O'KEEFE, « Finite Fixed-Point Problems », *4th Int. Conf. on Logic Programming (ICLP)*, J. L. Lassez ed., p. 729–743, 1987.

[Palsberg et O'Keefe 95] JENS PALSBERG et PATRICK M. O'KEEFE, « A Type System Equivalent to Flow Analysis », *22ndSymp. on Principles of Programming Languages (POPL)*, ACM Press, janvier 1995.

[Palsberg et Schwartzbach 91] JENS PALSBERG et MICHAEL I. SCHWARTZBACH, « Object-Oriented Type Inference », *6th Conf. on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, p. 146–161, ACM Press, octobre 1991.

[Palsberg et Schwartzbach 95] JENS PALSBERG et MICHAEL I. SCHWARTZBACH, « Safety Analysis versus Type Inference », *Information and Computation*, vol. 118, n° 1, avril 1995, p. 128–141.

[Rehof et Mogensen 96] JACOB REHOF et TORBEN ÆGIDIUS MOGENSEN, « Tractable Constraints in Finite Semilattices », *3rdStatic Analysis Symp. (SAS)*, R. Cousot et D. A. Schmidt ed., p. 285–301, Springer-Verlag, septembre 1996.

[Rehof et Mogensen 98] JACOB REHOF et TORBEN ÆGIDIUS MOGENSEN, « Tractable Constraints in Finite Semilattices », *Science of Computer Programming (SCP)*, 1998, Une version courte a été présentée à SAS'96 [Rehof et Mogensen 96].

[Ridoux 98] OLIVIER RIDOUX, *λ-Prolog de A à Z, ou presque*, Habilitation à diriger des recherches, Université de Rennes 1, avril 1998.

[Ross et Wright 88] KENNETH A. ROSS et CHARLES R. B. WRIGHT, *Discrete Mathematics*, Prentice-Hall, 1988, 2ème édition.

[Tarjan 83] ROBERT ENDRE TARJAN, « An improved algorithm for hierarchical clustering using strong components », *Information Processing Letters*, vol. 17, n° 1, juillet 1983, p. 37–41.

[Van Hentenryck et al. 95] PASCAL VAN HENTENRYCK, AGOSTINO CORTESI et BAUDOUIN LE CHARLIER, « Evaluation of the Domain *Prop* », *Journal of Logic Programming*, vol. 23, n° 3, juin 1995, p. 237–278.

[Warren 77] DAVID H. D. WARREN, *Implementing Prolog — Compiling Logic Programs*, D.A.I. Research Report n° 39, 40, University of Edinburgh, 1977.

[Warren 92] DAVID SCOTT WARREN, « Memoing for logic programs with application to abstract interpretation and partial deduction », *CACMSpecial Section on Logic Programming*, vol. 35, n° 3, mars 1992, p. 93–111.