VERBATIM

Projet exploratoire RNRT 2005-2006

Sous projet SP4
Analyses statiques pour la validation de codes

Livrable Lot1, Version 1.0

Constitution d'un état de l'art

Auteurs: B.d'Ausbourg et G.Durrieu
ONERA/DTIM - Consortium Verbatim
mailto:ausbourg@cert.fr,durrieu@cert.fr
http://www.onera.fr

Réseau National de Recherche en Télécommunications - Agence Nationale de la Recherche

Table des matières

Chapitre 1

Introduction

Ce rapport constitue le premier livrable du sous-projet SP4 (Analyses statiques pour la validation de codes) du projet RNRT VERBATIM.

Il cherche à constituer un reflet à jour des techniques utilisées et utilisables dès lors qu'il s'agit de réaliser un ensemble d'analyses statiques sur des programmes.

L'objectif affiché dans le projet Verbatim est de permettre la vérification de systèmes d'interaction multimodaux en ne se fondant que sur la disponibilité de leur code. Le principe retenu afin de réaliser cette vérification est le suivant : on cherchera à abstraire du code une représentation formelle du comportement du système. Le comportement observé est dicté par la propriété à vérifier. Le procédé de construction de la représentation abstraite et formelle fait appel à un ensemble de techniques d'analyse statique du code. L'application de ces techniques peut permettre de mieux appréhender sa structure et ses propriétés. Ces dernières peuvent être alors utilisées afin de réaliser des transformations sur ce code sans modifier la sémantique qui lui est attachée.

Ce rapport présente, dans un premier chapitre, un certain nombre de techniques qui ont pu être développées et qui concernent l'analyse statique des programmes. On pourra trouver dans des ouvrages généraux comme [ASU86] [WM94] [Wol96] [Muc97] ou [NNH99] des éléments plus complets sur la nature et l'utilisation de ces techniques pour la compilation des programmes. Les différentes sections de ce chapitre proposent d'autres références bibliographiques plus spoécifiques de chacune des techniques examinées.

Un second chapitre fait également le point sur les outils aujourd'hui existants et mettant en œuvre certaines des techniques décrites au premier chapitre.

Chapitre 2

Analyses statiques

2.1 Interprétation abstraite

Le terme d'interprétation abstraite [Cou00a] désigne une théorie de l'approximation sémantique des systèmes informatiques, utile lorsqu'il s'agit de *vérifier statiquement* des logiciels.

La production de logiciels fiables est en effet devenue une préoccupation fondamentale des concepteurs de système. Les méthodes manuelles classiques (revue de code, simulations, tests) s'avèrent insuffisantes avec l'accroissement de la taille et de la complexité des codes examinés.

L'idée de base de l'analyse statique est d'utiliser l'ordinateur pour découvrir les erreurs éventuelles dont un système peut être entaché. Il s'agit de déterminer *statiquement* (à l'analyse) certaines propriétés *dynamiques* (à l'exécution) des programmes.

Selon la nature des programmes et des propriétés considérés, on parle de *preuve* (les propriétés étant données par l'utilisateur et vérifiées par un assistant de preuve), de *typage* (permettant de vérifier automatiquement la cohérence des structures de données et de certaines fonctionnalités du programme), d'*analyse de flot de données* (utilisées à la compilation pour déterminer si certaines optimisations sont applicables), de *vérification exhaustive de modèles* ou « *model checking* » (un modèle fini de l'exécution du programme étant exploré systématiquement, ou du moins en partie si les ressources sont insuffisantes), etc. Toutes ces analyses statiques se formalisent en fin de compte dans la théorie de l'interprétation abstraite, ainsi que le montrent un certain nombre de publications :

- pour la preuve, voir : [Cou02] et [CC92b] ;
- pour le typage, voir : [Cou97] ;
- pour la vérification exhaustive de modèles, voir : [CC99], [Cou00b], [CC00] et [CC02] ;
- pour l'analyse de flots de données, voir : [CC77] et [CC00] ;
- pour l'analyse statique de programmes, voir : [CC76] [CC77].

Les problèmes abordés sont en général très complexes ; pour les traiter il faut le plus

souvent se résoudre à des compromis consistant à ne considérer qu'une approximation des comportements possibles, ou *sémantique*, du système analysé, d'où le terme d'interprétation abstraite. Cette sémantique décrit par exemple les traces d'exécution du programme (lorsqu'on s'intéresse aux propriétés de vivacité ou « *liveness* ») ou l'ensemble des états accessibles en cours d'exécution (lorsqu'on s'intéresse aux propriétés de sûreté ou « *safety* »). Elle n'est, en général, pas calculable, pour des raisons d'indécidabilité et de complexité : si on peut mathématiquement définir la sémantique d'un programme par une équation de point fixe, celle-ci ne peut être résolue en un temps fini, notamment en raison de la prise en compte de comportements éventuellement infinis du programme, d'où la nécessité d'approximations, rendant la sémantique calculable par ordinateur.

Par ailleurs, une conséquence inévitable de la complexité des problèmes posés est que pour tout analyseur statique, il existe des programmes et des propriétés qu'il n'est pas possible de vérifier en un temps fini avec des ressources informatiques finies. Les échecs se traduisent par le fait que:

- l'analyse ne termine pas ;
- l'analyse termine sans conclure par manque de ressources (par exemple manque de mémoire) ;
- l'analyse termine avec des résultats imprécis, ne permettant de conclure ni positivement ni négativement (cas d'incertitude également appelé « fausse alarme »).

En effet, l'abstraction conduisant à une interprétation abstraite de la sémantique d'un système ne fait que formaliser une certaine perte d'information, ayant pour conséquence que l'analyse statique ne peut répondre à toutes les questions. Plus précisément, toutes les réponses données par la sémantique abstraite sont justes du point de vue de la sémantique concrète ; cependant, certaines questions concrètes peuvent ne pas trouver de réponse dans la sémantique abstraite, et ce d'autant plus que le niveau d'abstraction est élevé.

Considérons par exemple les niveaux d'abstraction sémantique qui suivent [Cou00a] :

- une « sémantique des traces » T, qui est la sémantique de transitions t constituée par l'ensemble des paires d'états $\langle\sigma_i,\sigma_{i+1}\rangle$ figurant dans au moins une trace de la sémantique des traces : $t = \alpha_0(T) = \{\langle\sigma_i,\sigma_{i+1}\rangle : \sigma_0 \ldots \sigma_i\sigma_{i+1} \ldots \in T\}$.
- une « sémantique dénotationnelle » consistant à abstraire une trace finie par la paire constituée de son état initial et de son état final : $\alpha_d(\sigma_0\sigma_1 \ldots) = \langle\sigma_0,\sigma_{n-1}\rangle$, tandis que les traces infinies sont abstraites par l'état initial, la non terminaison étant notée $\perp$: $\alpha_d(\sigma_0\sigma_1 \ldots) = \langle\sigma_0, \perp\rangle$.
- une « sémantique naturelle » ignorant les comportements infinis : $\alpha_n(X) = \{\langle s,s'\rangle \in X \mid s' \neq \perp\}$.

On peut remarquer que l'abstraction α est croissante ($T \subseteq T' \Rightarrow \alpha(T) \subseteq \alpha(T')$), de même que la concrétisation.

Les trois sémantiques permettent de répondre à la question : « l'exécution d'un programme partant de l'état x peut-elle se terminer dans l'état y? » . Mais seules la sémantique

des traces et la sémantique dénotationnelle permettent de répondre à la question : « l'exécution d'un programme partant de l'état x se termine-t-elle toujours ? », la seule réponse valide en ce qui concerne la sémantique naturelle étant : « on ne sait pas » puisque les comportements infinis sont ignorés. Enfin, la sémantique des traces seule permet de répondre à la question : « l'état x peut-il être suivi de l'état y pendant l'exécution du programme ? », la sémantique dénotationnelle et la sémantique naturelle ne permettant pas d'y répondre directement puisque les états intermédiaires du calcul sont oubliés.

2.2 Un langage jouet

De manière à illustrer les différentes techniques d'analyses mises en œuvre ou proposées, on se dote d'un petit langage de programmation impératif, simple, mais proposant des constructions syntaxiques susceptibles de démontrer l'intérêt et l'apport des analyses statiques pour l'examen de programmes formulés dans ce langage.

2.2.1 Expressions

Les expressions de base dénotent toutes des valeurs entières

$$
\begin{array}{rcl}
E & \rightarrow & const\_int \\
 & \rightarrow & id \\
 & \rightarrow & E + E \mid E - E \mid E * E \mid E/E \mid E > E \mid E == E \\
 & \rightarrow & (\ E\ ) \\
 & \rightarrow & \texttt{lire}
\end{array}
$$

Une expression *lire* permet de lire une valeur entière dans le flot des entrées du programme. Les opérateurs de comparaison renvoient les valeurs entières 0 ou 1 selon que le résultat booléen de la comparaison est $faux$ ou $vrai$.

2.2.2 Instructions

Les instructions définies par le langage sont simples et tout à fait familières.

$$
\begin{array}{rcl}
I & \rightarrow & id\ =\ E\ ; \\
 & \rightarrow & \texttt{ecrire}\ E\ ; \\
 & \rightarrow & I\ I \\
 & \rightarrow & \texttt{if}\ (E)\ \{\ I\ \} \\
 & \rightarrow & \texttt{if}\ (E)\ \{\ I\ \}\ else\ \{\ I\ \} \\
 & \rightarrow & \texttt{while}\ (E)\ \{\ I\ \} \\
 & \rightarrow & \texttt{var}\ id_1, \ldots, id_n\ ;
\end{array}
$$

Concernant l'évaluation qui est faite des expressions arithmétiques dans les conditionnelles, une valeur 0 est interprétée comme une évaluation à la valeur booléenne $faux$ de la conditionnelle, toute autre valeur étant interprétée comme une évaluation à la valeur booléenne $vrai$ de cette conditionnelle. L'instruction `ecrire` permet d'écrire une valeur entière résultant de l'évaluation d'une expressions E sur un flot de sortie. L'instruction `var` est une instruction de déclarations de variables non intialisées.

2.2.3 Fonctions

Les fonctions associent une valeur de retour à un ensemble d'arguments :

$$F \rightarrow id\ (\ id,\ldots,id\ )\ \{\ \texttt{var}\ id,\ldots,id\ ;\ S\ \texttt{return}\ E;\ \}$$

Un appel de fonction constitue alors une forme d'expression particulière :

$$E \rightarrow id\ (\ E,\ldots,E\ )$$

2.2.4 Pointeurs

Concernant l'usage des pointeurs, le langage définit les expressions sur pointeurs E_p qui suivent :

$$\begin{aligned} E_p &\rightarrow \&id \\ &\rightarrow \texttt{malloc} \\ &\rightarrow *E \\ &\rightarrow \texttt{null} \end{aligned}$$

L'expression $\&id$ crée en fait un pointeur ou une référence vers une variable d'identifieur id. L'expression `malloc` permet d'allouer dynamiquement de la mémoire sur le tas. L'expression $*E$ permet de déréférencer une référence ou un pointeur. De manière à affecter des valeurs aux cellules élémentaires qui peuvent être allouées on se dote d'une nouvelle forme d'instruction :

$$S \rightarrow *id\ =\ E\ ;$$

Il faut noter qu'adresses et entiers constituent des domaines de valeurs distincts. Le langage ne permet pas de construire une arithmétique sur les pointeurs.

Le langage autorise également l'usage des pointeurs de fonctions. Pour cela, l'invocation d'une fonction revêt la forme d'une expression :

$$E \rightarrow (E)\ (\ E,\ldots,E\ )$$

2.2.5 Programme

Un programme est alors une collection de fonctions :

$$P \rightarrow F \ldots F$$

La fonction finale est celle qui lance l'exécution. Ses arguments sont fournis en séquence dans le flot d'entrée du programme, et la valeur retournée est concaténée au flot de sortie.

2.2.6 Exemples de programmes

Le programme suivant calcule la factorielle d'un entier qui lui est passé en paramètre.

```
fact(n) {
     var f;
     f = 1;
     while (n>0) {
         f = f*n;
         n = n-1;
     }
     return f;
}
```

Ce nouveau programme fait la même chose, mais est récursif :

```
factrec(n) {
     var f;
     if (n==0) { f = 1; }
     else { f = n*factrec(n-1);}
     return f;
}
```

Ce dernier programme effectue le même calcul mais avec une structure compliquée :

```
ff(p,x) {
     var f,q;
     if (*p==0) { f=1; }
     else {
          q = malloc;
          *q = (*p)-1;
          f = (*p)*((x)(q,x));
     }
     return f;
}
```

```
main() {
     var n;
     n = lire ;
     return ff(&n,ff);
}
```

2.3 Analyse de types

Le langage de programmation défini en section 2.2 n'est pas typé. Néanmoins, les diverses opérations qui sont définies par la structures des expressions ne s'appliquent qu'à certains arguments. On peut raisonnablement définir les restrictions suivantes :

- les opérations arithmétiques et de comparaisons ne s'appliquent que sur les entiers ;
- les opérations de lecture et d'écriture ne s'appliquent qu'aux entiers ;
- les expressions conditionnelles, dans les structures de contrôle doivent être entières ;
- seules les fonctions peuvent être invoquées ;
- l'opérateur unaire $*$ ne s'applique qu'aux pointeurs.

On cherche à savoir si ces exigences sont maintenues durant l'exécution. Mais ce problème n'est pas, en soi, décidable. C'est la raison pour laquelle on peut dès lors se rabattre sur une approximation conservative : le caractère typable d'un programme. Un programme est "typable" s'il satisfait un ensemble de contraintes de types que l'on dérive systématiquement de l'arbre syntaxique du programme. Cette condition implique que les exigences énoncées ci-dessus peuvent être garanties satisfaites par l'exécution du programme. La réciproque n'est pas vraie : l'analyse des types peut ainsi rejeter un programme qui ne violerait pas ces exigences à l'exécution.

Concernant l'analyse de types, on peut se référer à [Mil78] [ASU86] [WM94] [Muc97] [NNH99], et pour ce qui est des langages à objets [Cas97] [Age94] [Age95] [Bac97] [CCZ97] [DMM96] [OPS92] [PS91] [PS94] et [Pal01].

2.3.1 Types

On définit un langage de types qui décrit les valeurs possibles que peuvent prendre les types :

$$\begin{aligned} \tau &\rightarrow int \\ &\rightarrow \&\tau \\ &\rightarrow (\tau,\ldots,\tau) \rightarrow \tau \end{aligned}$$

Les termes présentés ci-dessus décrivent respectivement des entiers, des pointeurs, et des pointeurs sur fonctions.

2.3.2 Contraintes de types

Pour un programme donné, il est possible de générer un système de contraintes et de décider si le programme est typable ou ne l'est pas en vérifiant si l'ensemble des contraintes peuvent être satisfaites ou ne le peuvent pas.

Pour chaque identifieur id on introduit la variable de type $[\![id]\!]$, et pour chaque expression E une variable de type $[\![E]\!]$. Pour chaque construction du langage on définit les contraintes qui suivent :

$$
\begin{array}{rcl}
const\_int & : & [\![const\_int]\!] = \texttt{int} \\
E_1 \texttt{ op } E_2 & : & [\![E_1]\!] = [\![E_2]\!] = [\![E_1 \texttt{ op } E_2]\!] = \texttt{int} \\
\texttt{lire} & : & [\![lire]\!] = \texttt{int} \\
id = E & : & [\![id]\!] = [\![E]\!] \\
\texttt{ecrire } E & : & [\![E]\!] = \texttt{int} \\
\texttt{if } (E)\ S & : & [\![E]\!] = \texttt{int} \\
\texttt{if } (E)\ S_1 \texttt{ else } S_2 & : & [\![E]\!] = \texttt{int} \\
\texttt{while } (E)\ S & : & [\![E]\!] = \texttt{int} \\
id(id_1,\ldots,id_n)\{\ \ldots \texttt{ return } E;\ \} & : & [\![id]\!] = ([\![id_1]\!],\ldots,[\![id_n]\!]) \rightarrow [\![E]\!] \\
id(E_1,\ldots,E_n) & : & [\![id]\!] = ([\![E_1]\!],\ldots,[\![E_n]\!]) \rightarrow [\![id(E_1,\ldots,E_n)]\!] \\
(E)(E_1,\ldots,E_n) & : & [\![E]\!] = ([\![E_1]\!],\ldots,[\![E_n]\!]) \rightarrow [\![(E)(E_1,\ldots,E_n)]\!] \\
\&id & : & [\![\&id]\!] = \&[\![id]\!] \\
\texttt{malloc} & : & [\![\texttt{malloc}]\!] = \&\alpha \\
\texttt{null} & : & [\![\texttt{null}]\!] = \&\alpha \\
*E & : & [\![E]\!] = \&[\![*E]\!] \\
*id = E & : & [\![id]\!] = \&[\![E]\!]
\end{array}
$$

Chacune des occurrences de α dénote une variable de type. On peut aussi noter que les déclarations et références de variables ne génèrent aucune contrainte dans ce système de types. Ainsi, un programme donné conduit à produire un ensemble de contraintes sur des variables de types. Une solution affecte à chacune des variables de type un type de telle sorte que les contraintes soient satisfaites. Si une telle solution existe, alors le programme est typable.

2.3.3 Résoudre les contraintes

Lorsque des solutions à ces systèmes de contraintes existent, alors elles peuvent être calculées. On peut illustrer comment procède ce calcul d'unification en reprenant l'exemple compliqué du calcul de factorielle et générant l'ensemble des contraintes induites par les structures syntaxiques reconnues de ce programme :

$$[\![ff]\!] = ([\![p]\!],[\![x]\!]) \to [\![f]\!]$$
$$[\![*p == 0]\!] = \texttt{int}$$
$$[\![*p]\!] = [\![0]\!] = \texttt{int}$$
$$[\![f]\!] = [\![1]\!] = \texttt{int}$$
$$[\![p]\!] = \&[\![*p]\!]$$
$$[\![q]\!] = [\![\texttt{malloc}]\!]$$
$$[\![\texttt{malloc}]\!] = \&\alpha$$
$$[\![*q]\!] = [\![(*p) - 1]\!]$$
$$[\![q]\!] = \&[\![*q]\!]$$
$$[\![*p]\!] = \texttt{int}$$
$$[\![f]\!] = [\![(*p) * ((x)(q,x))]\!]$$
$$[\![(*p) * ((x)(q,x))]\!] = \texttt{int}$$
$$[\![(x)(q,x)]\!] = \texttt{int}$$
$$[\![x]\!] = ([\![q]\!],[\![x]\!]) \to [\![(x)(q,x)]\!]$$
$$[\![\texttt{lire}]\!] = \texttt{int}$$
$$[\![main]\!] = () \to [\![ff(\&n,ff)]\!]$$
$$[\![n]\!] = [\![\texttt{lire}]\!]$$
$$[\![\&n]\!] = \&[\![n]\!]$$
$$[\![ff]\!] = ([\![\&n]\!],[\![ff]\!]) \to [\![ff(\&n,ff)]\!]$$

Ces contraintes admettent une solution. Celle-ci consiste à affecter la valeur `int` à toutes les variables de types à l'exception de :

$$[\![p]\!] = \&\texttt{int}$$
$$[\![q]\!] = \&\texttt{int}$$
$$[\![malloc]\!] = \&\texttt{int}$$
$$[\![x]\!] = \phi$$
$$[\![ff]\!] = \phi$$
$$[\![\&n]\!] = \&\texttt{int}$$
$$[\![main]\!] = () \to \texttt{int}$$

où ϕ est le type $\phi = (\&\texttt{int},\phi) \to \texttt{int}$.

Puisque cette solution existe, on en conclut que le programme est correct du point de vue de son typage. Les types récursifs peuvent également s'avérer nécessaires pour décrire des structures de données. Par exemple, la séquence d'instructions suivantes :

```
var p;
```

```
p = malloc;
*p = p;
```

génère l'ensemble de contraintes :

$$[\![p]\!] = [\![malloc]\!]$$
$$[\![malloc]\!] = \&\alpha$$
$$[\![p]\!] = \&[\![p]\!]$$

qui admet pour solution $[\![p]\!] = \psi$ avec $\psi = \&\psi$.

2.3.4 Limites

L'analyse des types constitue une évaluation approximative des programmes. Certains programmes peuvent ainsi être rejettés abusivement. Par exemple le programme :

```
f(g,x) {
     var r;
     if (x==0) r=g; else r = f(2,0);
     return r+1;
}

main() {
     return f(null,1);
}
```

ne provoquera jamais d'erreur à l'exécution mais n'est cependant pas typable. En effet, il génère entre autres l'ensemble de contraintes suivantes pour lequel il n'existe pas de solution :

$$\texttt{int} = [\![r]\!] = [\![g]\!] = \&\alpha$$

Ceci dénote une incapacité à construire un schéma cohérent d'affectation de types. Néanmoins, l'exécution du programme ne sera pas erronnée. Cela est dû à l'invocation de `f`, dans la fonction `main` avec une valeur de second argument 1. Ceci provoque l'inovcation, dans le corps de la fonction `f`, de `f(2,0)`. Le paramètre formel `g` est ainsi de type `int` et l'affectation `r=g` est dès lors possible. Ceci n'aurait pu être le cas si, dans la fonction `main`, l'invocation avait été par exemple `f(null,0)`.

2.4 Treillis

Beaucoup de techniques utilisées pour réaliser des analyses statiques de programmes se fondent sur des éléments théoriques tirés de la théorie des treillis. On les présente brièvement ici. On pourra consulter [DP90] pour de plus amples développements.

2.4.1 Treillis

Un *ordre partiel* est une structure mathématique $L = (S, \sqsubseteq)$ dans laquelle S est un ensemble et $\sqsubseteq$ une relation sur S qui satisfait les conditions suivantes :

- réflexivité : $\forall x \in S : x \sqsubseteq x$
- transitivité : $\forall x,y,z \in S : x \sqsubseteq y \wedge y \sqsubseteq z \Rightarrow x \sqsubseteq z$
- antisymétrie :$\forall x,y \in S : x \sqsubseteq y \wedge y \sqsubseteq x \Rightarrow x = y$

Soit $X \subseteq S$. On dit que $y \in S$ est un majorant de X, et l'on écrit $X \sqsubseteq y$, si l'on a $\forall x \in X : x \sqsubseteq y$. De même, $y \in S$ est un minorant de X, et l'on écrit $y \sqsubseteq X$, si l'on a $\forall x \in X : y \sqsubseteq x$. Une borne supérieure est un plus petit majorant, noté $\sqcup X$, et défini par

$$X \sqsubseteq \sqcup X \wedge \forall y \in S : X \sqsubseteq y \Rightarrow \sqcup X \sqsubseteq y$$

De même, une borne inférieure est un plus grand majorant que l'on note $\sqcap X$ et que l'on définit par :

$$\sqcap X \sqsubseteq X \wedge \forall y \in S : y \sqsubseteq X \Rightarrow y \sqsubseteq \sqcap X$$

Un *treillis* est alors un ordre partiel dans lequel $\sqcup X$ et $\sqcap X$ existent pour toute partie $X \subseteq S$. Un treillis doit posséder un plus grand élément, unique, noté $\top$ et défini par $\top = \sqcup S$ ainsi qu'un plus petit élément, également unique, noté $\bot$ et défini par $\bot = \sqcap S$.

On s'intéresse, dans la plupart des cas, à des treillis finis. Un ordre partiel sur un ensemble fini constitue un treillis si $\top$ et $\bot$ existent et si toute paire d'éléments x et y possède une borne supérieure,notée $x \sqcup y$ et une borne inférieure notée $x \sqcap y$.

Un treillis peut se représenter graphiquement comme un graphe dans lequel les nœuds représentent les éléments de l'ensemble S et la relation d'ordre est la fermeture transitive des arcs conduisant des nœuds situés plus bas vers les nœuds situés plus haut sur le graphe. La figure 2.1 représente des structures d'ordre partiel qui sont effectivement des treillis.

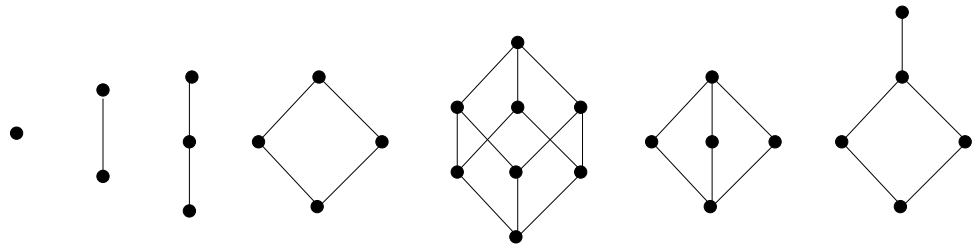

FIG. 2.1 – *Structures de treillis*

La figure 2.2 représente en revanche des structures d'ordres partiels qui ne sont pas des treillis. La structure de gauche ne permet pas de faire apparaître un plus grand élément $\top$. La structure de droite ne permet pas de faire apparaître une borne supérieure pour tout élément de la structure. Par exemple, on ne peut trouver le plus petit majorant $d \sqcup e$.

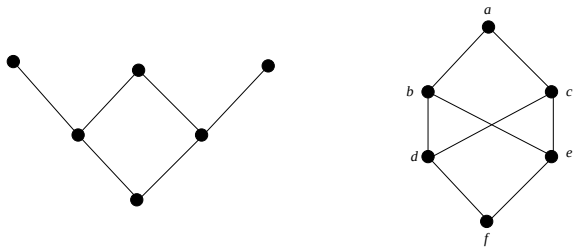

FIG. 2.2 – *Structures d'ordres partiels non treillis*

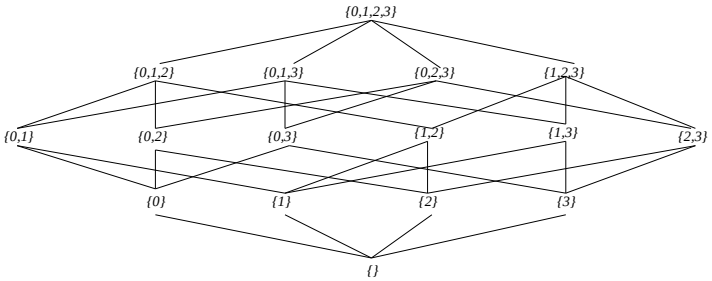

FIG. 2.3 – *Treillis associé à un ensemble* $E = \{0,1,2,3\}$

Il est intéressant de remarquer que tout ensemble E définit un treillis $(2^A, \subseteq)$ où l'on a $\perp = \emptyset$, $\top = E$, $x \sqcup y = x \cup y$ et $x \sqcap y = x \cap y$ $\forall x,y \in 2^A$. Par exemple, pour un ensemble $E = \{0,1,2,3\}$ le treillis défini par E est celui présenté à la figure 2.3.

La *taille* d'un treillis est définie comme la longueur du plus long chemin reliant $\perp$ à $\top$. Pour le treillis de l'ensemble puissance défini à la figure 2.3, la taille est de 4. De manière générale, le treillis $(2^A, \subseteq)$ a une taille $|A|$.

Si $L_1 = (S_1, \sqsubseteq_1), L_2 = (S_2, \sqsubseteq_2), \ldots, L_n = (S_n, \sqsubseteq_n)$ sont des treillis de taille finie, il est possible de définir le produit

$$L_1 \times L_2 \times \ldots \times L_n = (\{(x_1,x_2,\ldots,x_n) x_i \in S_i\}, \sqsubseteq)$$

dont la relation d'ordre partiel $\sqsubseteq$ est définie par

$$\forall X = (x_1,x_2,\ldots,x_n), Y = (y_1,y_2,\ldots,y_n)\ X \sqsubseteq Y \ \Leftrightarrow x_1 \sqsubseteq_1 y1 \wedge x_2 \sqsubseteq_2 y2 \wedge \ldots \wedge x_n \sqsubseteq_n y_n$$

De même, si L est un treillis de taille finie, son extension $ext(L)$ est illustrée par la figure 2.4 et l'on a $taille(ext(L)) = taille(L) + 1$. De même, si A est un ensemble fini, alors $plat(A)$ est un treillis mettant à plat l'ensemble des éléments de A comme le montre la figure 2.5. Finalement, si A et L sont définis comme ci-dessus, on peut définir un treillis de relation de taille fini comme

$$A \mapsto L = \{[a_1 \mapsto x_1, \ldots, a_n \mapsto x_n] \ | x_i \in L\}$$

. Il faut noter que $taille(A \mapsto L) = |A|.taille(L)$.

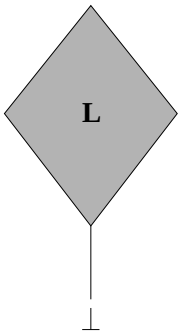

FIG. 2.4 – *Extension d'un treillis L*

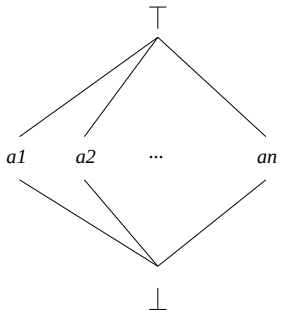

FIG. 2.5 – *Mise à plat d'un ensemble A*

2.4.2 Points fixes

La structure de treillis définie sur un ensemble est une structure intéressante : elle fait apparaitre des propriétés commodes sur les fonctions définies sur l'ensemble puissance. En particulier l'existence de point fixes.

Une fonction $f : L \rightarrow L$ est *monotone* si $\forall x,y \in S : x \sqsubseteq y \Rightarrow f(x) \sqsubseteq f(y)$. Ceci n'indique cependant pas que la fonction f est croissante ; par exemple les fonctions constantes répondent à la définition de cette monotonie. La composition de fonctions monotones est elle-même une fonction monotone.

Le point intéressant est constitué par le théorème du point fixe. Pour un treillis L de taille finie, toute fonction monotone $f : L \rightarrow L$ possède un plus petit point fixe unique défini par

$$fix(f) = \bigsqcup_{i \geq 0} f^i(\bot)$$

pour lequel on a $f(fix(f) = fix(f)$.

En effet, $\bot \sqsubseteq f(\bot)$ puisque $\bot$ est le plus petit élément du treillis. Comme f est monotone, on a donc $f(\bot) \sqsubseteq f^2(\bot)$. Par induction on déduit que $f^i(\bot) \sqsubseteq f^{i+1}(\bot)$. On

peut ainsi construire une chaîne croissante :

$$\bot \sqsubseteq f(\bot) \sqsubseteq f^2(\bot) \sqsubseteq \ldots$$

Puisque L est supposé de taille finie, il existe nécessairement une borne k telle que $f^k(\bot) = f^{k+1}(\bot)$. On définit alors $fix(f) = f^k(\bot)$. En effet $f(fix(f)) = f^{k+1}(\bot) = f^k(\bot) = fix(f)$: $fix(f)$ est donc bien un point fixe. Supposons qu'il existe un autre point fixe x. Par définition $\bot \sqsubseteq x$ et donc $f(\bot) \sqsubseteq f(x) = x$ puisque f est monotone. Par induction on a donc $fix(f) = f^k(\bot) \sqsubseteq x$. $fix(f)$ est donc le plus petit point fixe. L'antisymétrie permet de déterminer qu'il est unique.

Les points fixes permettent de résoudre des systèmes d'équations. Soit L un treillis de taille finie. Un système d'équations est de la forme :

$$\begin{aligned}
x_1 &= F_1(x_1, \ldots, x_n) \\
x_2 &= F_2(x_1, \ldots, x_n) \\
&\cdot \\
&\cdot \\
&\cdot \\
x_n &= F_n(x_1, \ldots, x_n)
\end{aligned}$$

où les x_i sont des variables et $F_i : L^n \to L$ une collection de fonctions monotones. Un tel système admet une plus petite solution unique qui est obtenue comme le plus petit point fixe de la fonction $F : L^n \to L^n$ définie par :

$$F(x_1, \ldots, x_n) = (F_1(x_1, \ldots, x_n), \ldots, F_n(x_1, \ldots, x_n))$$

On peut, de la même manière, trouver une solution à un système d'inéquations de la forme :

$$\begin{aligned}
x_1 &\sqsubseteq F_1(x_1, \ldots, x_n) \\
x_2 &\sqsubseteq F_2(x_1, \ldots, x_n) \\
&\cdot \\
&\cdot \\
&\cdot \\
x_n &\sqsubseteq F_n(x_1, \ldots, x_n)
\end{aligned}$$

en remarquant que la relation $x \sqsubseteq y$ est équivalente à $x = x \sqcap y$. Le systèmes d'inéquations peut alors être réécrit comme :

$$\begin{aligned}
x_1 &= x_1 \sqcap F_1(x_1, \ldots, x_n) \\
x_2 &= x_2 \sqcap F_2(x_1, \ldots, x_n) \\
&\cdot \\
&\cdot \\
&\cdot \\
x_n &= x_n \sqcap F_n(x_1, \ldots, x_n)
\end{aligned}$$

qui est alors un système d'équations de fonctions monotones. On est ainsi ramené au calcul de point fixe précédent.

2.5 Graphes de flot de contrôle

Les analyses sur les types développées en 2.3.1 étaient réalisées indépendamment du flot de contrôle des programmes. Elles s'attachaient exclusivement à la structure des constructions syntaxiques. Dès lors que l'on souhaite prendre en compte la structure du contrôle inhérente à l'exécution possible d'un programme, il convient de se doter d'une représentation du flot de contrôle. On le fait à travers la constitution d'un graphe de flot de contrôle.

Un graphe de flot de contrôle (*Control Flot Graph* ou CFG) est un graphe orienté dont les nœuds correspondent à des points de l'exécution du programme et dont les arcs représentent les transferts possibles du contrôle d'un point (nœud) vers un autre. Un graphe de flot de contrôle dispose toujours d'un point d'entrée et de sortie uniques.

De manière générale si v est un nœud du graphe $pred(v)$ désigne l'ensemble des nœuds qui précèdent v dans le graphe et $succ(v)$ l'ensemble des nœuds qui lui succèdent.

On peut construire un CFG par composition de constructions élémentaires attachées aux constructions de base des indstructions du langage. Par exemple, en se fondant sur le langage défini en 2.2, on peut construire le graphe de contrôle associé aux instructions simples.

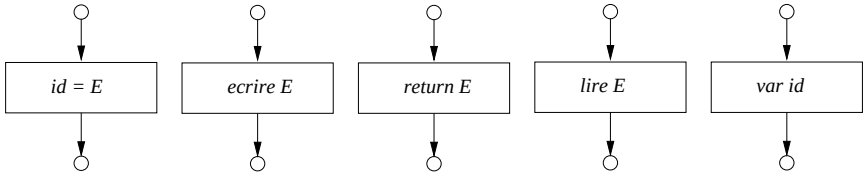

FIG. 2.6 – *CFG d'instructions élémentaires*

La figure 2.6 représente le flot de contrôle des instructions simples d'affectation, d'écriture, de lecture, de retour de fonction ou de déclaration. La séquence de deux instructions simples I_1 et I_2 est construite en superposant le nœud de sortie de I_1 et le nœud d'entrée de I_2 et en élminant ensuite ce nœud. La figure 2.7 illustre cette construction.

De la même manière la figure 2.8 illustre les constructions induites par les structures de contrôle `if` et `while` proposées par le langage.

Il est alors possible de construire, en utilisant cette approche, le graphe de flot de contrôle du programme `fact(n)` défini en 2.2. La figure 2.9 en donne une représentation.

On se reportera à [ASU86] [WM94] [Wol96] [Muc97] ou [NNH99] pour de plus amples renseignements. On pourra aussi consulter sur le sujet par exemple [Wol91] [JC97] [Hav97] et [Ram99].

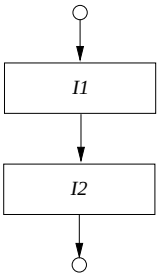

FIG. 2.7 – *CFG d'une séquence d'instructions élémentaires*

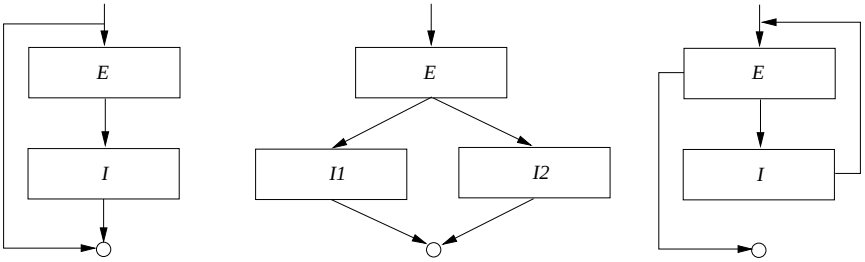

FIG. 2.8 – *CFG des structures de contrôle* `if` *et* `while`

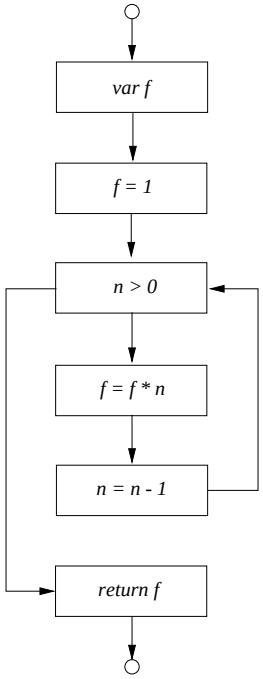

FIG. 2.9 – *CFG du programme* `fact`

2.6 Analyse des flots de données

Il existe une littérature abondante sur l'analyse des flots de données. On pourra bien sûr consulter les ouvrages généraux tels que [ASU86] [WM94] [Wol96] [Muc97] ou [NNH99]. On pourra également se reporter à [HU73] [HU74] [Ken76] [KU76] [Ros77] [Sha80] [RP86] [HDT87] [MR90] [TH92] [Sre95] [Dwy95] ou encore [AC99].

Une analyse de flot de données s'appuie sur un graphe de flot de contrôle et un treillis L de taille finie. Le treillis peut être fixé pour l'ensemble des programmes analysés ou fixé pour un programme particulier, paramétré par ce programme. Il dépend du type d'analyse de flots de données que l'on réalise. A chaque nœud v du graphe on associe une variable $[\![v]\!]$ qui prend ses valeurs dans les éléments de L. Pour chacune des constructions du langage de programmation on peut définir une contrainte qui relie la valeur de cette variable à celles d'autres nœuds (en particulier les nœuds voisins dans le graphe de contrôle).

On peut ainsi extraire d'un graphe de contrôle un ensemble de contraintes sur les variables. Si ces contraintes sont des équations ou des inéquations dont les parties droites peuvent être définie comme des fonctions monotones, alors on peut utiliser un algorithme de recherche de point fixe pour en calculer la plus petite solution.

2.6.1 Algorithmes de calcul de points fixes

Si le graphe de contrôle d'un programme est composé de l'ensemble des nœuds $V = \{v_1,v_2,\ldots,v_n\}$, alors on travaille avec le treillis produit L^n. En supposant que chaque nœud v_i génère l'équation de flot de données $[\![v_i]\!] = F_i([\![v_1]\!],\ldots,[\![v_n]\!])$, on peut construire alors la fonction $F : L^n \to L^n$ comme décrite plus haut :

$$F(x_1,\ldots,x_n) = (F_1(x_1,\ldots,x_n),\ldots,F_n(x_1,\ldots,x_n))$$

Un premier algorithme de calcul comme celui-ci

$x = (\bot,\ldots,\bot)$;
`do` { $t = x$; $x = F(x)$; }
`while` $(x \neq t)$;

permet de calculer le point fixe x. On peut lui trouver un meilleur algorithme qui exploite le fait que le treillis a une structure de produit L^n

$x_1 = \bot$; $\ldots x_n = \bot$;
`do` {
  $t_1 = x_1$; $\ldots t_n = x_n$;
  $x_1 = F_1(x_1,\ldots,x_n)$;
  $\ldots$
  $x_n = F_n(x_1,\ldots,x_n)$;
} `while` $(x_1 \neq t_1 \ \vee \ldots \vee \ x_n \neq t_n)$;

pour calculer le point fixe $(x_1, \ldots, x_n)$.

Ces algorithmes ne sont cependant pas très performants parce qu'ils recalculent, à chaque itération, l'information attachée à chacun des nœuds même si cette dernière n'a pas changé. On peut améliorer ces algorithmes en s'intéressant à la structure des contraintes.

De manière générale, toute variable $[\![v_i]\!]$ dépend de toutes les autres variables. Dans la réalité, le plus souvent, une F_i ne fonde son calcul que sur quelques unes de ces variables. On peut représenter ce fait par une relation :

$$dep : V \rightarrow 2^V$$

qui indique pour chaque nœud v le sous-ensemble des autres nœuds v_i pour lequels $[\![v]\!]$ intervient explicitement dans le calcul de $[\![v_i]\!]$, et par conséquent dans l'évaluation de la partie droite de l'équation de la contrainte attachée à v_i. En d'autres termes $dep(v)$ est l'ensemble des nœuds dont l'information dépend de l'information attachée à v. On peut dès lors proposer un nouvel algorithme

$x_1 = \bot$; $\ldots x_n = \bot$;
$q = [v_1, \ldots, v_n]$;
`while` $(q \neq [])$ {
  supposons $q = [v_i, \ldots]$
  $y = F_i(x_1, \ldots, x_n)$;
  $q = q$.`tail`() ;
  `if` $(y \neq x_i)$ {
    `for` $(v \in dep(v_i))q$.`append`(v) ;
    $x_i = y$;
  }
}

pour le calcul du point fixe $(x_1, \ldots, x_n)$. La file q maintient l'ensemble des nœuds pour lesquels un calcul de l'information qui leur est attachée doit être opéré. L'opérateur `tail` a pour effet de prendre l'ensemble de la file à l'exception de son premier élément qui est retiré.

A l'inverse, l'opérateur `append` ajoute un élément à la file. L'algorithme déclenche le calcul d'une variable $[\![v_i]\!]$ uniquement tant que les variables dont dépend $[\![v_i]\!]$ ne sont pas stables (test $y \neq x_i$ dans l'algorithme).

2.6.2 Application à la vivacité

On dit qu'une variable d'un programme est *vive* en un point de ce programme si sa valeur courante en ce point doit être consultée ou lue par une instruction en un autre point d'exécution du programme à la suite du point courant.

Soit par exemple le programme suivant :

```
var x,y,z;
x = lire ;
while (x>1) {
     y = x/2;
     if (y>3) x = x-y;
     z = x-4;
     if (z>0) x = x/2;
     z = z-1;
}
ecrire x ;
```

On cherche à obtenir des ensembles de variables vives en chaque point du programme. Le treillis que l'on associe à ce programme est alors composé d'éléments qui sont des ensembles des variables du programmes, ordonnés par la relation d'inclusion : $L = (2^{x,y,z}, \subseteq)$.

La figure 2.10 représente le graphe de flot de contrôle du programme. Pour chacun des nœuds v de ce graphe, on introduit une variable $[\![v]\!]$ qui dénote le sous-ensemble des variables du programmes qui sont vives à l'entrée du nœud. L'analyse que l'on réalise est conservative : l'ensemble calculé peut être trop large. Ainsi une variable déclarée vive peut ne pas l'être en fait à l'exécution. En revanche, on peut être sûr qu'une variable qui est calculée ne pas l'être ne le sera pas effectivement lors de l'exécution. On se donne pour cela la définition suivante :

$$COLLECTE(v) = \bigcup_{w \in succ(v)} [\![w]\!]$$

Ainsi, pour les nœud d'entrée et de sortie la contrainte est la suivante :

$$[\![entry]\!] = [\![exit]\!] = \{\}$$

Pour les instructions conditionnelles ou les instructions `ecrire` on se donne la contrainte :

$$[\![v]\!] = COLLECTE(v) \cup vars(E)$$

Pour une instruction d'affectation la contrainte est alors

$$[\![v]\!] = COLLECTE(v) \backslash \{id\} \cup vars(E)$$

Pour une instruction de déclaration

$$[\![v]\!] = COLLECTE(v) \backslash \{id_1, \ldots, id_n\}$$

Finalement les autres nœuds se voient attribuer la contrainte

$$[\![v]\!] = COLLECTE(v)$$

Il convient de préciser que $vars(E)$ dénote l'ensemble des variables intervenant dans une expression E. On peut assez facilement démontrer que ces contraintes constituent définissent des fonctions monotones sur l'ensemble des variables $[\![v_i]\!]$.

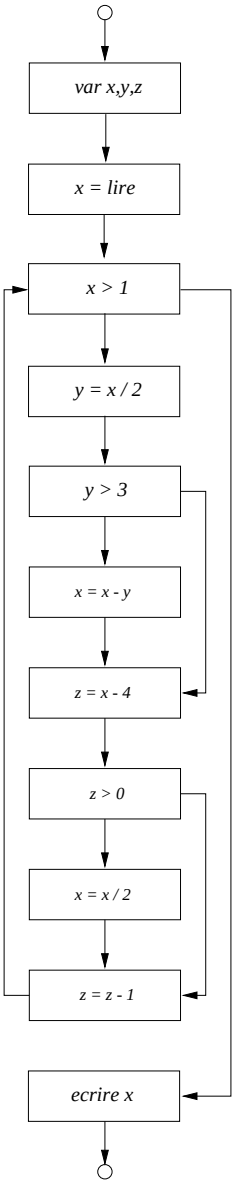

FIG. 2.10 – *CFG du programme*

Intuitivement, une variable est vive à l'entrée d'un nœud si elle doit être lue dans ce nœud ou si elle doit être lue dans un nœud successeur à moins que cette variable ne soit écrite dans le nœud courant.

En appliquant ces contraintes au graphe de flot de contrôle de la figure 2.10 on obtient le système d'équations suivant :

$$[\![entry]\!] = \{\}$$
$$[\![\texttt{var x,y,z}]\!] = [\![\texttt{x = lire}]\!] \backslash \{\texttt{x,y,z}\}$$

$[\![\texttt{x} = \texttt{lire}]\!] = [\![\texttt{x} > 1]\!] \backslash \{\texttt{x}\}$
$[\![\texttt{x} > 1]\!] = ([\![\texttt{y} = \texttt{x}/2]\!] \cup [\![\texttt{ecrire x}]\!]) \cup \{\texttt{x}\}$
$[\![\texttt{y} = \texttt{x}/2]\!] = ([\![\texttt{y} < 3]\!] \backslash \{\texttt{y}\}) \cup \{\texttt{x}\}$
$[\![\texttt{y} < 3]\!] = [\![\texttt{x} = \texttt{x} - \texttt{y}]\!] \cup [\![\texttt{z} = \texttt{x} - 4]\!]$
$[\![\texttt{x} = \texttt{x} - \texttt{y}]\!] = ([\![\texttt{z} = \texttt{x} - 4]\!] \backslash \{\texttt{x}\}) \cup \{\texttt{x,y}\}$
$[\![\texttt{z} = \texttt{x} - 4]\!] = ([\![\texttt{z} > 0]\!] \backslash \{\texttt{z}\}) \cup \{\texttt{x}\}$
$[\![\texttt{z} > 0]\!] = [\![\texttt{x} = \texttt{x}/2]\!] \cup [\![\texttt{z} = \texttt{z} - 1]\!]$
$[\![\texttt{x} = \texttt{x}/2]\!] = ([\![\texttt{z} = \texttt{z} - 1]\!] \backslash \{\texttt{x}\}) \cup \{\texttt{x}\}$
$[\![\texttt{z} = \texttt{z} - 1]\!] = ([\![\texttt{x} > 1]\!] \backslash \{\texttt{z}\}) \cup \{\texttt{z}\}$
$[\![\texttt{ecrire x}]\!] = [\![exit]\!] \cup \{\texttt{x}\}$
$[\![exit]\!] = \{\}$

Les étape de la mise en œuvre de l'algorithme de calcul du point fixe conduisent à la plus petite solution et sont décrites par le tableau 2.1 :

Nœud	**Passe 1**	**Passe 2**	**Passe 3**	**Passe 4**
$[\![entry]\!]$	{}	{}	{}	{}
[\![var x,y,z]\!]	{}	{}	{}	{}
$[\![\texttt{x} = \texttt{lire}]\!]$	{}	{}	{}	{}
$[\![\texttt{x} > 1]\!]$	{}	{x}	{x}	{x}
$[\![\texttt{y} = \texttt{x}/2]\!]$	{}	{x}	{x}	{x}
$[\![\texttt{y} < 3]\!]$	{}	{}	{x,y}	{x,y}
$[\![\texttt{x} = \texttt{x} - \texttt{y}]\!]$	{}	{x,y}	{x,y}	{x,y}
$[\![\texttt{z} = \texttt{x} - 4]\!]$	{}	{x}	{x}	{x}
$[\![\texttt{z} > 0]\!]$	{}	{}	{x,z}	{x,z}
$[\![\texttt{x} = \texttt{x}/2]\!]$	{}	{x}	{x,z}	{x,z}
$[\![\texttt{z} = \texttt{z} - 1]\!]$	{}	{x,z}	{x,z}	{x,z}
[\![ecrire x]\!]	{}	{x}	{x}	{x}
$[\![exit]\!]$	{}	{}	{}	{}

TAB. 2.1 – *Calcul de point fixe*

A l'examen de ces résultats, il apparaît clairement que les variables `y` et `z` ne sont jamais vives en même temps. Il apparait aussi que l'affectation de la variable `z` dans l'instruction `z = z-1` n'est jamais utilisée : en effet la variable `z` n'est pas vive en sortie du nœud associé à cette instruction. Cette instruction est inutile. Un compilateur pourrait utiliser ces deux considérations pour récrire le code de ce programme ainsi :

```
var x,yz;
x = lire ;
while (x>1) {
     yz = x/2;
```

```
     if (yz>3) x = x-yz;
     yz = x-4;
     if (yz>0) x = x/2;
}
ecrire x ;
```

ce qui permettrait d'économiser le coût d'une instruction d'affectation (dans le corps d'une boucle) et d'obtenir de meilleurs résultats tant en occupation mémoire qu'en allocation de registres.

2.6.3 Application aux expressions disponibles

Une expression dans un programme est *disponible* en un point de ce programme si sa valeur courante a déjà été calculée antérieurement dans l'exécution du programme. L'ensemble des expressions disponibles aux différents points d'exécution du programme peut être approché en réalisant une analyse du flot des données. Le treillis que l'on utilise pour ce calcul est celui composé à partir de toutes les expressions apparaissant dans le programme. La relation d'ordre est la relation de *contenance*, inverse de la relation d'inclusion des sous-ensembles. Soit par exemple le programme

```
var x,y,z,a,b;
z = a+b;
y = a*b;
while (y > a+b) {
     a = a+1;
     x = a+b;
}
```

qui utilise quatre expressions complexes. Le treillis utilisé peut être déclaré comme

$$L = (2^{\{\texttt{a+b},\texttt{a*b},\texttt{y>a+b},\texttt{a+1}\}}, \supseteq)$$

et prend la configuration donnée par la figure 2.11

Le graphe de flot de contrôle du programme est donné par la figure 2.12.

Pour chacun des nœuds v, on introduit la variable $[\![v]\!]$ à valeurs dans L. Cette variable contient le sous-ensemble des expressions disponibles en sortie du nœuds v. Par exemple, l'expression `a+b` est disponible pour l'instruction conditionnelle de la boucle du programme ci-dessus, mais n'est pas disponible à l'entrée de la dernière instruction d'affectation de cette même boucle. L'analyse réalisée est sûre et conservative : les ensembles construits peuvent être plus restreints que la réalité de l'exécution. Une expression analysée disponible le sera. Les contraintes sur les flux de données s'expriment à l'aide d'un nouvel opérateur $COLLECTE$ définit cette fois-ci par

$$COLLECTE(v) = \bigcap_{w \in pred(v)} [\![w]\!]$$

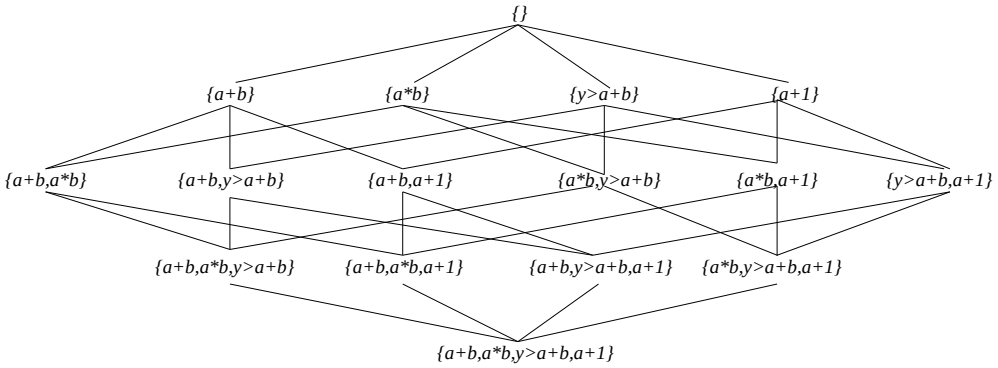

FIG. 2.11 – *Treillis des expressions du programme*

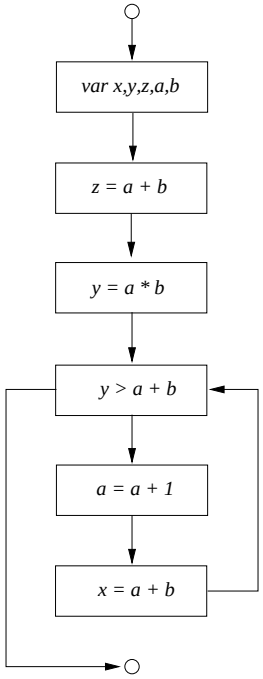

FIG. 2.12 – *Graphe du flot de contrôle du programme*

Pour le nœud d'entrée, on a la contrainte :

$$[\![entry]\!] = \{\}$$

Pour les instructions conditionnelles comportant une condition E ou les instructions `ecrire` E on se donne la contrainte

$$[\![v]\!] = COLLECTE(v) \cup exps(E)$$

Pour une instruction d'affectation de la forme $id = E$ la contrainte est alors

$$[\![v]\!] = (COLLECTE(v) \cup exps(E)) \downarrow \{id\}$$

Finalement les autres nœuds se voient attribuer la contrainte

$$[\![v]\!] = COLLECTE(v)$$

Il convient de préciser que la fonction $\downarrow id$ retire toutes les expressions qui contiennent une référence à la variable id et la fonctions $exps(E)$ dénote l'ensemble des expressions participant à construire E. On peut construire $exps(E)$ ainsi :

$$\begin{aligned}
&exps(int_const) = \emptyset \\
&exps(id) = \emptyset \\
&exps(\texttt{lire}) = \emptyset \\
&exps(E_1 \texttt{op} E_2) = \{E_1 \texttt{op} E_2\} \cup exps(E_1) \cup exps(E_2)
\end{aligned}$$

Intuitivement, une expression est disponible dans v si elle est disponible sur tous les prédécesseurs de v ou si elle est calculée par v, à moins que sa valeur soit modifiée par une instruction d'affectation. De nouveau, on peut montrer assez simplement que les parties droites des équations de contraintes constituent des fonctions monotones sur l'ensemble des variables v_i.

L'examen du programme génère les ensembles de contraintes suivantes

$$\begin{aligned}
&[\![entry]\!] = \{\} \\
&[\![\texttt{var x,y,z,a,b}]\!] = [\![entry]\!] \\
&[\![\texttt{z} = \texttt{a} + \texttt{b}]\!] = exps(\texttt{a} + \texttt{b}) \\
&[\![\texttt{y} = \texttt{a} * \texttt{b}]\!] = ([\![\texttt{x} = \texttt{a} + \texttt{b}]\!] \cup exps(\texttt{a} * \texttt{b})) \downarrow \{\texttt{y}\} \\
&[\![\texttt{y} > \texttt{a} + \texttt{b}]\!] = ([\![\texttt{y} = \texttt{a} * \texttt{b}]\!] \cap [\![\texttt{x} = \texttt{a} + \texttt{b}]\!]) \cup exps(\{\texttt{y} > \texttt{a} + \texttt{b}\}) \\
&[\![\texttt{a} = \texttt{a} + 1]\!] = ([\![\texttt{y} > \texttt{a} + \texttt{b}]\!] \cup exps(\texttt{a} + 1)) \downarrow \{\texttt{a}\} \\
&[\![\texttt{x} = \texttt{a} + \texttt{b}]\!] = ([\![\texttt{a} = \texttt{a} + 1]\!] \cup exps(\texttt{a} + \texttt{b})) \downarrow \{\texttt{x}\} \\
&[\![exit]\!] = [\![\texttt{y} > \texttt{a} + \texttt{b}]\!]
\end{aligned}$$

En mettant en œuvre l'algorithme de recherche de point fixe sur ce système d'équation, on aboutit à la plus petite solution suivante

$$\begin{aligned}
&[\![entry]\!] = \{\} \\
&[\![\texttt{var x,y,z,a,b}]\!] = \{\} \\
&[\![\texttt{z} = \texttt{a} + \texttt{b}]\!] = \{\texttt{a} + \texttt{b}\} \\
&[\![\texttt{y} = \texttt{a} * \texttt{b}]\!] = \{\texttt{a} + \texttt{b}, \texttt{a} * \texttt{b}\} \\
&[\![\texttt{y} > \texttt{a} + \texttt{b}]\!] = \{\texttt{a} + \texttt{b}, \texttt{a} * \texttt{b}, \texttt{y} > \texttt{a} + \texttt{b}\} \\
&[\![\texttt{a} = \texttt{a} + 1]\!] = \{\} \\
&[\![\texttt{x} = \texttt{a} + \texttt{b}]\!] = \{\texttt{a} + \texttt{b}\} \\
&[\![exit]\!] = \{\texttt{a} + \texttt{b}\}
\end{aligned}$$

qui confirme le diagnostic concernant `a+b` posé plus haut. On obervera que les variables disponible en un point du programme situé avant un nœud v sont en fait $COLLECTE(v)$. Sur la base de ces observations, il est possible de changer le programme en une version devenue plus efficace :

```
var x,y,z,a,b,aplusb;
aplusb = a+b;
z = aplusb;
y = a*b;
while (y > aplusb) {
     a = a+1;
     aplusb = a+b;
     x = aplusb;
}
```

2.6.4 Application aux expressions utilisables

Une expression est *utilisable* si, une fois calculée, elle est évaluée de nouveau avant que sa valeur ne change. Pour approcher cette propriété, on utilise le même treillis et les mêmes fonctions auxilliaires que pour les expressions disponibles. Pour chaque nœud v, la variable $[\![v]\!]$ dénote l'ensemble des expressions qui, à l'entrée du nœud, sont utilisables. On se donne une nouvelle définition de l'ensemble $COLLECTE$

$$COLLECTE(v) = \bigcap_{w \in succ(v)} [\![w]\!]$$

Pour le nœud de sortie, on a la contrainte :

$$[\![exit]\!] = \{\}$$

Pour les instructions conditionnelles comportant une condition E ou les instructions `ecrire` E on se donne la contrainte

$$[\![v]\!] = COLLECTE(v) \cup exps(E)$$

Pour une instruction d'affectation de la forme $id = E$ la contrainte est alors

$$[\![v]\!] = (COLLECTE(v) \downarrow \{id\}) \cup exps(E)$$

Finalement les autres nœuds se voient attribuer la contrainte

$$[\![v]\!] = COLLECTE(v)$$

Intuitivement, une expression est utilisable si elle est évaluée dans le nœud courant ou si elle doit être évaluée dans une étape d'exécution ultérieure à moins qu'une instruction d'affectation ne change sa valeur. Sur le programme suivant :

```
var x,a,b;
```

```
x = lire;
a = x-1;
b = x-2;
while (x>0) {
     ecrire a*b-x;
     x = x-1;
}
ecrire a*b;
```

l'analyse révèle que l'expression `a*b` est utilisable dans le corps de la boucle. On peut ainsi déplacer son évaluation (son calcul) hors de la boucle, en un point antérieur de l'exécution ou cette expression conservera son caractère d'expression utilisable. Cela pourrait conduire à une structure de programme plus effeicace :

```
var x,a,b,afoisb;
x = lire;
a = x-1;
b = x-2;
afoisb = a*b;
while (x>0) {
     ecrire afoisb-x;
     x = x-1;
}
ecrire afoisb;
```

2.6.5 Applications aux définitions vives

Les définitions vives en un point donnée d'un programme sont les instructions d'affectation au sein du programme qui ont pu définir la valeur courante en ce point des variables du programme. Pour effectuer cette analyse, on se dote d'un treillis construit sur l'ensemble des instructions d'affectation (qui sont repérées par des nœuds du graphe de flot de contrôle) présentes dans le programme. Par exemple, sur le programme suivant :

```
var x,y,z;
x = lire;
while (x>1) {
     y = x/2;
     if (y>3) x = x-y;
     z = x-4;
     if (z>0) x = x/2;
     z = z-1;
}
ecrire x;
```

le treillis est défini par

$$L = (2^{\texttt{x=lire,y=x/2,x=x-y,z=x-4,x=x/2,z=z-1}}, \subseteq)$$

Pour tout nœud v, la variable $[\![v]\!]$ dénote l'ensemble des affectations qui définissent les valeurs de variables au point du programme situé après le nœud. On définit alors $COLLECTE$

$$COLLECTE(v) = \bigcup_{w \in pred(v)} [\![w]\!]$$

Pour une instruction d'affectation de la forme $id = E$ la contrainte est alors

$$[\![v]\!] = (COLLECTE(v) \downarrow \{id\}) \cup \{v\}$$

Finalement les autres nœuds se voient attribuer la contrainte

$$[\![v]\!] = COLLECTE(v)$$

La fonction $\downarrow \{id\}$ retire de l'ensemble auquel elle s'applique toutes les instructions d'affectation de la variable id.

Le graphe de flot de contrôle du programme ci-dessus est donnée par la figure 2.13. Il est dès lors possible, en se fondant sur les définitions des ensembles $[\![v]\!]$ ci-dessus les équations de contraintes suivantes :

$$\begin{aligned}
&[\![\texttt{x} = \texttt{lire}]\!] = \{\texttt{x} = \texttt{lire}\}\\
&[\![\texttt{x} > 1]\!] = [\![\texttt{x} = \texttt{lire}]\!] \cup [\![\texttt{z} = \texttt{z} - 1]\!]\\
&[\![\texttt{y} = \texttt{x}/2]\!] = [\![\texttt{x} > 1]\!] \downarrow \{\texttt{y}\} \cup \{\texttt{y} = \texttt{x}/2\}\\
&[\![\texttt{y} > 3]\!] = [\![\texttt{y} = \texttt{x}/2]\!]\\
&[\![\texttt{x} = \texttt{x} - \texttt{y}]\!] = [\![\texttt{y} > 3]\!] \downarrow \{\texttt{x}\} \cup \{\texttt{x} = \texttt{x} - \texttt{y}\}\\
&[\![\texttt{z} = \texttt{x} - 4]\!] = ([\![\texttt{x} = \texttt{x} - \texttt{y}]\!] \cup [\![\texttt{y} > 3]\!]) \downarrow \{\texttt{z}\} \cup \{\texttt{z} = \texttt{x} - 4\}\\
&[\![\texttt{z} > 0]\!] = [\![\texttt{z} = \texttt{x} - 4]\!]\\
&[\![\texttt{x} = \texttt{x}/2]\!] = [\![\texttt{z} > 0]\!] \downarrow \{\texttt{x}\} \cup \{\texttt{x} = \texttt{x}/2\}\\
&[\![\texttt{z} = \texttt{z} - 1]\!] = ([\![\texttt{x} = \texttt{x}/2]\!] \cup [\![\texttt{z} > 0]\!]) \downarrow \{\texttt{z}\} \cup \{\texttt{z} = \texttt{z} - 1\}\\
&[\![\texttt{ecrire x}]\!] = [\![\texttt{x} > 1]\!]
\end{aligned}$$

Le tableau 2.2 présente l'application de l'algorithme de calcul de point fixe appliqué à cet ensemble d'équations de contraintes. Il donne l'ensemble des définitions vives en sortie de chacun des nœuds du graphe de flot de contrôle, après exécution de chacune des instuctions constituant le nœud.

Ces résultats être utilisée pour construire un graphe $d - u$ associant les définitions et les utilisations de variables en reliant les nœuds correpondants du programme.

Par exemple, dans le programme ci-dessus, en se fondant sur le graphe de la figure 2.13 et sur le tableau des résultats 2.2 on peut s'intéresser aux définitions de la variable `x` qui sont

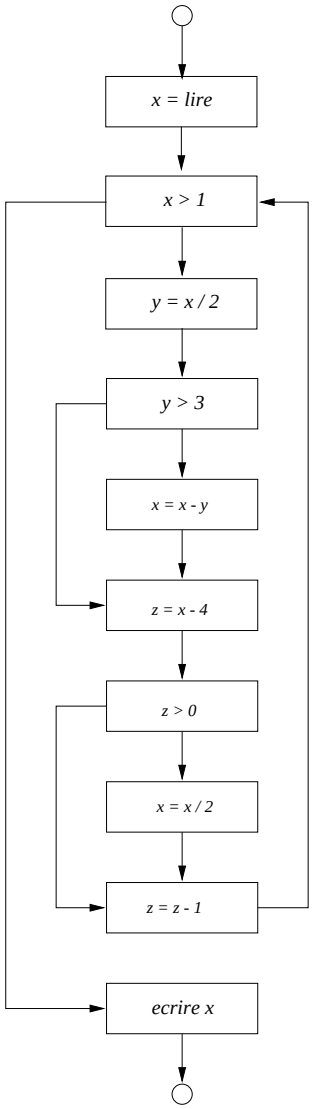

FIG. 2.13 – *Graphe de flot de contrôle du programme*

vives à l'évaluation de la conditionnelle `x>1`. Les nœuds du graphe de flot de contrôle qui précèdent le nœud associé à cette instruction sont ceux constitués par les intructions `x=lire` et `z=z-1`. La définition de l'opérateur $COLLECTE$ indique donc que c'est l'union des définitions vives en sortie de ces deux nœuds prédécesseurs qui constituent les définitions vives pour l'évaluation de la conditionnelle : `{1,2,3,5,6}`. Parmi ces définitions, seules les définitions `{1,3,5}` définissent `x`. On relie donc les nœuds correspondant à ces affectations (`x = lire`, `x = x-y`, `x = x/2`) au nœud associé à la conditionnelle `x>1`. Pour le programme ci-dessus, le graphe $d - u$ est donné par la figure 2.14.

Ce type de graphe est très intéressant. Il permet en particulier de procéder à l'élimination de code inutile ou au déplacement avantageux d'instructions.

Nœud	Passe 1	Passe 2	Passe 3
(1) $[\![x = lire]\!]$	{}	{1}	{1}
$[\![x > 1]\!]$	{}	{1}	{1,2,3,5,6}
(2) $[\![y = x/2]\!]$	{}	{1,2}	{1,2,3,5,6}
$[\![y > 3]\!]$	{}	{1,2}	{1,2,3,5,6}
(3) $[\![x = x - y]\!]$	{}	{2,3}	{2,3,6}
(4) $[\![z = x - 4]\!]$	{}	{1,2,3,4}	{1,2,3,4,5}
$[\![z > 0]\!]$	{}	{1,2,3,4}	{1,2,3,4,5}
(5) $[\![x = x/2]\!]$	{}	{2,4,5}	{2,4,5}
(6) $[\![z = z - 1]\!]$	{}	{1,2,3,5,6}	{1,2,3,5,6}
$[\![ecrire\ x]\!]$	{}	{1}	{1,2,3,5,6}

TAB. 2.2 – *Calcul du point fixe*

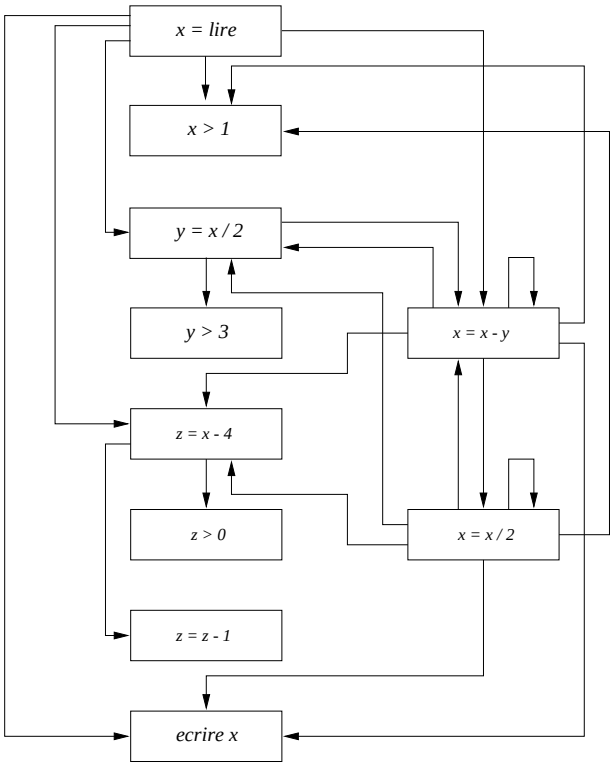

FIG. 2.14 – *Graphe $d - u$ du programme*

2.6.6 Types d'analyses de flots de données

Les analyses précédemment étudiées et relatives aux flots de données dans un programme peuvent être classées en fonction de la structure des contraintes qui définissent les variables associées aux nœuds du graphe de flot de contrôle.

Une analyse *avant* (en anglais *forwards*) est une analyse qui calcule une information relative à chacun des points du programme en fonction du comportement passé de ce programme. En d'autres termes, le calcul s'effectue en fonction de l'information collectée sur les prédécesseurs de ce point dans le graphe de flot de contrôle, et les expressions en partie droite des expressions de contraintes font intervenir les prédécesseurs des nœuds. Le calcul suit en quelque sorte le flot de contrôle. Les analyses sur les expressions disponibles ou sur les définitions vives constituent des exemples de ce type d'analyse.

Une analyse *arrière* (en anglais *backwards*) est une analyse qui calcule une information relative à chacun des points du programme en fonctionnement du comportement futur de ce programme. En d'autres termes, le calcul s'effectue en fonction de l'information collectée sur les successeurs de ce point dans le graphe de flot de contrôle,et les expressions en partie droite des expressions de contraintes font intervenir les successeurs des nœuds. Le calcul est réalisé à rebours du flot de contrôle. Les analyses sur les expressions utilisables ou sur la vivacité constituent des exemples de ce type d'analyse.

Une analyse *du possible* (en anglais *may*) décrit une information possiblement vraie et calcule ainsi une sur-approximation. Le calcul de vivacité ou des définitions vives constituent des exemples de ce type d'analyse. Elles sont caractérisées par le fait qu'elles font apparaître un opérateur $\bigcup$ en partie droite de l'expression des contraintes.

Une analyse *du certain* (en anglais *must*) décrit une information quidoit être vraie à coup sûr, et se fonde ainsi sur le calcul d'une sous-approximation. Le calcul des expressions disponibles ou des expressions utilisables constituent des exemples de ce type d'analyse. Elles sont caractérisées par le fait qu'elles font apparaître un opérateur $\bigcap$ en partie droite de l'expression des contraintes.

Le tableau ci-dessous récapitule cette classification ou ce typage :

	Avant	*Arrière*
Possible	Définitions vives	Vivacité
Certain	Expressions disponibles	Expressions utilisables

2.6.7 Exemple d'application aux variables initialisées

On cherche à définir une analyse qui permette de s'assurer que les variables sont intialisées avant d'être utilisées. Il faut donc calculer, en chaque point du programme, l'ensemble des variables intialisées. Le treillis utilisé est celui construit sur l'ensemble des variables du programme. L'intialisation est une propriété du passé. Le type d'analyse opérée est donc une analyse *avant*. D'autre part, on cherche une information certaine, donc le type d'analyse est une analyse *du certain*. Les contraintes s'expriment donc en termes de prédécesseurs et d'intersections. On peut dégager l'ensemble des contraintes suivantes permettant de définir l'analyse. Pour le nœud d'entrée on définit la contrainte

$$[\![entry]\!] = \{\}$$

Pour les instructions d'affectation la contrainte est alors

$$[\![v]\!] = \bigcap_{w \in pred(v)} [\![w]\!] \cup \{id\}$$

Pour tous les autres nœuds, la contrainte est

$$[\![v]\!] = \bigcap_{w \in pred(v)} [\![w]\!]$$

Ce calcul permet donc de propager, depuis le point d'entrée du programme, l'ensemble des variables qui sont intialisées par une instruction d'affectation. Un compilateur peut ainsi vérifier qu'une variable utilisée dans une expression a effectivement été intialisée.

2.6.8 Exemple d'application au calcul des signes

Un exemple désormais classique en analyse statique des programmes est celui du calcul du signe des expressions (+,0,-). Pour cela, on utilise le treillis *Signes* particulièrement simple qu'illustre la figure 2.15.

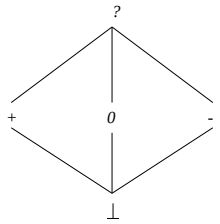

FIG. 2.15 – *Treillis des signes*

Le symbole ? dénote que la valeur du signe d'une expression n'est pas constante tandis que le symbole $\perp$ dénote que cette valeur n'est pas connue, est indéterminée. Le treillis complet sur lequel se fonde l'analyse est le treillis de la relation

$$Vars \mapsto Signes$$

où $Vars$ est l'ensemble des variables du programme. Pour chacun des nœuds v du graphe de flot de contrôle, on affecte la variable $[\![v]\!]$ qui dénote une table donnant le signe de toutes les variables du programme à l'entrée du nœud. On définit l'opérateur

$$COLLECTE(v) = \bigsqcup_{w \in pred(v)} [\![w]\!]$$

Pour les déclarations de variables on utilise la fonction constante retournant ?

$$[\![v]\!] = [id_1 \mapsto ?, \ldots, id_n \mapsto ?]$$

tandis que pour une instrution d'affectation (`id = E`) la contrainte sera

$$[\![v]\!] = COLLECTE(v)[id \mapsto eval(COLLECTE(v),E)]$$

et pour tous les autres nœuds la contrainte sera

$$[\![v]\!] = COLLECTE(v)$$

La fonction *eval* réalise une évaluation abstraite des expressions et est définie par :

$$eval(\sigma,id) = \sigma(id)$$
$$eval(\sigma,const\_int) = signe(const\_int)$$
$$eval(\sigma,E_1 \texttt{ op } E_2) = \overline{\texttt{op}}(eval(\sigma,E_1),eval(\sigma,E_2))g$$

où *signe* donne le signe d'une constante entière et $\overline{\texttt{op}}$ est une définition abstraite de l'opérateur que l'on peut définir de la manière décrite par les tables ci-dessous.

+	⊥	0	-	+	?
⊥	⊥	⊥	⊥	⊥	⊥
0	⊥	0	-	+	?
-	⊥	-	-	?	?
+	⊥	+	?	+	?
?	⊥	?	?	?	?

-	⊥	0	-	+	?
⊥	⊥	⊥	⊥	⊥	⊥
0	⊥	0	+	-	?
-	⊥	-	?	-	?
+	⊥	+	+	?	?
?	⊥	?	?	?	?

*	⊥	0	-	+	?
⊥	⊥	0	⊥	⊥	⊥
0	0	0	0	0	0
-	⊥	0	+	-	?
+	⊥	0	-	+	?
?	⊥	0	?	?	?

/	⊥	0	-	+	?
⊥	⊥	⊥	⊥	⊥	⊥
0	⊥	?	0	0	?
-	⊥	?	?	?	?
+	⊥	?	?	?	?
?	⊥	?	?	?	?

>	⊥	0	-	+	?
⊥	⊥	0	⊥	⊥	⊥
0	⊥	0	+	0	?
-	⊥	0	?	0	?
+	⊥	+	+	?	?
?	⊥	?	?	?	?

==	⊥	0	-	+	?
⊥	⊥	⊥	⊥	⊥	⊥
0	⊥	+	0	0	?
-	⊥	0	?	0	?
+	⊥	0	0	?	?
?	⊥	?	?	?	?

L'abstraction réalisée par cette analyse engendre inéluctablement une perte d'information. C'est ainsi par exemple que l'expression `(4>0)==1` est évaluée à la valeur `?`, ce qui ne constitue pas un résultat très affiné. On remarquera aussi que l'évaluation de `+/+` donne

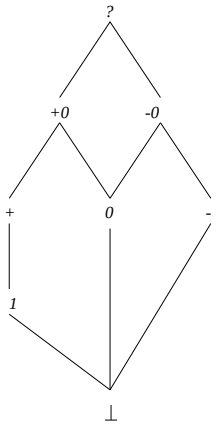

FIG. 2.16 – *Treillis des signes enrichi*

`?` : ceci parce que l'on travaille avec des valeurs entières et que `x/y` avec `x<y` est arrondi à la valeur `0`.

Pour traiter ces situations de façon plus précise, il faudrait enrichir le treillis des signes avec des valeurs abstraites supplémentaires telles `1`, `+0` (pour ≥ 0), `-0` (pour ≤ 0) afin de garder trace des expresions évaluées. La figure 2.16 illustre ce treillis renrichi. Parallèlement, il faudrait également étendre la définition des opérateurs abstraits avec des tables de 8×8 opérateurs. Le tableau suivant donne l'abstraction de l'opérateur d'addition.

+	⊥	1	+	0	-	+0	-0	?
⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥
1	⊥	+	+	1	?	+	?	?
+	⊥	+	+	+	?	+	?	?
0	⊥	1	+	0	-	+0	-0	?
-	⊥	?	?	-	-	?	-	?
+0	⊥	+	+	+0	?	+0	?	?
-0	⊥	?	?	-0	-	?	-0	?
?	⊥	?	?	?	?	?	?	?

De manière générale, on trouvera d'autres éléments concernant les analyses de flots de données dans [ASU86]

2.6.9 Exemple d'application à la propagation des constantes

L'analyse de propagation des constantes cherche à déterminer, en tout point du programme, les variables qui ont une valeur constante. L'analyse se structure comme celle de l'analyse des signes décrite en section 2.6.8, avec un treillis comme celui de la figure 2.17

Les opérateurs sont tous abstraits de la manière suivante, par exemple en ce qui concerne

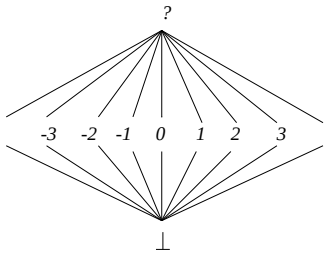

FIG. 2.17 – *Treillis des constantes*

l'addition

$$\lambda n \lambda m.\ \texttt{if}\ (n \neq ? \wedge m \neq ?)\ \{n+m\}\ \texttt{else}\ \{?\}$$

Sur la base de cette analyse, le programme suivant

```
var x,y,z;
x = 15;
y = lire;
z = 4*x+y;
if (x<0) {y = z-3;} else {y=12;}
ecrire y;
```

peut être traduit, en propageant les constantes en :

```
var x,y,z;
x = 15;
y = lire;
z = 60+y;
if (0) {y = z-3;} else {y=12;}
ecrire y;
```

programme qui, suite à une analyse des définitions vives et à une analyse d'élimination de code, peut encore être réduit en :

```
ecrire 12;
```

2.7 Extension et réduction

Une analyse d'intervalle calcule pour chaque variable entière une borne inférieure et une borne supérieure de ses valeurs possibles. Le treillis permettant de décrire une variable unique est défini par

$$Interval = (\{[l,h] \mid l,h \in N\ \wedge\ l \leq h\}, \sqsubseteq)$$

avec

$$N = \{-\infty, \ldots, -2, -1,0,1,2, \ldots, +\infty\}$$

est l'ensemble des entiers naturels étendu des points infinis. L'ordre sur les intervalles est défini par

$$[l_1,h_1] \sqsubseteq [l_2,h_2] \Leftrightarrow l_2 \leq l_1 \ \wedge \ h_1 \leq h_2$$

et correspond à une inclusion de points. Ce treillis est illustré par la figure 2.18.

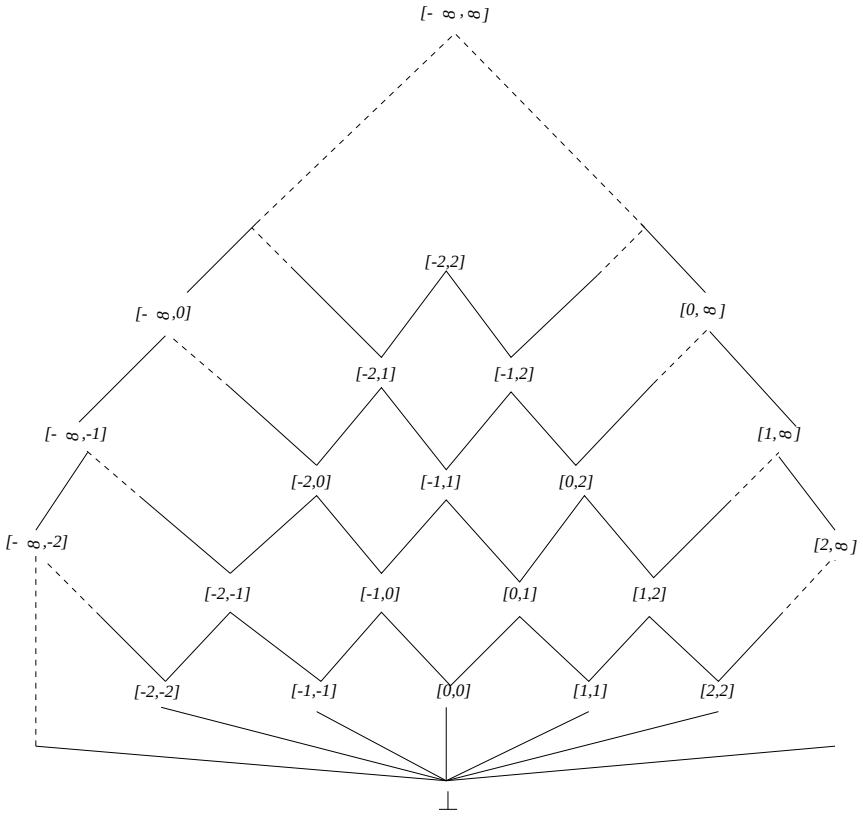

FIG. 2.18 – *Treillis des intervalles*

Il est clair que l'on n'a pas un treillis de taille finie, puisqu'il peut contenir par exemple la chaîne

$$[0,0] \sqsubseteq [0,1] \sqsubseteq [0,2] \sqsubseteq [0,3] \sqsubseteq [0,4] \sqsubseteq [0,5] \ldots$$

Le treillis que l'on devrait utiliser est donné par

$$L = Vars \mapsto Interval$$

où, pour le nœud d'entrée, on retourne la fonction constante retournant l'élément $\top$:

$$[\![entry]\!] = \lambda x.[-\infty, +\infty]$$

alors que pour une fonction d'affectation la contrainte s'exprime comme

$$[\![v]\!] = COLLECTE(v)[id \mapsto eval(COLLECTE(v),E)]$$

et pour tous les autres nœuds

$$[\![v]\!] = COLLECTE(v)$$

avec

$$COLLECTE(v) = \bigcup_{w \in pred(v)} [\![w]\!]$$

et avec la fonction *eval* réalisant l'évaluation abstraite des expressions :

$$eval(\sigma,id) = \sigma(id)$$
$$eval(\sigma,const_int) = [const\_int,const\_int]$$
$$eval(\sigma,E_1 \texttt{ op } E_2) = \overline{\texttt{op}}(eval(\sigma,E_1),eval(\sigma,E_2))g$$

où les opérateurs arithmétiques abstraits sont tous définis par :

$$\overline{op}([l_1,h1],[l_2,h2]) = [\texttt{min}_{x \in [l_1,h_1], y \in [l_2,h_2]}\ x \texttt{ op } y, \texttt{max}_{x \in [l_1,h_1], y \in [l_2,h_2]} x \texttt{ op } y]$$

et où les opérateurs de comparaison sont définis par

$$\overline{op}([l_1,h1],[l_2,h2]) = [0,1]$$

Le treillis des intervalles a une taille infinie. Dès lors, il n'est pas envisageable d'utiliser l'algorithme de calcul de convergence vers le point fixe fondé sur la monotonie des fonctionnelles puisque cet algorithme ne converge plus dans ce cas. Cela signifie que pour le treillis L^n la séquence d'approximants $F_i(\top,\ldots,\top)$ ne converge pas non plus. On utilise une texhnique d'*extension* consistant à introduire une fonction $w : L^n \to L^n$ telle que la séquence

$$(F \circ w)^i(\top,\ldots,\top)$$

qui converge vers un point fixe plus grand que tout $F_i(\top,\ldots,\top)$. La fonction d'*extension* grossit l'information sur le programme pour permettre d'assurer la terminaison des calculs. Cette fonction opère sur un sous-ensemble fini $B \subset N$ qui doit contenir $-\infty$ et ∞. Dans la pratique, B peut être initialement construit en intégrant toutes les constantes entières qui apparaissent dans le programme. Pour un intervalle on a :

$$w([l,h]) = [max\{i \in B \mid i \leq l\},min\{i \in B \mid h \leq i\}]$$

L'application de cette extension fournit un résultat plus grand que le résultat ciblé. Il s'agit alors de réduire la solution obtenue à travers une opération de *réduction* qui permet d'affiner le résultat.

Si l'on définit $ptfix = \bigsqcup F^i(\top,\ldots,\top)$ et $ptfixw = \bigsqcup (F \circ w)^i(\top,\ldots,\top)$ alors on a $ptfix \sqsubseteq ptfixw$. Cependant, on a aussi $ptfix \sqsubseteq F(ptfixw) \sqsubseteq ptfixw$, ce qui signifie qu'une application de F affine le résultat obtenu.

On peut prendre l'exemple du programme suivant :

```
y = 0;
```

```
while (lire) {
     x = 7;
     x = x + 1;
     y = y + 1;
}
```

Sans faire appel aux techniques d'extension, l'analyse permet d'obtenir la séquence d'approximations divergente qui suit au point situé après l'instruction de boucle :

$[\mathtt{x} \mapsto \top, \mathtt{y} \mapsto [0,0]]$
$[\mathtt{x} \mapsto [8,8], \mathtt{y} \mapsto [0,1]]$
$[\mathtt{x} \mapsto [8,8], \mathtt{y} \mapsto [0,2]]$
$[\mathtt{x} \mapsto [8,8], \mathtt{y} \mapsto [0,3]]$
.
.
.

Si on applique une fonction d'*extension* en se fondant sur l'existence d'un ensemble B que l'on peut définir par $B = \{-\infty,0,1,7,\infty\}$ construit à l'aide des constantes apparaissant dans le programme, on obtient alors la séquence convergente qui suit :

$[\mathtt{x} \mapsto \top, \mathtt{y} \mapsto [0,0]]$
$[\mathtt{x} \mapsto [7,\infty], \mathtt{y} \mapsto [0,1]]$
$[\mathtt{x} \mapsto [7,\infty], \mathtt{y} \mapsto [0,7]]$
$[\mathtt{x} \mapsto [7,\infty], \mathtt{y} \mapsto [0,\infty]]$

Le résultat obtenu concernant `x` est décevant. Mais une application de la fonction de réduction sur ce résultat permet d'affiner en

$[\mathtt{x} \mapsto [8,8], \mathtt{y} \mapsto [0,\infty]]$

2.8 Analyse interprocédurale

Les techniques d'analyse présentées jusqu'à présent portent sur le corps d'une fonction uniquement. On parle dans ce cas d'analyse *intraprocédurale*. Lorsque l'on considère des programmes complets, comportant des appels de fonctions, il est nécessaire de procéder à des analyses *interprocédurales*. Une alternative possible consiste à analyse chaque fonction isolément en faisant les hypothèses les plus pessimistes sur les résultats obtenus par appels de fonctions.

On trouvera dans des ouvrages généraux comme [ASU86] [WM94] [Wol96] [Muc97] ou [NNH99] la description de techniques d'analyse interprocédurale. On pourra également se reporter à [Hav94].

2.8.1 Graphes de flot de contrôle de programmes

On se place toujours en l'absence de pointeurs. Pour construire le graphe de flot de contrôle d'un programme comportant des appels de fonctions, on procède comme si l'on substituait à ces appels de fonction le corps de la fonction appelée.

On commence par construire le flot de contrôle de chacun des corps de fonctions. On connecte ou couple ces graphes entre eux pour refléter fidèlement la structure des appels de fonctions.

Pour cela, on introduit un ensemble de variables supplémentaires. Pour chaque fonction `f`, on introduit la variable `ret_f` qui contient la valeur retournée par l'instruction `return` de la fonction. En chaque point du programme où une fonction est appelée, on introduit la variable `appel_i`, avec `i` comme un identifieur unique, qui doit contenir la valeur calculée par l'appel de fonction. En chaque poit d'appel, pour chacune des variables locales ou des paramètres formels de la fonction *appelante* dénommés `x`, on introduit la variable de sauvegarde `sauve_i_x` destinée à conserver sa valeur à travers l'appel de la fonction. En tout point d'appel, et pour tout paramète formel `x` de la fonction *appelée*, on intoduit la variable temporaire `temp_i_x`.

Pour simplifier, on fait l'hypothèse que les appels de fonction constituent la partie droite d'une instruction d'affectation :

```
x = f(E1,...,En);
```

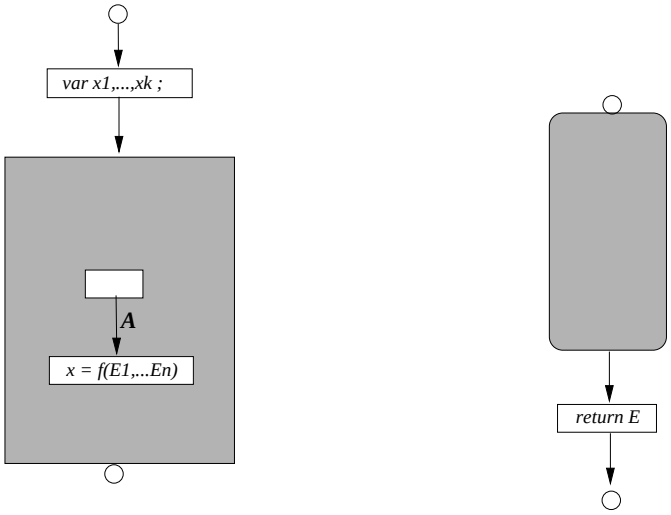

FIG. 2.19 – *Appel de fonction dans un programme*

Considérons la structure de programme définie par la figure 2.19. Le corps de la fonction de gauche comporte un appel à la fonction `f` dont le corps est décrit en partie droite de la figure. On fait l'hypothèse que la fonction appelante dispose d'un certain nombre de paramètres formels `b1,...bn` et a déclaré les variables locales `x1,...,xk`. On suppose

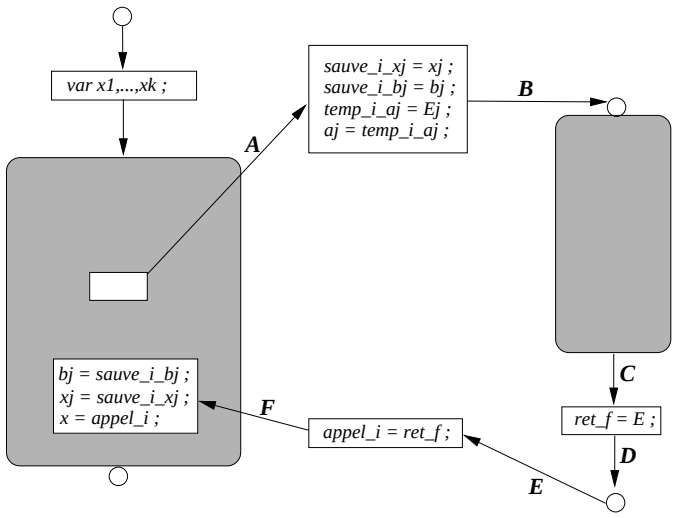

FIG. 2.20 – *Graphe de contrôle interprocédural*

également que les paramètres formels de la fonction appelée sont `a1,...am`. La figure 2.20 illustre le fragment de graphe de contrôle construit pour refléter l'appel à la fonction `f` dans le corps de la fonction appelante.

Le transfert du contrôle vers l'instruction `x=f(E1,...,En)` représenté par la flèche *A* dans la figure 2.19 est représenté par les flèches *A,B,C,D,E,F* dans la figure 2.20. Ce nouvel enchaînement réalise, dans le graphe de contrôle finalement obtenu, l'encapsulation du graphe de contrôle de la fonction appelée en réalisant les opérations suivantes :

- introduction d'une séquence d'instruction de sauvegarde des variables locales et des paramètres formels de la fonction appelante ; ce sont les instructions d'affectation des variables de type `sauve_i_bj` et `sauve_i_xj`;
- introduction des instructions de passage des paramètres; ce sont les instructions d'affectation des valeurs de paramètres réels `Ej` aux variables temporaires `temp_i_aj` et d'affectation des valeurs de ces variables aux paramètres formels `aj` de la fonction appelée ;
- transformation de l'instruction `return E` de la fonction appelée en une affectation de la variable de retour `ret_f = E` ;
- remplacement de la partie droite de l'instruction `x = f(E1,...,f(En)` dans le corps de la fonction appelante par l'instruction résultante `appel_i = ret_f`;
- introduction d'une séquence d'instructions de restauration des valeurs préalablement sauvegardées des paramètres formels et des variables locales de la fonction appelante ;
- remplacement de l'instruction `x = f(E1,...,f(En)` dans le corps de la fonction appelante par l'instruction d'affectation `x = appel_i`;

On peut essayer d'appliquer cette stratégie au programme suivant :

```
foo(x,y) {
     x = 2*y ;
     return x+1 ;
}

main() {
     var a,b ;
     a = lire ;
     b = foo(a,20);
     return b ;
}
```

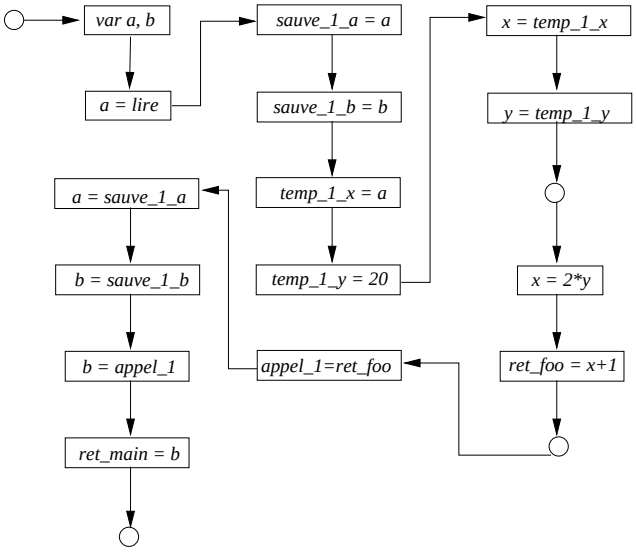

FIG. 2.21 – *Graphe de contrôle associé à un appel de fonction dans un programme*

Le graphe de flot de contrôle de la figure 2.21 est construit selon le procédé exposé. Il permet de se ramener à un cadre de traitement des analyses similaire au cadre intraprocédural.

Il faut également noter que le graphe de contrôle construit ne comporte qu'une seule description du flot de contrôle de chacune des fonctions. Ainsi par exemple, dans un programme où la fonction identité `id` serait appelée deux fois comme ci-dessous :

```
id(a) {
     return a ;
}
```

```
f1() {
     var x ;
     x = id(20);
     return x ;
}
```

```
f2()  {
      var y ;
      y = id(21);
      return y ;
}
```

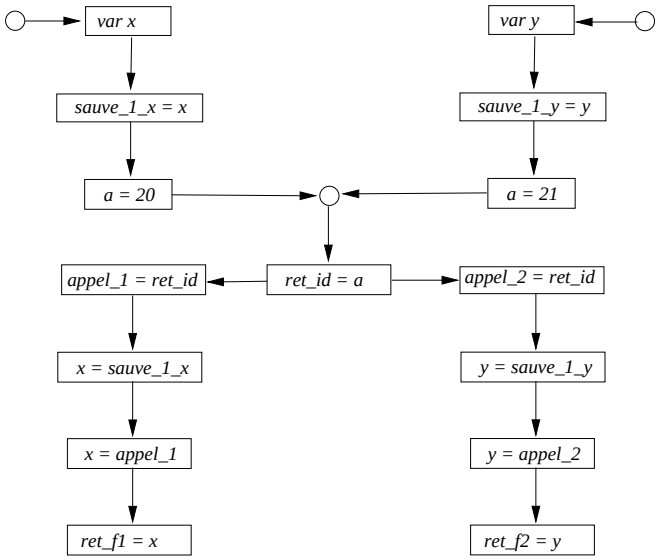

FIG. 2.22 – *Graphe de contrôle associé à deux appels de fonction dans un programme*

le graphe de contrôle est alors celui donné par la figure 2.22. Si l'on effectue une analyse de propagation de constantes sur ce graphe, alors les valeurs retournées par `f1` et `f2` ne sont pas données pour constantes à cause de la fusion des deux appels à la fonction `id`.

Pour lever cette difficulté, on crée plusieurs copies du contrôle associé au corps de la fonction appelée. On parle alors d'analyse *polyvariante* par opposition à l'approche précédente *monovariante*. Différentes tactiques peuvent être utilisées pour décider du nombre de copies à produire. La plus simple consiste à créer une copie par appel de la fonction. Ce procédé permet par exemple d'effectuer une analyse permettant le report des constantes comme en témoigne la figure 2.23.

2.8.2 Application à la recherche de code inutile

Un exemple d'application de l'analyse interprocédurale consiste à rechercher, dans un programme, les fonctions qui ne sont jamais appelées et qui introduisent par conséquent du code inutile pouvant être retiré du programme. Ce problème peut surgir lorsque l'on compile un programme avec une importante bibliothèque de fonctions. L'analyse effectuée porte sur une version monovariante du graphe de flot de contrôle du programme mais elle est formulée ici de la même façon que les analyses intraprocédurales présentées jusqu'ici.

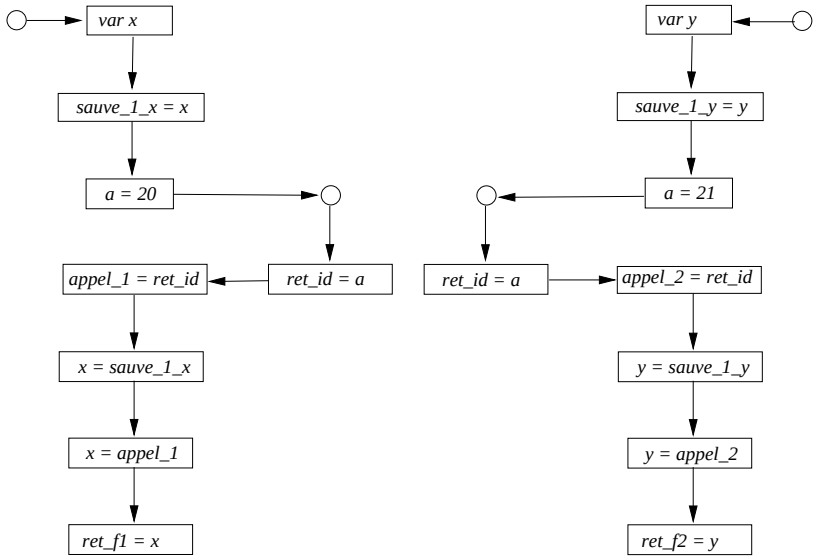

FIG. 2.23 – *Graphe de contrôle polyvariant*

Le treillis est constitué par l'ensemble puissance des noms de fonctions apparaissant dans le programme, et pour tout nœud v du graphe on introduit une variable de contrainte $[\![v]\!]$ dénotant l'ensemble des fonctions pouvant être appelées dans le futur. On note *entrée(id)* le nœud d'entrée de la fonction nommée *id*. En ce qui concerne les instructions d'affectation, les conditionnelles ou les instructions de sortie, l'expression de la contrainte est donnée par :

$$[\![v]\!] = \bigcup_{w \in succ(v)} [\![w]\!] \cup funcs(E) \cup \bigcup_{f \in funcs(E)} [\![entree(f)]\!]$$

tandis que pour tous les autres nœuds on définit

$$[\![v]\!] = \bigcup_{w \in succ(v)} [\![w]\!]$$

avec $funcs$ défini par

$$funcs(id) = funcs(const\_int) = funcs(\texttt{lire}) = \emptyset$$
$$funcs(E_1 \texttt{ op } E_2) = funcs(E_1) \cup funcs(E_2)$$
$$funcs(id(E1, \ldots, E_n)) = \{id\} \cup funcs(E_1) \cup \ldots \cup funcs(E_n)$$

Toute fonction qui n'est alors pas mentionnée dans la valeur résultante de la variable $[\![entree(\texttt{main})]\!]$ est inutile.

2.8.3 Analyse du flot de contrôle

L'analyse du flot de contrôle, dans un contexte interprocédural, se complique, par rapport au schéma présenté précédemment, dès lors que, par exemple, on se trouve en présence de pointeurs sur des fonctions. Dans ce cas, un appel de fonction calculé revêt la forme

$$E \rightarrow (E)(E_1, \ldots, E_n)$$

L'expression (E) masque l'identité réelle de la fonction appelée : cette dernière résulte de l'adresse obtenue par le calcul de l'expression lors de l'exécution du programme.

Une première approximation peut être effectuée : on considère dans ce cas que toute fonction dont la signature correspond aux paramètres d'appel est susceptible d'être invoquée. C'est cependant une approximation très grossière. Elle peut être affinée en effectuant une analyse du flot de contrôle.

Le treillis utilisé pour cette analyse est l'ensemble puissance d'un ensemble de tokens contenant les adresses $\&id$ pour chacun des noms de fonctions id ordonné par la relation d'inclusion des sous-ensembles. Pour tout nœud v de l'arbre syntaxique on introduit la variable $[\![v]\!]$ dénotant l'ensemble des fonctions ou des pointeurs sur fonctions pouvant constituer le résultat de l'évaluation opérée sur le nœud v. Pour la déclaration d'un nom de fonction id on a la contrainte

$$\{\&id\} \subseteq [\![id]\!]$$

et pour les instructions d'affectation $id = E$ on introduit la contrainte

$$[\![E]\!] \subseteq [\![id]\!]$$

et, finalement, pour les appels de fonctions calculés de type $(E)(E_1, \ldots, E_n)$ on a, pour toute définition de fonction f avec les arguments $a_1, \ldots, a_n$ et l'expression E' en retour la contrainte :

$$\&f \in [\![E]\!] \Rightarrow [\![E_i]\!] \subseteq [\![a_i]\!] \wedge [\![E']\!] \subseteq [\![(E)(E_1, \ldots, E_n)]\!]$$

On obtient au final un ensemble d'expressions de contraintes portant sur des inclusions de sous-ensembles. La solution à ce système de contraintes est donnée par la recherche d'une fermeture sur ces ensembles de contraintes.

Algorithme de recherche de la plus petite solution

Cette recherche de fermeture peut se fonder sur un algorithme mettant en œuvre un ensemble de tokens $\{t_1, \ldots, t_k\}$ et une collection de variables $\{x_1, \ldots, x_n\}$ dont les valeurs sont des sous-ensembles de tokens. Les tokens représentent en fait des adresses de fonctions du programme. L'objectif de l'algorithme est de lire une séquence de contraintes de la forme $\{t\} \subseteq x$ ou $t \in x \Rightarrow y \subseteq z$ et de produire une solution possible pour le système de contraintes.

L'algorithme utilise une structure de données simple. A chaque variable on fait correspondre un nœud dans un graphe sans cycle. Chaque nœud se voit associé un vecteur de bits

appartenant à $\{0,1\}^k$ et le vecteur est initialisé à 0^k. A chaque bit du vecteur on associe une liste de couples de variables utulisés pour représenter des contraintes : le couple (x_i,x_j) représente la contrainte $x_i \subseteq x_j$. Les arcs du graphe entre les nœuds reflètent les contraintes d'inclusions. Les vecteurs de bits représentent à tout moment la solution minimale.

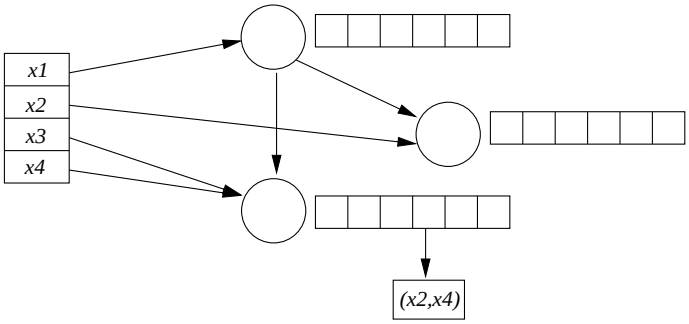

FIG. 2.24 – *Graphe de contraintes*

La figure 2.24 représente un graphe de contraintes. Celles-ci sont ajoutées une par une. Une contrainte de la forme $\{t\} \subseteq x_i$ est traitée de la façon suivante : on recherche le nœud du graphe associé à x_i et on place le bit correspondant à t du vecteur de bit à la valeur 1. Si la liste de couples de variables associée à ce bit n'est pas vide, alors on trace les arcs entre les nœuds correspondant à x_j et x_k pour tous les couples de variables (x_j,x_k). Une contrainte de la forme $t \in x_i \Rightarrow x_j \subseteq x_k$ est prise en compte en scrutant tout d'abord la valeur du bit t dans le vecteur de bit du nœud associé à x_i. Si cette valeur est 1, ce qui indique que la contrainte $t \subseteq x_i$ a déjà été rencontrée, alors on construit l'arc reliant les nœuds x_j et x_k. Sinon, la contrainte $t \subseteq x_i$ n'a pas encore été rencontrée, on introduit le couple (x_j,x_k) dans la liste des variables à relier associée à ce bit t. Les arcs correspondants seront tracés ultérieurement si on rencontre la contrainte $t \subseteq x_i$.

Si un nouvel arc introduit un cycle dans le graphe, alors tous les nœuds de ce cycle sont fondus en un seul nœud : leurs vecteurs de bits sont réunis (on en fait l'union) et les listes de couples de variables sont concaténées. La correspondance entre variables et nœuds du graphe est mise à jour en conséquence. D'autre part, lorsqu'une nouvelle valeur de bit est mise à jour sur un vecteur d'un nœud n, elle est propagée à tous les autres vecteurs des nœuds reliés à n par un arc.

On essaie d'appliquer cette technique au programme suivant :

```
inc(i) {return i+1 ;}
dec(j) {return j-1 ;}
ide(k) {return k ;}

foo(n,f) {
   var r;
   if (n==0) {f = ide ;}
```

```
   r = (f)(n);
   return r;
}

main() {
   var x,y;
   x = lire;
   if (x>0) {y = foo(x,inc); } else (y = foo(x,dec); }
   return y;
}
```

Les règles d'analyse du flot de contrôle permettent de générer l'ensemble de contraintes suivant :

$$\{\&\texttt{inc}\} \subseteq [\![\texttt{inc}]\!]$$
$$\{\&\texttt{dec}\} \subseteq [\![\texttt{dec}]\!]$$
$$\{\&\texttt{ide}\} \subseteq [\![\texttt{ide}]\!]$$
$$[\![\texttt{ide}]\!] \subseteq [\![\texttt{f}]\!]$$
$$[\![(\texttt{f})(\texttt{n})]\!] \subseteq [\![\texttt{r}]\!]$$
$$\{\&\texttt{inc}\} \in [\![\texttt{f}]\!] \Rightarrow [\![\texttt{n}]\!] \subseteq [\![\texttt{i}]\!] \wedge [\![\texttt{i}+1]\!] \subseteq [\![(\texttt{f})(\texttt{n})]\!]$$
$$\{\&\texttt{dec}\} \in [\![\texttt{f}]\!] \Rightarrow [\![\texttt{n}]\!] \subseteq [\![\texttt{j}]\!] \wedge [\![\texttt{j}-1]\!] \subseteq [\![(\texttt{f})(\texttt{n})]\!]$$
$$\{\&\texttt{ide}\} \in [\![\texttt{f}]\!] \Rightarrow [\![\texttt{n}]\!] \subseteq [\![\texttt{k}]\!] \wedge [\![\texttt{k}]\!] \subseteq [\![(\texttt{f})(\texttt{n})]\!]$$
$$[\![\texttt{lire}]\!] \subseteq [\![\texttt{x}]\!]$$
$$[\![\texttt{foo(x,inc)}]\!] \subseteq [\![\texttt{y}]\!]$$
$$[\![\texttt{foo(x,dec)}]\!] \subseteq [\![\texttt{y}]\!]$$
$$\{\&\texttt{foo}\} \subseteq [\![\texttt{foo}]\!]$$
$$\{\&\texttt{foo}\} \in [\![\texttt{foo}]\!] \Rightarrow [\![\texttt{x}]\!] \subseteq [\![\texttt{n}]\!] \wedge [\![\texttt{inc}]\!] \subseteq [\![\texttt{f}]\!] \wedge [\![(\texttt{f})(\texttt{n})]\!] \subseteq [\![\texttt{foo(x,inc)}]\!]$$
$$\{\&\texttt{foo}\} \in [\![\texttt{foo}]\!] \Rightarrow [\![\texttt{x}]\!] \subseteq [\![\texttt{n}]\!] \wedge [\![\texttt{dec}]\!] \subseteq [\![\texttt{f}]\!] \wedge [\![(\texttt{f})(\texttt{n})]\!] \subseteq [\![\texttt{foo(x,dec)}]\!]$$

Au final, l'application de l'algorithme de recherche d'une solution à cet ensemble de contraintes donne les valeurs suivantes pour les variables non vides :

$$[\![\texttt{inc}]\!] = \{\&\texttt{inc}\}$$
$$[\![\texttt{dec}]\!] = \{\&\texttt{dec}\}$$
$$[\![\texttt{ide}]\!] = \{\&\texttt{ide}\}$$
$$[\![\texttt{f}]\!] = \{\&\texttt{inc}, \&\texttt{dec}, \&\texttt{ide}\}$$
$$[\![\texttt{foo}]\!] = \{\&\texttt{foo}\}$$

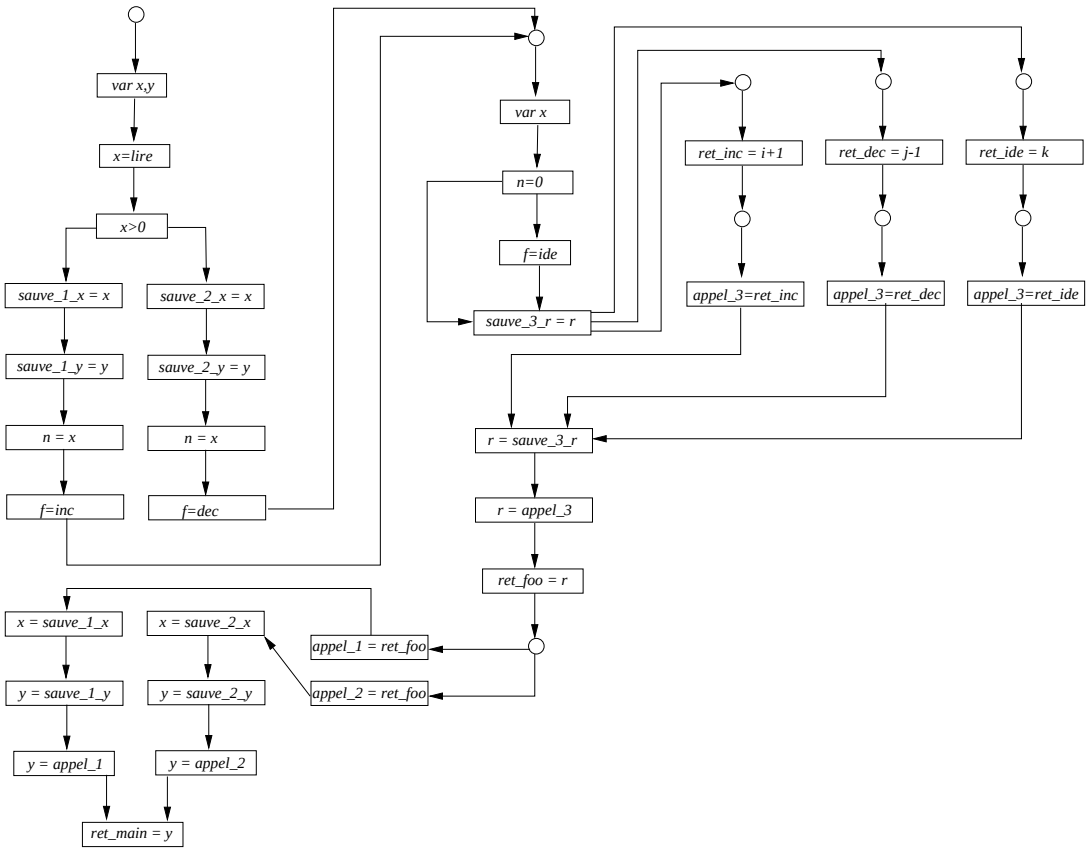

FIG. 2.25 – *Graphe de flot de contrôle interprocédural en présence de pointeurs sur fonctions*

Ce résultat, et particulièrement celui affectant la variable $[\![\texttt{f}]\!]$ permet de repérer que le paramètre `f` peut avoir la valeur des trois adresses. On peut alors construire le graphe de flot de contrôle inteprocédural de la figure 2.25.

Ce type d'analyse est un peu simplifié dans le contexte des langages orientés objet grâce à la structure offerte par la structure hiérarchique des classes. La question est de savoir quelle implantation de la méthode `m` peut être réellement invoquée par l'expression `x.m(a,b,c)`. Pour répondre à cette solution, la manière la plus simple consiste à parcourir la bibliothèque des classes et à sélectionner toute méthode désignée par `m` et dont la signature accepte les paramètres réels utilisés à l'invocation. Il est possible également, en effectuant une analyse de la hiérarchie des classes, de restreindre le champ de scrutation à la seule hiérarchie de classes recouverte par la déclaration de type de `x`.

2.9 Analyses sur pointeurs

On étend les analyses au traitement des pointeurs et de la mémoire dynamique. Concernant les analyses de programmes avec pointeurs on pourra se référer à [LH88] [CWZ90]

[Deu92] [HHN92] [HHN94] [GH96] [FGL96] [Gau97] [Ghi98] [BRS99] [RH98] et [Zha98].

Dans le langage minimal donné au paragraphe 2.2, l'instruction `malloc` permet d'allouer une cellule unique de mémoire prise dans le tas. Il n'y a pas de structuration des données possible. Néanmoins, cette hypothèse permet de présenter plus simplement des algorithmes constituant une solution acceptable au traitement général de l'analyse des pointeurs.

Une analyse de pointeurs a pour premier objectif de permettre la collecte de toutes les cibles possibles des pointeurs. Pour ce faire, toute varaible id d'un programme constitue la cible possible $\&id$, de même que toute instruction `malloc` définit la cible `malloc_i` où `i` est un identifiant unique référençant la localisation de l'instruction d'allocation. L'ensemble $Cibles$ dénote l'ensemble de cibles possibles pour les pointeurs d'un programme. L'analyse permet de construire une fonction pt qui, pour chaque variable pointeur p, retourne l'ensemble $pt(p)$ des éléments de $Cibles$ pouvant être évalué comme cibles possibles de la variable p.

L'analyse opérée sera conservative : elle fournira une sur-approximation de cet ensemble. Ce résultat peut permettre de traiter une part des problèmes d'*alias*. En particulier, il est possible de déterminer si deux variables pointeurs p et q peuvent constituer des alias en vérifiant que $pt(p) \cap pt(q) \neq \emptyset$.

2.9.1 Algorithme d'Andersen

Pour toute variable du programme id on introduit ([And94]) la variable $[\![id]\!]$ qui dénote l'ensemble des cibles possiblement pointées par la variable. On fait l'hypothèse que toute manipulation de pointeurs et d'adresse s'opère à travers les constructions suivantes :

1) $id = \texttt{malloc}$
2) $id_1 = \&id_2$
3) $id_1 = id_2$
4) $id_1 = *id_2$
5) $*id_1 = id2$
6) $id = \texttt{null}$

Il est important de noter ici que cette restriction peut paraître contraignante. Toutefois, il est possible de traiter des constructions plus complexes et de normaliser celles-ci en s'appuyant sur des variables temporaires.

Pour chacune de ces opérations sur pointeurs, on génère les contraintes suivantes :

$$
\begin{aligned}
id = \texttt{malloc}; \quad & \{\texttt{malloc\_i}\} \subseteq [\![id]\!] \\
id_1 = \&id_2; \quad & \{\&id_2\} \subseteq [\![id_1]\!] \\
id_1 = id_2; \quad & [\![id_2]\!] \subseteq [\![id_1]\!] \\
id_1 = *id_2; \quad & \&id \in [\![id_2]\!] \Rightarrow [\![id]\!] \subseteq [\![id_1]\!] \\
*id_1 = id_2; \quad & \&id \in [\![id_1]\!] \Rightarrow [\![id_2]\!] \subseteq [\![id]\!]
\end{aligned}
$$

Les deux dernières contraintes s'appliquent en principe à toutes les variables id. neanmoins on ne les applique qu'aux variables dont l'adresse $\&id$ est explicitement mentionnée dans le programme. Ainsi, si l'on considère le programme suivant :

```
var p,q,x,y,z ;
p = malloc ;
x = y ;
x = z ;
*p = z ;
p = q ;
q = &y ;
x = *p ;
p = &z;
```

alors l'algorithme génère les contraintes suivantes :

$$\texttt{malloc\_1} \subseteq [\![\texttt{p}]\!]$$
$$[\![\texttt{y}]\!] \subseteq [\![\texttt{x}]\!]$$
$$[\![\texttt{z}]\!] \subseteq [\![\texttt{x}]\!]$$
$$\&\texttt{y} \in [\![\texttt{p}]\!] \Rightarrow [\![\texttt{z}]\!] \subseteq [\![\texttt{y}]\!]$$
$$[\![\texttt{q}]\!] \subseteq [\![\texttt{p}]\!]$$
$$\{\&\texttt{y}\} \subseteq [\![\texttt{q}]\!]$$
$$\&\texttt{y} \in [\![\texttt{p}]\!] \Rightarrow [\![\texttt{y}]\!] \subseteq [\![\texttt{x}]\!]$$
$$\&\texttt{z} \in [\![\texttt{p}]\!] \Rightarrow [\![\texttt{z}]\!] \subseteq [\![\texttt{x}]\!]$$
$$\{\&\texttt{z}\} \subseteq [\![\texttt{pg}]\!]$$

L'utilisation de l'algorithme présenté dans le cadre du paragraphe 2.8.3 permet de converger vers la solution minimale suivante :

$$pt(p) = [\![p]\!] = \{\texttt{malloc\_1}, \&\texttt{y}, \&\texttt{z}\}$$
$$pt(q) = [\![q]\!] = \{\&\texttt{y}\}$$

ce qui donne un résultat précis.

2.9.2 Algorithme de Steensgaard

Il existe un algorithme moins précis mais assez communément utilisé ([Ste96]) qui se fonde sur un ensemble constitué des objets `malloc_i` et, pour chaque variable id, des objets

id et $*id$. On utilise les mêmes structures normalisées qu'en 2.9.1pour la manipulation des pointeurs. On génère pour chacune de ces contructions une contrainte d'équivalence :

$$\begin{array}{rl} id = \texttt{malloc}; & *id \sim \texttt{malloc\_i} \\ id_1 = \&id_2; & *id_1 \sim id_2 \\ id_1 = id_2; & id_1 \sim id_2 \\ id_1 = *id_2; & id_1 \sim *id_2 \\ *id_1 = id_2; & *id_1 \sim id_2 \end{array}$$

Ces contraintes introduisent une relation d'équivalence sur les différents objets. La fonction résultante peut être définie par :

$$pt(p) = \{\&id \mid *p \sim id\} \cup \{\texttt{malloc\_i} \mid *p \sim \texttt{malloc\_i}\}$$

. On se restreint, comme fait précédemment, aux instances d'adresses de type $\&id$ qui apparaissent explicitement comme telles dans les programmes. Dans le cas de programmes typés, les types permettent de limiter les cibles pour lesquelles une incompatibilité de types serait détectée.

A partir du programme précédent, l'algorithme produit les contraintes suivantes :

$*\texttt{p} \sim \texttt{malloc\_1}$
$\texttt{x} \sim \texttt{y}$
$\texttt{x} \sim \texttt{z}$
$*\texttt{p} \sim \texttt{z}$
$\texttt{p} \sim \texttt{q}$
$*\texttt{p} \sim \texttt{y}$
$\texttt{x} \sim *\texttt{p}$
$*\texttt{p} \sim \texttt{z}$

Ces contraintes construisent la relation d'équivalence illustrée par la figure 2.26. L'examen de cette relation d'équivalence implique le résultat suivant :

$$pt(p) = pt(q) = \{\texttt{malloc\_1}, \&x, \&y, \&z\}$$

Ce résultat est moins précis que celui obtenu par application de l'algorithme d'Andersen. En restreignant ce résultat aux seules adresses explicitement mentionnées dans le code du programme on obtient la solution

$$pt(p) = pt(q) = \{\texttt{malloc\_1}, \&y, \&z\}$$

qui est, s'agissant de `p`, aussi précise que celle obtenue par l'algorithme d'Andersen, mais moins précise s'agissant de `q`.

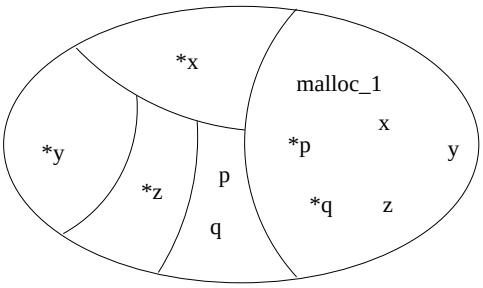

FIG. 2.26 – *Relation d'équivalence induite par les contraintes*

2.9.3 Analyse de pointeurs interprocédurale

Concernant l'analyse interprocédurale en présence de pointeurs on pourra consulter [Ban79] [Wei80] [Mye81] [LR91] [LR92] [CBC93] [LRZ93] [GH95] [HRS95] [WL95] [KRS96] [ZRL96] [Wil97].

On essaie d'étendre l'analyse au traitement des appels de fonctions, pour lesquels on fait l'hypothèse qu'ils revêtent la forme suivante : $id_1 = (id_2)(a_1, \ldots, a_n)$; où les id_i et les a_i sont des variables. De la même manière, on fait l'hypothèse que les expressions de retour (instructions `return`) ne mettent en œuvre que des variables. Ces simplifications peuvent être opérées sur la plupart des codes de manière à se ramener à un code sur lequel réaliser l'analyse.

On étend donc l'algorithme d'Andersen avec les contraintes appropriées suivantes. Toute référence à une fonction constante f génère la contrainte :

$$\{\&f\} \subseteq [\![f]\!]$$

L'appel à la fonction calculée génère la contrainte :

$$\&f \in [\![id_2]\!] \Rightarrow [\![a_1]\!] \subseteq [\![x_1]\!] \wedge \ldots \wedge [\![a_n]\!] \subseteq [\![x_n]\!] \wedge [\![id]\!] \subseteq [\![id_1]\!]$$

pour toute occurrence de la définition de fonction $f(x_1, \ldots, x_n)\{\ldots$ `return` $id;\}$

2.9.4 Exemple : analyse de pointeurs nuls

On cherche par cette analyse à détecter des déréférences de pointeurs nuls. En pratique, on cherche à s'assurer que `*p` n'est exécuté que lorsque le pointeur `p` est initialisé à une autre valeur que la valeur `null`. On considère que le programme est normalisé et que sa structure obéit à la forme suivante :

1) $id =$ `malloc`
2) $id_1 = \&id_2$
3) $id_1 = id_2$

4) $id_1 = *id_2$
5) $*id_1 = id2$
6) $id = \texttt{null}$

Le treillis de base *Null* que l'on utilise est celui décrit par la figure 2.27. dans lequel la

FIG. 2.27 – *Treillis d'affectation de pointeurs non nuls*

valeur `IN` signifie que la valeur du pointeur est initialisée, `NN` signifiant qu'elle est non nulle. On construit alors le treillis de relations $Cibles \mapsto Null$ où $Cibles$ désigne l'ensemble des cibles pointées par les variables pointeurs du programme. Ainsi, pour tout nœud v du graphe de flot de contrôle du programme, la variable de contrainte $[\![v]\!]$ dénote une table de symbole donnant l'état (inconnue, initialisée, non nulle) de chaque cible de variable en ce point. Pour les instructions de déclaration de variable on génère la contrainte

$$[\![v]\!] = [id_1 \mapsto ?, \ldots, id_n \mapsto ?]$$

Pour les nœuds correspondant aux manipulations sur pointeurs identifiées on génère les contraintes suivantes :

$$\begin{aligned}
id = \texttt{malloc};\quad & [\![v]\!] = COLLECTE(v)[id \mapsto \texttt{NN}] \\
id_1 = \&id_2;\quad & [\![v]\!] = COLLECTE(v)[id_1 \mapsto \texttt{NN}] \\
id_1 = id_2;\quad & [\![v]\!] = COLLECTE(v)[id_1 \mapsto COLLECTE(v)(id_2)] \\
id_1 = *id_2;\quad & [\![v]\!] = droite(COLLECTE(v), id_1, id_2) \\
*id_1 = id_2;\quad & [\![v]\!] = gauche(COLLECTE(v), id_1, id_2) \\
id = \texttt{null};\quad & [\![v]\!] = COLLECTE(v)[id \mapsto \texttt{IN}]
\end{aligned}$$

et pour tous les autres nœuds la contrainte est la suivante

$$[\![v]\!] = COLLECTE(v)$$

en définissant

$$COLLECTE(v) = \bigsqcup_{w \in pred(v)} [\![w]\!]$$

$$droite(\sigma,x,y) = \sigma[x \mapsto \sigma(y) \sqcup \bigsqcup_{\&p \in pt(y)} \sigma(p)]$$

$$gauche(\sigma,x,y) = \sigma[_{\&p \in pt(x)} p \mapsto \sigma(p) \sqcup \sigma(y)]$$

Après analyse, l'évaluation de `*p` est garantie sauve au point v du programme si $[\![v]\!](p) =$ `NN`. La précision de cette analyse dépend de la qualité de l'analyse des cibles pointées.

On peut considérer le programme erroné suivant :

```
var p,q,r,n ;
p = malloc ;
q = &p ;
n = null;
*q = n;
*p = r ;
```

L'algorithme d'Andersen calcul les ensembles de cibles suivants :

$$pt(\texttt{p}) = \{\texttt{malloc\_1}\}$$
$$pt(\texttt{q}) = \{\&\texttt{p}\}$$
$$pt(\texttt{r}) = \{\}$$
$$pt(\texttt{n}) = \{\}$$

Sur la base de ces informations, l'analyse de pointeurs nuls génère les contraintes suivantes :

$$[\![\texttt{varp,q,r,n;}]\!] = [\texttt{p} \mapsto ?,\texttt{q} \mapsto ?,\texttt{r} \mapsto ?,\texttt{n} \mapsto ?]$$
$$[\![\texttt{p = malloc;}]\!] = [\![\texttt{varp,q,r,n;}]\!][p \mapsto \texttt{NN}]$$
$$[\![\texttt{q = \&p;}]\!] = [\![\texttt{p = malloc;}]\!][q \mapsto \texttt{NN}]$$
$$[\![\texttt{n = null;}]\!] = [\![\texttt{q = \&p;}]\!][n \mapsto \texttt{IN}]$$
$$[\![\texttt{*q = n;}]\!] = [\![\texttt{n = null;}]\!](p) \sqcup [\![\texttt{n = null;}]\!](n)]$$
$$[\![\texttt{*p = r;}]\!] = [\![\texttt{*q = n;}]\!]$$

dont la solution minimale est

$$[\![entree]\!] = [\texttt{p} \mapsto ?,\texttt{q} \mapsto ?,\texttt{r} \mapsto ?,\texttt{n} \mapsto ?]$$
$$[\![\texttt{p} = \texttt{malloc}]\!] = [\texttt{p} \mapsto \texttt{NN},\texttt{q} \mapsto ?,\texttt{r} \mapsto ?,\texttt{n} \mapsto ?]$$
$$[\![\texttt{q} = \&\texttt{p}]\!] = [\texttt{p} \mapsto \texttt{NN},\texttt{q} \mapsto \texttt{NN},\texttt{r} \mapsto ?,\texttt{n} \mapsto ?]$$
$$[\![\texttt{n} = \texttt{null}]\!] = [\texttt{p} \mapsto \texttt{NN},\texttt{q} \mapsto \texttt{NN},\texttt{r} \mapsto ?,\texttt{n} \mapsto \texttt{IN}]$$
$$[\![*\texttt{q} = \texttt{n}]\!] = [\texttt{p} \mapsto \texttt{IN},\texttt{q} \mapsto \texttt{NN},\texttt{r} \mapsto ?,\texttt{n} \mapsto \texttt{IN}]$$
$$[\![*\texttt{p} = \texttt{r}]\!] = [\texttt{p} \mapsto \texttt{IN},\texttt{q} \mapsto \texttt{NN},\texttt{r} \mapsto ?,\texttt{n} \mapsto \texttt{IN}]$$

A l'aide de cette information, il est alors possible de constater que lorsque l'instruction `*p=r` est évaluée, la variable `p` peut contenir la valeur `null` tandis que la variable `r` n'a pas du tout été initialisée.

2.9.5 Analyse des structures

On a jusqu'à présent considéré le `tas` comme une structure de données qui serait amorphe. Il peut être analysé de façon plus détaillée en utilisant une analyse de structure. Pour ce faire, on utilise un treillis un peu plus complexe de graphes de structure du tas. Ce sont des graphes dirigés dans lesquels les nœuds représentent les cibles possibles des pointeurs. Les graphes sont ordonnés par l'inclusion de leurs ensembles d'arcs. Ainsi, $\bot$ est le graphe sans arcs et $\top$ est le graphe complètement connecté. Les cibles des pointeurs sont en fait une représentation abstraite des cellules mémoires qui peuvent être créées durant une exécution. L'existence d'un arc entre deux nœuds implique qiue la cellule mémoire source peut contenir une référence vers la cellule mémoire cible de l'arc. Plus formellement, le treillis est alors

$$2^{Cibles \times Cibles}$$

ordonné par l'inclusion des sous-ensembles.

Pour tout nœud du graphe de contrôle v, on introduit la variable de contrainte $[\![v]\!]$ qui dénote un graphe décrivant toutes les cellules mémoires possiblement utilisées à ce point du programme. Les nœuds du graphe de flot de contrôle correspondant génèrent l'ensemble de contraintes suivantes :

$$\begin{aligned}
id = \texttt{malloc};\quad & [\![v]\!] = COLLECTE(v) \downarrow id \cup \{\&\texttt{id},\texttt{malloc_i}\} \\
id_1 = \&id_2;\quad & [\![v]\!] = COLLECTE(v) \downarrow id_1 \cup \{\&\texttt{id_1},\&\texttt{_2}\} \\
id_1 = id_2;\quad & [\![v]\!] = affecte(COLLECTE(v),id_1,id_2) \\
id_1 = *id_2;\quad & [\![v]\!] = droite(COLLECTE(v),id_1,id_2) \\
*id_1 = id_2;\quad & [\![v]\!] = gauche(COLLECTE(v),id_1,id_2) \\
id = \texttt{null};\quad & [\![v]\!] = COLLECTE(v) \downarrow id
\end{aligned}$$

et pour tous les autres nœuds la contrainte

$$[\![v]\!] = COLLECTE(v)$$

en définissant

$$COLLECTE(v) = \bigcup_{w \in pred(v)} [\![w]\!]$$

$$\sigma \downarrow x = \{(s,t) \in \sigma | s \neq \&x\}$$

$$affecte(\sigma,x,y) = \sigma \downarrow x \cup \bigcup_{(\&y,t)\in\sigma} \{(\&x,t)\}$$

$$droite(\sigma,x,y) = \sigma \downarrow x \cup \bigcup_{(\&y,s),(s,t)\in\sigma} \{(\&x,t)\}$$

$$gauche(\sigma,x,y) = \bigcup_{(\&x,s),(\&y,t)\in\sigma} \sigma \downarrow s \cup \{(s,t)\}$$

On peut considérer alors le programme suivant:

```
var x,y,n,p,q ;
x = mallox ;
y = malloc ;
*x = null;
*y = y;
n = lire ;
while (n>0) {
     p = malloc ;
     q = malloc ;
     *p = x ;
     *q = y ;
     x = p;
     y = q;
     n = n-1;
}
```

L'analyse des contraintes produit, au point situé derrière la structure de boucle `while`, les graphes de structures de la figures 2.28. Ceux-ci permettent d'établir que les ensembles de cellules mémoires qui peuvent être pointées par `x` et `y` sont toujours disjoints. Néanmoins, ces graphes ne permettent pas de dire beaucoup plus que cela.

Il faut toutefois noter que l'analyse effectue un calcul des relations de pointage tenant compte du flot de contrôle. Pour chaque point du programme, cette relation est donnée par

$$pt(p) = \{t | (\&p,t) \in [\![v]\!]\}$$

Cette analyse est plus précise que l'algorithme d'Andersen, mais également plus coûteuse à mettre en œuvre. Si l'on considère le programme :

```
x = &y ;
```

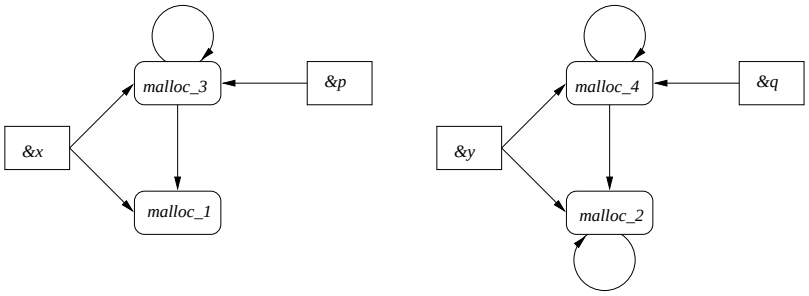

FIG. 2.28 – *Graphes de structure du* tas

```
x = &z ;
```

l'algorithme d'Andersen calculera, à la fin de la séquence d'instructions, une valeur de $pt(\mathtt{x}) = \{\&\mathtt{y},\&\mathtt{z}\}$ tandis que l'analyse de structures (tenant compte de la structure du flot de contrôle) calculera une valeur $pt(\mathtt{x}) = \{\&\mathtt{z}\}$.

2.10 Conclusion

Ce chapitre a décrit un ensemble de techniques et d'outils de base intervenant dans l'analyse statique des programmes. Les problématiques envisagées, dès lors qu'il devient nécessaire d'analyser des codes, font, dans la pratique, appel à ces différentes techniques. On peut toutefois parfois en trouver des variantes ou des extensions, mais le fond des analyses opérées demeure identique.

Il conviendrait cependant de pouvoir établir la correction des analyses proposées. Ceci requiert de disposer d'une sémantique formelle du langage de programmation servant à rédiger les codes. Les preuves, formelles et complètes, des analyses envisagées constituent un exercice laborieux. Dans la pratique, on se contentera souvent d'apporter des arguments suffisamment convaincants concernant la correction des techniques d'analyse proposées.

Chapitre 3

Quelques outils pour l'analyse statique

3.1 Les outils d'analyse

3.1.1 Analyse statique : généralités

L'abstraction sémantique, quoi qu'il en soit, est une nécessité dans le cadre d'une exploitation pratique de l'analyse statique, laquelle concrétise les cas où le choix du niveau d'abstraction sémantique rend cette dernière « machinable ».

L'analyse statique s'avère utile chaque fois que l'interaction humaine (conduite de preuve, instrumentation de code, test) doit être limitée pour des raisons de coût. Mais une analyse statique doit pouvoir être menée à bien dans un temps raisonnable et avec les moyens informatiques courants.

Les outils d'analyse statique ne peuvent donc prendre en considération que des question limitées et évitent d'aborder le problème général consistant à analyser statiquement une propriété quelconque pour un programme quelconque, ce qui est indécidable, notamment pour les langages de programmation généraux tels que C. Les limitations portent sur les familles de programmes et les catégories de propriétés analysables.

Il s'agit alors d'obtenir un bon rapport performances/coût, en choisissant judicieusement l'approximation sémantique calculée par l'outil d'analyse statique de façon à obtenir le plus souvent possible une information utile à un coût acceptable. Mais cette approximation produit aussi des incertitudes, qui peuvent ou non être acceptables. Par exemple :

- dans une analyse de type flots de données (par exemple élimination des tests de bornes de tableaux dans un code objet), l'incertitude conduit à ne pas mettre en œuvre toutes les optimisations possibles, ce qui est acceptable si l'optimisation des programmes est efficace en moyenne,
- lorsqu'on examine des programmes du point de vue de leur typage, certains programmes sans erreur à l'exécution peuvent être rejetés par le compilateur, ce qui est acceptable s'il est possible de les écrire autrement (éventuellement au prix d'une certaine perte d'efficacité à l'exécution),

- dans une analyse de temps maximal d'exécution (« *Worst Case Execution Time* », WCET), une analyse trop imprécise surestime les temps d'exécution, ce qui est acceptable si l'estimation reste dans les limites fixées pour l'application (un exemple est donné par `http://www.absint.com`, traité dans le cadre du projet DAEDALUS),
- en ce qui concerne l'analyse statique d'erreurs à l'exécution (« *Run-Time Errors* », RTE), une analyse trop imprécise conduit à des fausses alarmes (une erreur potentielle est signalée comme possible, sans que cela soit certain), ce qui est acceptable dans les applications non critiques pour lesquelles on recherche seulement à détecter les erreurs les plus grossières (un exemple est donné par `http://www.polyspace.com`). La détection de fausses alarmes peut par contre devenir inacceptable dès lors qu'on traite des applications critiques. Un exemple en est l'outil d'analyse statique développé dans le cadre du projet RNTL ASTRÉE.

En fin de compte, les outils d'analyse statique automatiques et utilisables pratiquement (c'est-à-dire terminant toujours et utilisant des ressources informatiques finies) :

- limitent plus ou moins la classe des programmes pris en considération,
- limitent plus ou moins la famille des propriétés prises en considération,
- admettent plus ou moins de cas d'incertitude,

toutes les variantes étant *a priori* possibles.

3.1.2 Classification des outils

Un outil d'analyse statique de programmes a pour but de démontrer qu'un programme satisfait certaines propriétés lors de son exécution quel que soit l'environnement d'exécution.

Une telle démonstration procède en deux temps:

- on détermine un invariant inductif, valide pour tous les comportements possibles du programme dans les environnements d'exécution considérés ;
- on démontre que cet invariant inductif implique la propriété à démontrer (par exemple : l'absence d'erreurs à l'exécution).

La difficulté réside dans la détermination de l'invariant inductif, Il existe essentiellement trois façons de faire (qui sont souvent combinées):

- demander à l'utilisateur (méthodes déductives) ;
- calculer exactement par énumération de tous les cas possibles (vérification de modèles « *model-checking* ») ;
- approcher les cas possibles (analyse statique par sous-approximation ou sur-approximation).

Puisqu'on se place dans le cadre de traitements informatiques, une limitation importante par rapport à un raisonnement mathématique général est que propriétés et abstractions des comportements des programmes examinés doivent avoir une représentation finie.

Dans tous les cas, les propriétés à démontrer doivent être représentées en machine de manière finie, ce qui limite considérablement les possibilités par rapport à un raisonnement mathématique à la main (de toutes façons impossible à cause de la complexité du travail).

Il existe essentiellement deux stratégies pour déterminer quelle classe d'invariants est prise en considération dans un outil de vérification :

- soit on considère une classe d'invariants très générale (par exemple les prédicats du premier ordre comme dans les outils d'assistance à la preuve) ;
- soit on considère une classe d'invariants plus restreinte (par exemple les formules booléennes pour le « *model-checking* » ou les intervalles d'entiers pour l'analyse statique pour lesquels il existe des algorithmes de manipulation plus ou moins complexes et efficaces).

Selon le cas, les outils d'analyses statique se classent dans un parmi deux types principaux :

- Les outils d'analyse généraux, dont l'objectif est de traiter tout programme d'un langage de programmation général (tels C, ADA, etc.) par analyse de propriétés d'intérêt général, avec risque d'apparition d'incertitude dans un petit nombre de cas. L'avantage est que l'outil d'analyse est utilisable pour un grand nombre d'applications, appartenant à des domaines divers, ce qui permet de partager les coûts de conception. Les outils d'analyse généraux sont très utiles quand ils permettent de trouver des erreurs certaines. Pour certains domaines d'application, toutefois (comme les RTE mentionnées plus haut, ou le domaine des logiciels embarqués), les cas d'incertitude sont inacceptables. Or, La précision de l'outil d'analyse ne peut être indéfiniment améliorée, compte tenu de la diversité des applications prises en considération. Un problème supplémentaire posé par ces outils généraux et par le fait qu'ils prennent en considération une classe d'invariants très générale est la représentation et la manipulation de ces derniers, qui requiert en général une assistance humaine ;
- Les outils d'analyse spécifiques, dont l'objectif est de traiter des programmes appartenant à une classe restreinte, par analyse de propriétés utiles pour cette famille d'applications particulière, le but étant d'éviter si possible toute fausse alarme. Pour ce faire, l'outil d'analyse peut exploiter des connaissances spécifiques aux programmes de la classe considérée et de leurs environnements d'exécution. Une démarche communément appliquée pour démontrer une propriété d'un programme consiste à définir une propriété plus forte qui soit démontrable par induction. La conception de l'outil d'analyse peut alors être faite de telle façon que l'abstraction se fasse sans perte d'information pour ces propriétés inductives. L'effort d'abstraction repose, dans le cas de ces outils, sur le concepteur de l'outil d'analyse, alors que dans le domaine de la preuve ou de la vérification de modèles de logiciels il est à la charge de l'usager, qui doit

fournir les hypothèses d'induction ou le modèle adéquat. Un outil d'analyse statique spécialisé est capable d'instancier seul les invariants dans les classes qu'il connaît ; si c'est dernières sont suffisamment riches, on peut envisager d'éliminer toute fausse alarme.

Les outils d'analyse statique sont, dans tous les cas, des outils complexes, ce qui explique que peu d'entre eux soient utilisables industriellement.

En termes d'interprétation abstraite, on peut distinguer:

- les analyseurs statiques faisant des abstractions finitaires (permettant donc une vérification exhaustive du modèle abstrait) ou « *model checkers* », qui laissent le plus souvent à l'utilisateur le soin de proposer un modèle fini de son programme ;
- les analyseurs statiques utilisant des abstractions infinitaires et des techniques permettant d'accélérer la convergence de l'interprétation abstraite [CC77].

3.1.3 Inventaire (non exhaustif)

On peut distinguer les analyseurs pour lesquels le modèle fini booléen du programme doit être fourni par l'utilisateur de ceux offrant une aide à l'abstraction finie, voire cherchant à automatiser cette aide, soit au risque de fausses alarmes, soit au risque de non-terminaison de l'analyse statique.

L'avantage des abstractions finies est que l'exploration de l'espace d'états abstrait peut être exhaustive et facile à mettre en oeuvre (nombre d'algorithmes et d'implémentations sont disponibles). Elle limite cependant par avance la famille des propriétés démontrables et peut rendre impossible à obtenir la précision requise (voir [CC92a]). Les solutions possibles sont :

- de demander à l'utilisateur de fournir l'abstraction (comme souvent en « *abstract software model checking* ») (*SMV*, *SPIN*) ;
- d'utiliser une analyse préalable pour calculer l'abstraction finie, ce qui peut éventuellement conduire à des incertitudes (*PathFinder*, *FeaVer*) ;
- de calculer une suite d'approximations finies de plus en plus précises en les essayant successivement (ce qui revient en fin de compte à explorer l'espace d'états concret et donc peut ne pas se terminer ou exploser très rapidement) (*SLAM*, *BLAST*) ;
- ou enfin d'utiliser des abstractions infinitaires, dont il faut maîtriser la divergence potentielle [CC77] (*ESC*, *CAVEAT*, *CANVAS*, *FLUCTUAT*, etc.).

On trouve des outils basés sur ces trois approches.

- L'analyseur *ESP* de *Microsoft* [DLS02] non disponible commercialement, permet de démontrer des propriétés temporelles simples spécifiées par des automates finis. Un exemple d'application en a été la vérification que, dans le compilateur gcc, toute lecture d'un fichier est précédée d'une ouverture de ce fichier. *ESP* utilise un algorithme

appelé *simulation de propriété*, dérivant de l'analyse de flot de données par un regroupement spécifique des états symboliques d'exécution et de l'état de la propriété. L'instance de l'algorithme décrite s'exécute en temps polynomial.

- L'analyseur *UNO* de *Bell Labs* [Hol02], qualifié de « relativement simple » par ses auteurs, est dédié à la recherche dans la programmation C d'erreurs de non initialisation, d'usage de pointeur nul et d'index en dehors des bornes. Il possède des capacités d'extension autorisant l'introduction de propriétés définies par l'usager. Les abstractions employées incluent l'analyse de flot DEF-USE pour le défaut d'initialisation ou la propagation symbolique de contraintes syntaxiques du programme pour les indices de tableaux.
- L'analyseur *BANE* de l'Université de Berkeley [AFFS98] est principalement utilisé pour l'analyse de flot de données et le typage. Il est basé sur une seule abstraction prédéfinie (dite « *constraint-based* » ou « *set based* », voir les explications de [CC95]).
- LESAR [Rat92] est un vérificateur de modèles universitaire développé à l'IMAG ; il permet de vérifier des propriétés de sûreté sur des spécifications booléennes écrites en Lustre [HCRP91]. Le programme d'entrée à vérifier combine en général la spécification proprement dite avec un autre programme Lustre spécifiant la propriété à vérifier. L'outil vérifie ensuite que la sortie booléenne de l'ensemble est toujours vraie. Les assertions sont considérées comme des hypothèses sur l'environnement du programme (et donc supposées toujours vérifiées).
- *SMV* est un vérificateur exhaustif de modèles finis (« *model-checker* ») conçu à l'Université de Carnegie Mellon [CGL93]. Le modèle est un système de transition fini et la spécification est donnée en logique temporelle. *SMV* a été principalement utilisé pour la vérification de matériel et celle de protocoles. La difficulté est de construire (manuellement) un modèle fini à partir du texte du programme.
- L'analyseur *SPIN* de *Bell Labs* est un vérificateur exhaustif de modèles finis (« *model checker* ») décrits dans un langage spécifique : *PROMELA*, et dont les propriétés sont décrites en logique temporelle *LTL* [Hol03]. Par rapport à *SVM*, le langage *PROMELA* est un peu plus expressif puisqu'il permet de décrire un système de processus parallèles communiquant par variables partagées et/ou par messages synchrones ou asynchrones avec des données finies. De plus le modèle abstrait est calculé automatiquement (quelquefois avec l'aide de l'utilisateur). Cependant, seuls les protocoles d'échanges de messages sont modélisés ;les aspects calcul ne peuvent être pris en considération (SPIN est conçu en premier lieu pour la vérification de protocoles de communication). Le modèle doit être fini et, si la taille de celui-ci excède ses capacité de traitement, *SPIN* offre la solution de recourir à la simulation aléatoire.

Au dessus de ces différents outils, et en utilisant certains, citons l'outil *BANDERA* de l'Université du Kansas [DHJ+01].

On peut considérer BANDERA comme un atelier de vérification ; la démarche appliquée consiste à prendre en entrée des programmes source en Java, qui sont traduits en langage intermédiaire et auxquels on applique, en fonction de la propriété que l'on désire vérifier,

différents algorithmes d'analyse statique et d'abstraction : on construit donc en principe un modèle du programme source différent pour chaque propriété à vérifier ; on procède enfin à la traduction du programme résultant dans le langage d'entrée *TRANS* de *SMV* ou *PROMELA* de *SPIN* (machines d'états finis dans les deux cas). L'analyseur utilise ensuite l'un de ces vérificateurs exhaustifs de modèles (« *model-checkers* ») pour conduire la vérification. Les propriétés à vérifier sont exprimées dans un « langage naturel stylisé » et les contre-exemples éventuellement exhibés sont aussi reformulés dans les termes de la spécification de départ. La démarche vise en fin de compte :

1. à construire une abstraction optimale du programme source, afin de pouvoir le traiter de façon efficace par vérification de modèle ;
2. à éviter à l'usager d'avoir à se préoccuper des aspects liés à la modélisation (et en particulier du formalisme utilisé).

- *Prover Technology* (`http://www.prover.com`) produit plusieurs outils commerciaux : *Prover CL*, un outil de vérification de modèles pour les logiques combinatoires et *Prover SL*, un outil de vérification de modèles pour les logiques séquentielles. Ils ont pour base l'algorithme de Stålmarck [SS00], procédure de preuve en logique propositionnelle. Une version spécifique de cet outil, le *Design Verifier*, s'interface notamment en tant que *Plugin* [Est03] avec l'environnement *SCADE Suite* utilisé en particulier par Airbus pour le développement d'applications avioniques.
- L'analyseur Java *PathFinder* de la NASA *Ames* [BHPV00] analyse du « *bytecode* » Java. La première version *JPF1* procède par traduction en *PROMELA* puis vérification exhaustive du modèle fini par *SPIN*. La deuxième version *JPF* permet d'utiliser l'abstraction de prédicats (un moyen simple de spécifier des abstractions, dont on trouvera une introduction dans [Cou03]) fournis par l'utilisateur. L'analyseur utilise le vérificateur de formules logiques *SVC*. *JFP* permet de faire l'analyse statique de programme par l'intermédiaire de *BANDERA* ainsi que l'analyse dynamique de programmes.
- L'analyseur *FeaVer* de *Bell Labs* permet à l'utilisateur de spécifier une abstraction pour extraire un modèle fini d'un programme C afin de vérifier des propriétés temporelles spécifiées par des formules de logique temporelle ou des automates finis [HS99]. L'extraction du modèle est réalisée par l'outil *Modex*. La vérification exhaustive de modèle est faite par *SPIN*.
- L'analyseur *SLAM* de *Microsoft* [BNR03] non disponible commercialement, permet de démontrer des propriétés temporelles simples spécifiées par des automates finis. Il est utilisé en interne pour la vérification de contrôleurs (« *drivers* ») de la famille de systèmes d'exploitation *Windows*, pour vérifier par exemple que seules les séquences valides d'opérations sont possibles (comme tout verrou bloqué doit être relâché). Il est basé sur l'abstraction de prédicats avec codage booléen et raffinement. Il est donc possible que l'analyse ne termine pas. Le temps de calcul de l'analyse est essentiellement exponentiel par rapport au nombre de variables à prendre en compte dans le programme.

- L'analyseur *BLAST* de l'Université de Berkeley [HJRS02] a pour objectif de vérifier qu'un point du programme (correspondant en général à une erreur à l'exécution) est inaccessible. La technologie est très comparable à celle de *SLAM*.
- L'analyseur *ESC* de *Compaq* [LN98] est utilisable pour prouver l'absence d'erreurs à l'exécution de programmes Modula ou Java. Basé sur l'utilisation du démonstrateur de théorèmes *simplify*, les preuves requièrent des annotations de l'utilisateur qui doit fournir des invariants de boucles et les interfaces de modules. Une des difficultés est de comprendre en cas d'échec pourquoi le démonstrateur n'a pas été capable de faire la preuve, sachant qu'il n'y a pas d'interaction aisée.
- L'outil *CAVEAT* est basé sur la logique de Hoare. Il permet d'effectuer des preuves de type RTE et des preuves de propriétés fonctionnelles; il dispose de moyens d'interaction pour aider à la terminaison de la démonstration.
- L'analyseur *TVLA* de l'Université de Tel Aviv [LAS02], [LAMS04] procède par traduction de la sémantique du programme et de sa spécification en formules de la logique du premier ordre qui sont ensuite abstraites par une abstraction standard à 3 valeurs (vrai, faux, inconnu).
- L'analyseur *CANVAS* d'IBM [RWFS02] permet de spécifier des interfaces de modules Java™ et JavaBeans™ dans le langage de spécification *EASL* et de vérifier que le client utilise le module conformément à sa spécification. La vérification est faite par l'analyseur statique *TVLA*.
- L'*analyseur statique d'erreurs à l'exécution* de *Polyspace* Technologies est largement diffusé commercialement [RS00]. Il est basé sur la notion de « propriété approximative », destinée à réduire la complexité des problèmes traités; le coût de l'analyse reste néanmoins important comme le nombre des fausses alarmes engendré par la méthode. Il est disponible pour C, C++ et Ada.
- L'*analyseur de temps maximal d'exécution* (*WCET*) *aiT* [FHL+01], [FH04] d'*Absint* disponible commercialement, qui fait l'analyse au niveau du code objet et peut donc être très précis (en prenant en compte des comportements possibles des caches et des pipelines) mais donc forcément lié à un certain nombre de modèles de processeurs. Il a été testé avec succès par Airbus France sur des logiciels avioniques [TSH+03].
- L'*analyseur statique d'erreurs à l'exécution* développé dans le cadre du projet RNTL ASTRÉE est spécialisé pour les programmes synchrones écrits en C [BCC+03], [Mau04]. Il a été testé avec succès par Airbus France sur des logiciels avioniques. Il utilise des abstractions spécialisées prenant en compte la théorie du contrôle et du filtrage, ce qui permet d'obtenir une grande efficacité et une grande précision.
- L'analyseur *FLUCTUAT* du *CEA* [Mar02] est spécialisé dans l'analyse de la précision des calculs numériques des programmes. L'analyse prend en compte les erreurs d'arrondi dans les calculs flottants par rapport au opérations réelles et trace leur influence sur la suite des calculs. Ceci permet de déterminer en chaque point du programme l'origine des imprécisions et l'influence respective des différentes sources possibles.

3.2 Conclusion

L'inventaire sommaire de la section précédente, bien que loin d'être exhaustif, permet de faire un certain nombre de constatations :

- l'éventail des outils développés est large, quoique peu d'entre eux aient atteint le stade industriel.
- les développements, particulièrement dans le domaine industriel, concernent principalement la vérification de modèles (*model checking*), car c'est la méthode permettant de viser un fonctionnement le plus automatique possible pour une certaine gamme de spécifications et de propriétés. Dans le domaine de la preuve inductive, on montre qu'il existe [Cou00b] une abstraction finitaire, permettant de construire un outil d'analyse (automatique) sans fausse alarme pour une famille réduite à un programme et à une propriété ; si la famille n'est pas finie (mais suffisamment bien définie pour permettre une caractérisation des propriétés évitant les fausses alarmes), l'abstraction est forcément infinitaire [CC92a], et produit des outils d'analyse du type de ceux développés par exemple dans le cadre du projet RNTL ASTRÉE [BCC+02], [BCC+03], [Mau04].
- il existe une grande variété d'outils dédiés à des domaines d'application spécifiques ou à des types de propriétés particuliers.
- aucun outil d'analyse statique n'existe à l'heure actuelle à notre connaissance qui soit spécifiquement dédié à la vérification d'IHM. Celle-ci, toutefois, ne présente pas de difficulté spécifique (notamment quant à l'aspect arithmétique), et les outils tels que LESAR, SPIN ou SMV lui sont applicables.

Bibliographie

[AC99] G. Filé A. Cortesi. Static analysis. In *Proceedings of 6th International Symposium (SAS'99)*, Venice,Italy, 1999.

[AFFS98] A. Aiken, M. Fähndrich, J.S. Foster, and Z. Su. Constructing Type- and Constraint-Based Program Analyses. In A. Leroy and A. Ohori, editors, *Second International Workshop on Types in Compilation (TIC98)*, volume 1473 of *Lecture Notes in Computer Science*, pages 78–96, Kyoto, Japan, March 1998. Springer-Verlag.

[Age94] Ole Agesen. Constraint-based type inference and parametric polymorphism. In *First International Static Analysis Symposium*, pages 78–100, 1994.

[Age95] Ole Agesen. *Concrete Type Inference : Delivering Object-Oriented Applications.* PhD thesis, Stanford University, 1995.

[And94] Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language.* Report 94/19, DIKU University of Copenhagen, May 1994.

[ASU86] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques and Tools.* Addison Wesley, interéditions edition, 1986.

[Bac97] David F. Bacon. *Fast and effective optimization of statically typed object-oriented languages.* PhD thesis, University of California, Berkeley, 1997.

[Ban79] John P. Banning. An efficient way to find the side effects of procedure calls and the aliases of variables. In *6th POPL*, pages 29–41. ACM, 1979.

[BCC+02] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. Design and Implementation of a Special-Purpose Static Program Analyzer for Safety-Critical Real-Time Embedded Software, invited chapter. In T. Mogensen, D.A. Schmidt, and I.H. Sudborough, editors, *The Essence of Computation: Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones*, volume 2566 of *Lecture Notes in Computer Science*, pages 85–108. Springer-Verlag, 2002.

[BCC+03] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A Static Analyzer for Large Safety-Critical Software. In *PLDI 2003–ACM SIGPLAN SIGSOFT Conference on Programming Language Design and Implementation*, pages 196–207, San Diego, California, USA, June 7–14 2003. ACM Press.

[BHPV00] G. Brat, K. Havelund, S. Park, and W. Visser. Java PathFinder – A second generation of a Java model checker. In *Proceedings of the Workshop on Advances in Verification*, Chicago, Illinois, July 2000.

[BNR03] T. Ball, M. Naik, and S. Rajamani. From Symptom to Cause: Localizing Errors in Counterexample Traces. In *Conference Record of the 30th ACM SIGPLAN-SIGART Symposium on Principles of Programming Languages (POPL)*, volume 38 issue 1 of *ACM SIGPLAN Notices*, pages 97–105, New Orleans, Louisiana, January 2003. ACM Press.

[BRS99] M. Benedikt, T. Reps, and M. Sagiv. A decidable logic for describing linked data structures. In *European Symposium On Programming (ESOP'99)*, number 1576 in Lecture Notes in Computer Science, pages 2–19. Springer-Verlag, 1999.

[Cas97] Giuseppe Castagna. *Object-oriented programming, a unified foundation.* Progress in theoretical computer science. Birkhäuser, 1997.

[CBC93] J.D. Choi, M. Burke, and P. Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *20th POPL*, pages 232–245, 1993.

[CC76] P. Cousot and R. Cousot. Static Determination of Dynamic Properties of Programs. In B. Robinet, editor, *Proceedings of the 2nd International Symposium on Programming*, pages 106–130, Paris, April 13–15 1976. Dunod.

[CC77] P. Cousot and R. Cousot. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Conference Record of the 4th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 238–252, Los Angeles, CA, 1977. ACM Press.

[CC92a] P. Cousot and R. Cousot. Comparing the Galois Connection and Widening/Narrowing Approaches to Abstract Interpretation, invited paper. In M. Bruynooghe and M. Wirsing, editors, *Programming Language Implementation and Logic Programming, Proceedings of the 4th International Symposium, PLILP'92*, volume 631 of *Lecture Notes in Computer Science*, pages 269–295, Louvain, Belgium, August 1992. Springer-Verlag.

[CC92b] P. Cousot and R. Cousot. Inductive Definitions, Semantics and Abstract Interpretation. In *Conference Record of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 83–94, Albuquerque, NM, August 1992. ACM Press.

[CC95] P. Cousot and R. Cousot. Formal Language, Grammar and Set-Constraint-Based Program Analysis by Abstract Interpretation. In *Conference Record of FPCA'95 SIGPLAN/SIGARCH/WG2.8 Conference on Functional Programming and Computer Architecture*, pages 170–181, La Jolla, CA, June 25–28 1995. ACM Press.

[CC99] P. Cousot and R. Cousot. Refining Model Checking by Abstract Interpretation. *Automated Software Engineering Journal, special issue on Automated Software Analysis*, 6(1):69–95, 1999.

[CC00] P. Cousot and R. Cousot. Temporal Abstract Interpretation. In *Conference Record of the 27th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Programming Languages (POPL)*, pages 12–25, Boston, MA, January 19–21 2000. ACM Press.

[CC02] P. Cousot and R. Cousot. On Abstraction in Software Verification. In E. Brinksma and K.G. Larsen, editors, *Proceedings of the l4th International Conference on Computer-Aided Verification (CAV'2002)*, volume 2404 of *Lecture Notes in Computer Science*, pages 37–56, Copenhague, July 27–31 2002. Springer-Verlag.

[CCZ97] S. Collin, D. Colnet, and O. Zendra. Type inference for late binding : the small eiffel compiler. In *Joint Modular Languages Conference*, volume 1204 of *LNCS*, pages 67–81. Springer-Verlag, 1997.

[CGL93] E. Clarke, O. Grumberg, and D. Long. Verification tools for finite-state concurrent systems. In J.W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *A Decade of Concurrency - Reflections and Perspectives, Proceedings of the REX School-Symposium*, volume 803 of *Lecture Notes in Computer Science*, Noordwijkerhout, The Netherlands, June 1-4 1993. Springer-Verlag.

[Cou97] P. Cousot. Types as Abstract Interpretations, invited paper. In *Conference Record of the 24th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Programming Languages (POPL)*, pages 316–331, Paris, January 15–17 1997. ACM Press.

[Cou00a] P. Cousot. Interprétation abstraite. *Technique et Science Informatique*, 19(1–2–3):155–164, January 2000.

[Cou00b] P. Cousot. Partial Completeness of Abstract Fixpoint Checking, invited paper. In B.Y. Choueiry and T. Walsh, editors, *Proceedings of the 4th International Symposium on Abstraction, Reformulations and Approximation, SARA'2000*, volume 1864 of *Lecture Notes in Artificial Intelligence*, pages 1–25, Horseshoe Bay, TX, USA, July 26-29 2000. Springer-Verlag.

[Cou02] P. Cousot. Constructive Design of a Hierarchy of Semantics of a Transition System by Abstract Interpretation. *Theoretical Computer Science*, 277(12):47–103, April 2002.

[Cou03] P. Cousot. Verification by Abstract Interpretation. In *Proceedings of the International Symposium on Verification Theory & Practice Honoring Zohar Manna's 64th Birthday*, volume 2772 of *Lecture Notes in Computer Science*, pages 243–268, Taormina, Italy, June 29–July 4 2003. Springer-Verlag.

[CWZ90] David R. Chase, Mark Wegman, and F.K. Zadeck. Analysis of pointers and structures. In *17th POPL*, pages 296–310. ACM, 1990.

[Deu92] Alain Deutsch. A storeless model of aliasing and its abstractions using finite representations of right-regular equivalence relations. In IEEE, editor, *International Conference on Computer Languages*, pages 2–13, 1992.

[DHJ+01] M.B. Dwyer, J. Hatcliff, R. Joehanes, S. Laubach, C.S. Pasareanu, R.W. Visser, and H. Zheng. Tool-Supported Program Abstraction for Finite-State Verifica-

tion. In *Proceedings of the 23rd International Conference on Software Engineering (ICSE01)*, pages 177–187, Toronto, Canada, May, 12–13 2001.

[DLS02] M. Das, S. Lerner, and M. Seigle. ESP: Path-Sensitive Program Verification in Polynomial Time. In *PLDI'02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, Berlin, Germany, June 2002.

[DMM96] Amer Diwan, J. Eliot B. Moss, and Kathryn S. McKinley. Simple and effective analysis of typed object-oriented programs. In *Proc. OOPSLA'96*, volume 31(10) of *SIGPLAN Notices*, pages 292–305. ACM Press, 1996.

[DP90] B.A. Davey and H.A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, 1990.

[Dwy95] Matthew B. Dwyer. *Data Flow Analysis for Verifying Correctness Properties of Concurrent Programs*. PhD thesis, University of Massachusetts, Amherst, MA, September 1995.

[Est03] Esterel Technologies. *SCADE Suite : Design Verifier User Manual*, March 2003.

[FGL96] Pascal Fradet, Ronan Gaugne, and Daniel LeMetayer. An inference algorithm for the static verification of pointer manipulation. Rapport de Recherche RR-2895, IRISA, Rennes, Campus Universitaire de Beaulieu, Mai 1996.

[FH04] C. Ferdinand and R. Heckmann. aiT: Worst-Case Execution Time Prediction by Static Program Analysis. In R. Jacquart, editor, *Building the Information Society. IFIP 18th World Computer Congress, Topical Sessions*, pages 377–383, Toulouse, France, August, 22-27 2004. Kluwer Academic Publishers.

[FHL+01] C. Ferdinand, R. Heckmann, M. Langenbach, F. Martin, M. Schmidt, H. Theiling, S. Thesing, and R. Wilhelm. Reliable and Precise WCET Determination for a Real-Life Processor. In T.A. Henzinger and C. M. Kirsch, editors, *Embedded Software First International Workshop, EMSOFT 2001*, volume 2211 of *Lecture Notes in Computer Science*, pages 469–485, Tahoe City, CA, USA, October, 8–10 2001. Springer-Verlag.

[Gau97] Ronan Gaugne. *Techniques d'analyse statique pour l'aide à la mise au point de programmes avec manipulation explicite de pointeurs*. PhD thesis, Université de Rennes 1, Octobre 1997.

[GH95] Rakesh Ghiya and Laurie J. Hendren. Connection analysis: a practical interprocedural heap analysis for c. In C.H. Huang, P. Sadayappan, U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *8th Workshop on Languages and Compilers for Parallel Computing*, number 1033 in LNCS. Springer-Verlag, 1995.

[GH96] Rakesh Ghiya and Laurie J. Hendren. Is it a tree, a dag, or a cyclic graph? a shape analysis for heap-directed pointers in c. In *23rd POPL*, pages 1–15. ACM, 1996.

[Ghi98] Rakesh Ghiya. *Putting Pointer Analysis to Work*. PhD thesis, School of Computer Science, McGill University, January 1998.

[Hav94] Paul Havlak. *Interprocedural Symbolic Analysis*. PhD thesis, CRPC,Rice University, May 1994.

[Hav97] Paul Havlak. Nesting of reducible and irreducible loops. *ACM Transactions on Programming Languages and Systems*, 19(4):557–567, July 1997.

[HCRP91] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. Programmation et vérification de systèmes réactifs à l'aide du langage flot de données synchrone LUSTRE. *Technique et Science Informatique*, 10(2):139–158, 1991.

[HDT87] Susan Horwitz, Alan Demers, and Tim Teitelbaum. An efficient general iterative algorithm for dataflow analysis. *Acta Informatica*, 24(6):679–694, 1987.

[HHN92] Laurie J. Hendren, Joseph Hummel, and Alexandru Nicolau. Abstractions for recursive pointer data structures: Improving the analysis and transformation of imperative programs. In *PLDI*, pages 249–260. ACM, 1992.

[HHN94] Laurie J. Hendren, Joseph Hummel, and Alexandru Nicolau. A general data dependence test, pointer-based data structures. In *PLDI*, pages 218–229. ACM, 1994.

[HJRS02] T.A. Henzinger, R. Jhala, Majumdar R., and G. Sutre. Lazy Abstraction. In *Proceedings of the 29th Annual Symposium on Principles of Programming Languages (POPL)*, pages 58–70. ACM Press, 2002.

[Hol02] G. J. Holzmann. UNO: Static Source Code Checking for User-Defined Properties. In *Proceedings of the 6th World Conference on Integrated Design & Process Technology (IDPT 2002)*, Pasadena, CA, USA, June 23–28 2002.

[Hol03] G. J. Holzmann. *The Spin Model Checker – Primer and Reference Manual*. Addison-Wesley, August 2003. ISBN 0-32122-862-6.

[HRS95] Susan Horwitz, Thomas Reps, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *22nd POPL*, pages 49–61. ACM, 1995.

[HS99] G. J. Holzmann and M. H. Smith. Software Model Checking, Extracting Verification Models from the Source Code. In W. Jianping, S. T. Chanson, and G. Qiang, editors, *Formal Methods for Protocol Engineering and Distributed Systems, (Conference Proceedings FORTE/PSTV99)*, pages 481–497, Beijing, China, October 5–8 1999. Kluwer Academic Publishers.

[HU73] M.S. Hecht and J.D. Ullman. Analysis of a simple algorithm for global data flow problems. In ACM, editor, *POPL*, pages 207–217, 1973.

[HU74] M.S. Hecht and J.D. Ullman. Characterizations of reducible flow graphs. *JACM*, 1974.

[JC97] Johan Janssen and Henk Corporaal. Making graphs reducible with controlled node splitting. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 19(0.6):1031–1052, November 1997.

[Ken76] K. Kennedy. A comparison of two algorithms for data flow analysis. *SIAM J. Comput*, 5(1):158–180, March 1976.

[KRS96] J. Knoop, O. Rüthing, and B. Steffen. Towards a tool kit for the automatic generation of interprocedural data flow analyses. *Journal of Programming Languages*, 4(4):211–246, 1996.

[KU76] John B. Kam and Jeffrey D. Ullman. Global data flow analysis and iterative algorithms. *Journal of the ACM*, 23(1):158–171, 1976.

[LAMS04] T. Lev-Ami, R. Manevich, and M. Sagiv. TVLA: A System for Generating Abstract Interpreters. In R. Jacquart, editor, *Building the Information Society. IFIP 18th World Computer Congress, Topical Sessions*, pages 367–375, Toulouse, France, August, 22-27 2004. Kluwer Academic Publishers.

[LAS02] T. Lev-Ami and M. Sagiv. A System for Implementing Static Analyses. In M.V. Hermenegildo and G. Puebla, editors, *Static Analysis, 9th International Symposium, SAS 2002 Proceedings*, volume 2477 of *Lecture Notes in Computer Science*, pages 280–301, Madrid, Spain, September 17–20 2002. Springer-Verlag.

[LH88] J.R. Larus and P.N. Hilfinger. Detecting conflicts between structure accesses. *SIGPLAN Notices*, 23(7):21–34, June 1988. (PLDI 88).

[LN98] R. M. Leino and G. Nelson. Extended Static Checker for Modula-3. In K. Koskimies, editor, *Compiler Construction: 7th International Conference CC'98*, volume 1383 of *Lecture Notes in Computer Science*, pages 302–305. Springer-Verlag, April 1998.

[LR91] William Landi and Barbara G. Ryder. Pointer-induced aliasing : a clarification. In *18th POPL*, pages 93–103, 1991.

[LR92] William Landi and Barbara G. Ryder. A safe approximate algorithm for interprocedural pointer aliasing. *SIGPLAN Notices*, 27(7):235–248, July 1992.

[LRZ93] William Landi, Barbara G. Ryder, and Sean Zhang. Interprocedural modification side effect with pointer aliasing. *SIGPLAN Notices*, 28(6):56–67, 1993. Proceedings of PLDI'93.

[Mar02] M. Martel. Propagation of Roundoff Errors in Finite Precision Computations: A Semantics Approach. In D. Le Métayer, editor, *Programming Languages and Systems, llth European Symposium on Programming, ESOP 2002*, volume 2305 of *Lecture Notes in Computer Science*, pages 194–208, Grenoble, France, April 8–12 2002. Springer-Verlag.

[Mau04] L. Mauborgne. ASTRÉE: Verification of Absence of Run-Time Error. In R. Jacquart, editor, *Building the Information Society. IFIP 18th World Computer Congress, Topical Sessions*, pages 385–392, Toulouse, France, August, 22-27 2004. Kluwer Academic Publishers.

[Mil78] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, 1978.

[MR90] T.J. Marlowe and Barbara G. Ryder. Properties of data flow frameworks. *Acta Informatica*, 28:121–163, 1990.

[Muc97] S.S. Muchnick. *Compiler Design Implementation.* Morgan and Kaufmann, 1997.

[Mye81] E. W. Myers. A precise interprocedural data flow algorithm. In *8th POPL*, pages 219–230. ACM, 1981.

[NNH99] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag, 1999.

[OPS92] Nicholas Oxhoj, Jens Palsberg, and Michael I. Schwartsbach. Making type inference practical. In O.L. Madsen, editor, *Proc. ECOOP'92*, volume 615 of *LNCS*, pages 329–349. Springer-Verlag, 1992.

[Pal01] Jens Palsberg. Type-based analysis and its applications. In *PASTE'01 Workshop on Program Analysis for Software Tools and Engineering*, 2001.

[PS91] Jens Palsberg and Michael I. Schwartsbach. Object-oriented type inference. In *Proc. OOPSLA'91*, volume 26(10) of *SIGPLAN Notices*, pages 77–101. ACM Press, 1991.

[PS94] Jens Palsberg and Michael I. Schwartsbach. *Object-oriented Type Systems*. John Wiley and Sons, 1994.

[Ram99] C. Ramalingam. Identifying loops in almost linear time. *ACM Transactions on Programming Languages and Systems*, 21(3):175–188, March 1999.

[Rat92] C. Ratel. *Définition et réalisation d'un outil de vérification formelle de programme Lustre: Le système Lesar*. Thesis, Université Joseph Fourier, June 1992.

[RH98] S. Horwitz R. Hasti. Using static single assignment form to improve flow-insensitive pointer analysis : An efficient general iterative algorithm for dataflow analysis. In *PLDI'98*, 1998.

[Ros77] Barry K. Rosen. High-level data flow frameworks. *CACM*, 20(10):712–724, October 1977.

[RP86] Barbara G. Ryder and Marvin C. Paull. Elimination algorithms for data flow analysis. *ACM Computing Surveys*, 18(3):277–316, September 1986.

[RS00] F. Randimbivololona and J. Souyris. Improving avionics software verification cost effectiveness Abstract-interpretation-based technology contribution. In *DASIA 2000*, Montreal (Canada), May 22–26 2000.

[RWFS02] G. Ramalingam, A. Warshavsky, J.H. Field, and M. Sagiv. Deriving Specialized Program Analyses for Certifying Component-Client Conformance. *ACM SIGPLAN Notices*, 37(5):83–94, 2002.

[Sha80] Micha Sharir. Structural analysis: a new approach to flow analysis in optimizing compilers. *Computer Languages*, 5(3-4):141–153, 1980.

[Sre95] Vugranam C. Sreedhar. *Efficient Program Analysis using DJ Graphs*. PhD thesis, School of Computer Science,McGill University, Montreal, September 1995.

[SS00] M. Sheeran and G. Stålmarck. A tutorial on Stålmarck's proof procedure for propositional logic. *Formal Methods in System Design*, 16(1), January 2000.

[Ste96] B. Steensgaard. Points-to analysis in almost linear time. In *POPL'96*, pages 32–41, January 1996.

[TH92] S. Tjiang and J. Hennessy. Sharlit: A tool for building optimizers. *SIGPLAN Notices*, 27(7):82–93, July 1992.

[TSH+03] S. Thesing, J. Souyris, R. Heckmann, F. Randimbivololona, M. Langenbach, R. Wilhelm, and C. Ferdinand. An Abstract Interpretation-Based Timing Validation of Hard Real-Time Avionics. In *Proceedings of the International Conference on Dependable Systems and Networks DSN-2003*, San Francisco, CA, June 22–25 2003.

[Wei80] W.E. Weihl. Interprocedural data flow analysis in the presence of pointers, procedure variables, and label variables. In *7th POPL*, pages 83–94. ACM, 1980.

[Wil97] Robert P. Wilson. *Efficient Context-Sensitive Pointer Analysis For C Programs.* PhD thesis, Stanford University, Computer Systems Laboratory, December 1997.

[WL95] Robert P. Wilson and Monica S. Lam. Efficient context-sensitive pointer analysis for c programs. *SIGPLAN Notices*, 30(6):1–12, 1995. Programming Language Design and Implementation (PLDI).

[WM94] Reinhard Wilhelm and Dieter Maurer. *Les compilateurs : théorie, construction, génération.* Masson, 1994.

[Wol91] Michael Wolfe. Flow graph anomalies: What's in a loop? Technical Report CS/E 92-012, Oregon Graduate Institute of Science and Technology, OGI Department of Computer Science and Engineering, February 1991.

[Wol96] Michael Wolfe. *High Performance Compilers for Parallel Computing.* Addison-Wesley, 1996.

[Zha98] Sean Zhang. *Practical Pointer Aliasing Analyses for C.* PhD thesis, Rutgers University, August 1998.

[ZRL96] Sean Zhang, Barbara G. Ryder, and William Landi. Program decomposition pointer analysis: a step toward practical analyses. *Software Engineering Notes*, 21(6):81–92, November 1996. SIGSOFT'96: 4th ACM-SIGSOFT Symposium on the Foundations of Software Programming.