IS, un domaine numérique abstrait pour l'analyse de programmes manipulant des adresses

Mathias Péron - master informatique Systèmes et Logiciels

réalisé sous la direction
de Nicolas Halbwachs
année 2004/2005

Résumé

Dans le cadre de la vérification de propriétés de sûreté nous proposons un nouveau domaine numérique abstrait pour l'analyse de programmes manipulant des entiers comme adresses.

Les opérations sur ces adresses, qui peuvent être des adresses mémoires ou encore des adresses de composants dans des systèmes sur puces, sont restreintes par rapport à celles utilisées sur les entiers généraux . Ainsi, les domaines abstraits classiques proposés pour l'analyse des numériques ne sont pas réellement appropriés à une analyse dédiée aux adresses.
Nous nous sommes tournés vers la conception d'une analyse moins coûteuse que les analyses classiques mais permettant d'exprimer des propriétés d'égalité et de non-égalité de deux adresses. De ces propriétés dépendent de nombreuses autres, comme les propriétés d'exclusion mutuelle, que notre nouveau domaine, nommé IS , permet donc d'analyser dans un bon compromis entre précision et complexité.

Notre domaine défini, nous l'avons implémenté dans le cadre d'un outil de vérification par interprétation abstraite, et effectué des analyses dont nous exposons les résultats.

Table des matières

Introduction

La conception de programmes *corrects* a toujours été considérée comme une tâche complexe. Tout un chacun a expérimenté le temps nécessaire à l'élimination des erreurs non intentionnelles de programmation, les bugs, souvent bien plus important que le temps dispensé dans l'écriture du programme même.

L'utilisation des logiciels informatiques se généralise et l'on est de plus en plus dépendant de leur fiabilité. Les conséquences d'un comportement anormal peuvent être dramatiques, soit en terme de pertes financières, comme ce fut le cas par exemple pour le crash en 1996 de l'Ariane 5, soit en terme de vies humaines dans le cas de systèmes critiques (pilotage des centrales nucléaires, logiciels pour l'aéronautique, ...).

Si l'industrie du logiciel a pris conscience de la nécessité d'assurer la correction des programmes après leur conception, il s'avère que la complexité des logiciels semble suivre la loi de Moore et que de nombreux aspects complexes tels que la manipulation de structures de données non bornées, la concurrence, les processus parallèles, etc compliquent à l'extrême la validation du logiciel.

Contexte et définitions

La technique de validation la plus utilisée est probablement le *test*, qui consiste a explorer certaines traces d'exécution du programme dans un environnement donné et vérifier si elles sont conformes à ce qui est attendu ou, d'une manière plus rigoureuse, conformes à une spécification formelle. Une partie seulement des comportements peut alors être testée et de nombreux bugs ne sont pas détectés. La notion de *couverture* permet alors de quantifier la qualité des tests effectués et d'évaluer la confiance que l'on peut placer dans le logiciel testé.

L'objet de ce stage se situe dans le domaine de la *vérification*, qui est une autre classe de techniques de validation, basée sur des méthodes formelles permettant de prouver qu'un système est conforme à ses spécifications quelle que soit la situation à laquelle il est soumis.

Vérification

Le problème de la vérification de systèmes à nombre infinis d'états est connue pour être indécidable. Plusieurs approches ont alors été proposées ces vingt dernières années :

- se restreindre à des *cas décidables*, comme par exemple l'arithmétique de Presburger. Cependant les analyses sont coûteuses et le cadre d'application n'est pas général. Nous renvoyons le lecteur à [BW94, FS00].
- l'utilisation de *méthodes déductives* qui permettent d'exhiber une démonstration des propriétés souhaitées du système. Il s'agit de méthodes semi-automatiques car elles nécessitent généralement l'intervention de l'utilisateur pour guider la génération de la preuve – on

parle de preuve interactive. Cette approche ne peut plus s'appliquer sur des programmes de grande taille et s'avère de toute façon très coûteuse.

Une autre approche consiste à utiliser des modèles fini ou infini des programmes à vérifier et développer des techniques pour garantir sur ces modèles certaines propriétés. C'est cette approche que l'on va développer à présent.

Lorsqu'on souhaite analyser un programme, on en considère une forme abstraite. La plupart des systèmes peuvent être formalisés par des *systèmes de transitions.*

DÉFINITION 1 (SYSTÈME DE TRANSITIONS). *Un système de transitions est un triplet* $(Q, \rightarrow_Q, Q_{init})$ *où* Q *est un ensemble d'états,* $\rightarrow_Q \subset Q \times Q$ *la relation de transition entre un état et ses successeurs possibles, et où* $Q_{init} \subset Q$ *est l'ensemble des états initiaux.*

Nous nous plaçons dans le cadre de la vérification de *propriétés*, qui peuvent être vues comme un certain ensemble de chemins du système de transition partant de Q_{init}. Il est classique de distinguer deux grands types de propriétés

- les propriétés de *sûreté*, qui expriment que « quelque chose de mauvais » ne se produit jamais au cours de l'exécution du système. L'absence de dépassements arithmétiques par exemple est une propriété de sûreté. Elles sont caractérisées par le fait que leur violation est détectable par une exécution finie.
- les propriétés de *vivacité*, qui spécifient que « quelque chose de bon » se produit immanquablement durant l'exécution du système. Un exemple de telle propriété est la garantie de service. On voit qu'une telle propriété ne peut pas être violée par une exécution finie.

On ne s'intéressera dans ces travaux qu'à la vérification de propriété de sûreté. Il s'avère que dans le contexte des systèmes de transitions, montrer de telles propriétés peut toujours se ramener à prouver qu'un certain ensemble d'états d'*erreur* (noté Q_{err}) est inaccessible.

DÉFINITION 2 (ACCESSIBILITÉ). *Un état* q *(*resp. q'*) est dit accessible (*resp. *co-accessible) depuis un état* q' *(*resp. q*) si il existe un chemin entre* q *et* q'*.*

On note $\mathsf{Acc}(X)$ *(*resp. $\mathsf{CoAcc}(X)$*) l'ensemble de tous les états accessibles (*resp. *co-accessibles) depuis un état de* X*. Si on note* $post(X)$ *l'ensemble des états successeurs d'un état de* X *et* $pre(X)$ *l'ensemble des états prédécesseurs d'un état de* X *(selon* $\rightarrow_Q$*), il est connu que* $\mathsf{Acc}(X)$ *(*resp. $\mathsf{CoAcc}(X)$*) est la solution de l'équation de point fixe sur post (*resp. *pre).*

$$\mathsf{Acc}(X) = \bigcup_{n \geq 0} post^n(X) \qquad \mathsf{CoAcc}(X) = \bigcup_{n \geq 0} pre^n(X)$$

À l'évidence la preuve d'une propriété peut alors être effectuée de deux manières :

- par une *analyse en avant* en vérifiant que $\mathsf{Acc}(Q_{init}) \cap Q_{err} = \emptyset$
- ou par une *analyse en arrière* en vérifiant que $\mathsf{CoAcc}(Q_{err}) \cap Q_{init} = \emptyset$

Le coeur du problème de la vérification des propriétés des sûretés est donc le calcul de points-fixes.

Dans certains cas ce calcul peut être effectué exactement par la technique de « model-checking ». On traduit le programme vers un modèle fini ([GL93]), souvent un système de transitions étiquetées (STE), sur lequel les propriétés attendues sont exprimées en logique temporelle (par exemple la LTL, logique temporelle linéaire [Lam80]) et vérifiées par exploration exhaustive ([BCM+90]). Outre une explosion combinatoire possible la difficulté est de trouver pour *chaque* problème donné – *i.e.* un programme et un ensemble de propriétés à vérifier – un modèle fini adéquat.

Dans le cas général on ne sait que calculer des approximations des points-fixes. C'est ce que permet la théorie de *l'interprétation abstraite* sur des modèles infinis. Il s'agit d'une méthode totalement automatique, ce en quoi elle diffère des précédentes. Il suffit en effet de concevoir un *domaine abstrait* pour une classe de propriétés, dans lequel la sémantique du programme sera approximée pour obtenir une analyse statique d'un programme sur ces propriétés. Le modèle n'est donc pas lié à un programme en particulier mais à toute une classe de programmes.

Sujet et objectifs

Le sujet de ce MASTER est la vérification de propriétés de sûreté de programmes qui manipulent des entiers utilisés comme adresses.

Il peut s'agir d'adressage mémoire, ou encore d'adressage de composants par exemple dans les systèmes sur puces (SoCs). Dans la plupart des cas, les opérations sur ces adresses sont très restreintes par rapport à celles utilisées sur les entiers généraux. Elles sont principalement des incrémentations, parcours d'intervalle ou encore des comparaisons d'égalité.

De nombreuses propriétés dépendent de l'égalité ou non – on dira de la non-égalité[1] – des adresses. C'est le cas de l'accès en *exclusion mutuelle* d'un composant ou d'une zone mémoire. Des propriétés d'exclusion de lecture/écriture sont aussi au coeur même de la vérification des protocoles de communication ou encore des programmes parallèles.

Nous verrons que l'égalité/non-égalité d'entiers représentant des adresses est mal prise en compte par les domaines numériques abstraits proposés jusqu'alors pour l'analyse par interprétation abstraite.

L'objectif est donc de développer une analyse dédiée aux adresses, éventuellement moins coûteuse que les analyses classiques sur les variables numériques, mais capable de traiter des propriétés d'égalité et de non-égalité. Le cadre théorique que nous allons utiliser est celui de l'interprétation abstraite, qui est un domaine d'excellence de l'équipe d'accueil, et il s'agira donc de proposer un nouveau domaine abstrait pour cette classe de problème. Enfin on s'attachera à implémenter ce domaine abstrait dans un outil d'analyse à des fins d'expérimentation.

Plan du mémoire

Ce mémoire est articulé de la façon suivante :

- *Le premier chapitre* a pour objet l'état de l'art sur l'analyse des programmes par interprétation abstraite.
  L'ensemble des principes de cette théorie sont présentés, suivi d'une description des domaines abstraits existants pour l'analyse des propriétés numériques. Enfin la méthode d'analyse statique par interprétation abstraite est décrite, agrémentée d'un exemple.
- *Le second chapitre* présente le résultat théorique de nos travaux qu'est le domaine numérique abstrait IS .
  Une première partie fixe les propriétés que doit représenter le domaine pour répondre au problème efficacement, *i.e.* constituer un bon compromis entre la précision et la complexité de l'analyse. Puis, un travail important est effectué sur la représentation canonique du domaine. Enfin les dernières sections définissent l'ensemble des formalismes nécessaires pour respecter le cadre de l'interprétation abstraite.
- *Le troisième chapitre* présente notre implémentation du domaine IS dans un prototype d'outil d'analyse et les résultats obtenus par cet outil sur différents programmes.

[1] que les anglophones appellent « disequality »

Pour chacun d'entre eux une discussion est engagée sur les propriétés que le domaine IS permet de prouver.

– *Le dernier chapitre*, expose les perspectives de ces travaux, qui à court terme s'inscrivent dans une implémentation plus efficace du domaine, et à moyen terme, dans l'utilisation du domaine pour la vérification de systèmes concurrents.

Chapitre 1

Interprétation abstraite

L'interprétation abstraite est une théorie générale d'approximation des sémantiques introduite par Patrick et Radhia Cousot [CC77] (référence historique à laquelle on préférera l'article [Cou01]) à la fin des années 70.

Généralement cette théorie n'est abordée que pour l'approximation de systèmes dynamiques discrets, dont l'analyse peut se réduire au calcul de points fixes. L'interprétation abstraite est alors vue comme une théorie de résolution approchée d'équations de points fixes.

C'est ainsi que nous allons en présenter les principes en première section. La section 1.2 rappellera l'ensemble des abstractions proposées pour les ensembles de valeurs numériques. Enfin la section 1.3 exposera l'utilisation de l'interprétation abstraite pour l'analyse statique de programmes.

1.1 Principes

Le théorème de Kleene* établit l'existence d'une plus petite solution[1] à l'équation de point fixe $x = F(x)$, où $x \in L$, $F : L \to L$ continue* et $(L, \bot, \top, \sqsubseteq, \sqcup, \sqcap)$ est un treillis complet* :

$$\mathrm{lfp}(F) = \bigsqcup_{n \geq 0} F^n(\bot) \tag{1.1}$$

La résolution de l'équation (1.1) pose deux problèmes :

- *la manipulation de valeurs complexes*, c'est à dire la complexité du domaine de calcul L. Dans le cadre de l'analyse de programme L est l'ensemble 2^Q des parties de l'ensemble des états Q du programme. On parlera du treillis *concret* – noté C. Ce treillis est généralement infini et donc difficilement représentable de manière efficace et non manipulable algorithmiquement.
  Il suffit de considérer l'ensemble $\{(x, y, z, n) \in \mathbb{N}^4 \mid x^n + y^n = z^n \wedge n \geq 3\}$, cher à Andrew Wiles et Pierre de Fermat, pour s'en convaincre.
- *la résolution itérative de l'équation.* Dans le cas général, il est impossible de calculer la limite d'une itération infinie.

1.1.1 Abstraction

L'interprétation abstraite répond au premier problème en introduisant la notion de *domaine abstrait*. Il s'agit d'approximer un ensemble de valeurs de C par une valeur plus « simple » du domaine abstrait.

[1] notée *lfp* pour « least fixed point »

Il faut donc choisir celui-ci assez expressif pour permettre de vérifier les propriétés qui nous intéressent. Nous avons besoin de définir :

- une fonction d'abstraction, notée α, qui transforme une valeur du domaine concret en une valeur abstraite.
- une fonction de concrétisation, notée γ, qui à partir d'une valeur abstraite renvoie une valeur concrète. En reprenant pour exemple l'analyse de programme, $\gamma(a)$ est un ensemble maximal X qui s'abstrairait en a.
- la fonction abstraite $F^{\#}$ de F que l'on peut définir par

$$F^{\#} = \alpha \circ F \circ \gamma$$

Ces définitions sont illustrées par la figure 1.1.

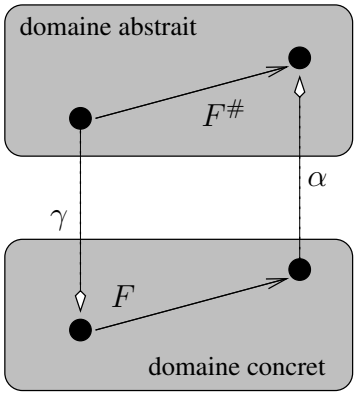


FIG. 1.1 – Abstraction : fonctions α et γ, fonction abstraite $F^{\#}$

L'idée sous-jacente est de calculer le point fixe de la fonction $F^{\#}$ dans le domaine abstrait. On souhaite que le résultat concrétisé de cette équation de point fixe abstraite soit une sur-approximation du résultat de l'équation de point fixe concrète. Pour cela le couple de fonctions (α, γ) doit vérifier certaines conditions que formalise la notion de connexion de Galois.

On notera que la définition donnée de $F^{\#}$ va à l'encontre de la volonté d'avoir une complexité moindre pour le calcul du point fixe de $F^{\#}$ que pour celui de F, puisqu'elle fait appel au calcul de F dans le domaine concret. Nous verrons comment ce problème est contourné.

DÉFINITION 3 (CONNEXION DE GALOIS). *Soit deux treillis $(C, \sqsubseteq_C)$ et $(A, \sqsubseteq_A)$, appelés respectivement concret et abstrait. On appelle connexion de Galois entre C et A un couple de fonctions (α, γ) tel que :*

$$\forall c \in C, \forall a \in A \qquad \alpha(c) \sqsubseteq_A a \iff c \sqsubseteq_C \gamma(a)$$

où $\alpha : C \to A$ est appelée abstraction et $\gamma : A \to C$ est appelée concrétisation.

D'une connexion de Galois découlent de nombreuses propriétés. Entre autres, l'extensivité de $\gamma \circ \alpha$:

$$\forall c \in C \quad c \sqsubseteq_C \gamma(\alpha(c))$$

Cette propriété, illustrée à la figure 1.2 exprime le fait que l'on a bien une sur-approximation – une perte d'information – d'un ensemble lorsqu'il est abstrait et que l'on en calcule de nouveau la signification concrète.

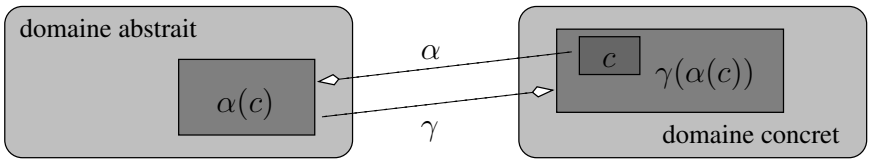


FIG. 1.2 – Connexion de Galois : extensivité de $\gamma \circ \alpha$

Résultat fondamental Le théorème suivant est la clef de voûte de l'interprétation abstraite :

THÉORÈME 1 (APPROXIMATION DE POINT-FIXE). *Soit C et A deux treillis reliés par une connexion de Galois (α, γ), F une fonction continue de C dans lui-même et G une fonction continue de A dans lui-même, alors si G est une approximation supérieure de $F^\#$* i.e.

$$\forall c \in C \quad \alpha(F(c)) \sqsubseteq_A G(\alpha(c))$$

alors

$$\mathit{lfp}(F) \sqsubseteq_C \gamma(\mathit{lfp}(G))$$

Ce résultat signifie que la concrétisation du plus petit point-fixe d'une approximation supérieure G de $F^\#$ dans le domaine abstrait est une sur-approximation du plus petit point-fixe de F dans le domaine concret. Comme pressenti, c'est en choisissant $F^\#$ pour G qu'on obtiendra le résultat le plus précis.

Ce théorème est puissant à deux titres : non seulement il assure que l'on a bien une sur-approximation du point-fixe de F en effectuant nos calculs dans le domaine abstrait, mais de plus il permet d'utiliser des fonctions abstraites plus simples que $F^\#$ ce qui est généralement nécessaire au vu des considérations précédentes sur la calculabilité de F[2].

1.1.2 Convergence

Il reste le problème de la résolution de l'équation de point fixe. On est amené, pour une certaine fonction G continue à calculer la limite de la suite

$$x_0 = \bot, \quad x_{n+1} = G(x_n) \tag{1.2}$$

Si le treillis abstrait est fini ou de profondeur* finie le calcul itératif du point-fixe converge.

En fait, les meilleurs résultats sont obtenus par *extrapolation*, *i.e.* grâce à l'application d'un opérateur d'*élargissement* qui consiste à deviner la limite d'une suite infinie au vu de ses premiers termes.

DÉFINITION 4 (OPÉRATEUR D'ÉLARGISSEMENT). *Soit $(L, \sqsubseteq, \sqcup)$ un treillis complet. Un opérateur d'élargissement est une fonction $\nabla : L \times L \to L$ satisfaisant les deux propriétés suivantes :*

1. $\forall a_1, a_2 \in L \quad a_1 \sqcup a_2 \sqsubseteq a_1 \nabla a_2$
2. *pour toute suite croissante $(x_n)_{n \geq 0}$ dans L, la suite définie par*

$$a_0 = x_0, \quad a_{n+1} = a_n \nabla x_{n+1}$$

converge en un nombre fini *d'itérations*

[2]Nous verrons à la section 1.3 comment on peut construire de façon systématique la fonction G, dans le cadre de l'analyse de programmes

Cet opérateur introduit une nouvelle approximation, celle de la limite de la suite (1.2) que l'on calcule par la suite

$$y_0 = \bot, \quad y_{n+1} = y_n \nabla G(y_n)$$

qui converge en un nombre fini d'itérations vers une limite $\hat{y}$ qui un post-point-fixe de G (et donc une approximation correcte de $\mathrm{lfp}(G)$).

Si $\hat{y}$ n'est pas un point fixe de G on peut parfois améliorer l'approximation $\hat{y}$ en lui appliquant la fonction G. C'est ce qu'on appelle la *séquence descendante*.

Théorème 2 (Séquence descendante). *Si $\hat{y} \sqsupseteq G(\hat{y})$, alors la suite*

$$z_0 = \hat{y}, \quad z_{n+1} = G(z_n)$$

décroît et tous ses termes sont des approximations supérieures de lfp(G).

La suite $(z_n)_{n\geq 0}$ converge vers un point fixe de G, mais il n'est pas garanti que ce soit en un nombre fini de termes. Cependant tous ses termes étant des approximations correctes de plus en plus précises, dans la pratique on calcule les premiers termes jusqu'à un certain seuil.

Nous clôturons cette section par un résumé des principes énoncés à la figure 1.3.

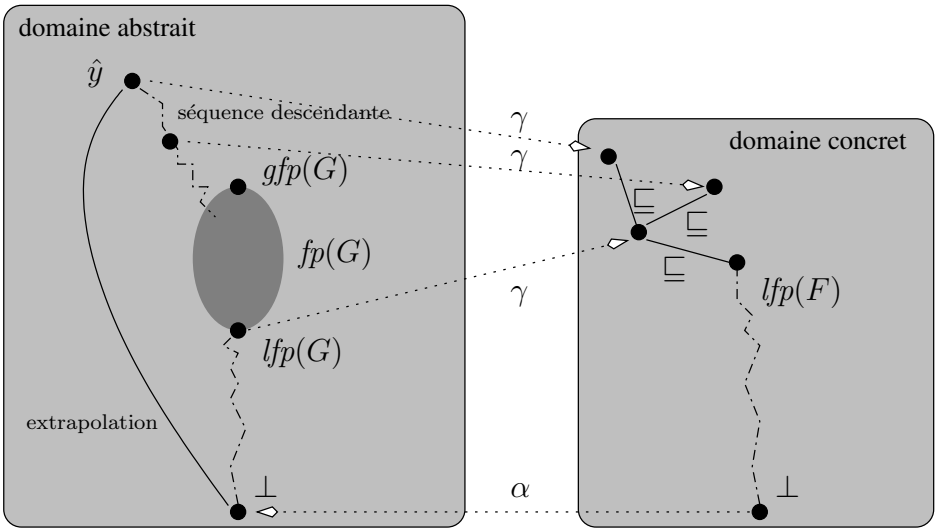


Fig. 1.3 – Interprétation abstraite : abstraction de point-fixe, convergence

1.2 Domaines numériques abstraits

Dans les programmes que l'on souhaite vérifier, nous allons nous intéresser plus particulièrement aux aspects numériques.

Il va s'agir d'interprétations abstraites où le but sera d'attacher à chaque point de contrôle d'un programme une approximation de l'ensemble des valuations possibles sur toutes les exécutions en ce point. Une valuation est une fonction ρ qui à une variable (ensemble Id) associe une valeur (ensemble $\mathcal{N}$)

$$\rho : Id \to \mathcal{N}$$

Le domaine concret que l'on considère est alors l'ensemble $2^{\mathcal{N}^n}$ où $\mathcal{N}$ est l'ensemble numérique $\mathbb{Z}$, $\mathbb{Q}$ ou $\mathbb{R}$ et n le nombre de variables.

Les principaux domaines numériques abstraits proposés dans la littérature sont présentés dans cette section par ordre chronologique d'étude. Il est cependant important de les comparer selon leur puissance d'expression, comparaison que l'on trouvera à la table 1.1.

On distingue deux grandes familles de domaines abstraits selon qu'ils permettent ou non d'exprimer que les valeurs de deux variables distinctes (ou plus) sont liées : les premiers sont qualifiés de *relationnels*. Le premier proposé dans cette famille fut celui des égalités linéaires [Kar76].

non−relationels	signes $0 \leq x$ intervalles $a \leq x \leq b$	congruences simples $x = a \mod b$
2−relationnels	zones $a \leq \pm x, \quad x - y \leq b$ ocotogones $\pm x \pm y \leq a$	
n−relationnels	octaèdres $\pm x \pm y \pm z \leq a$ polyèdres $ax + bx + cz \leq d$ égalités linéaires $ax + by = c$	congruences linéaires $ax + by = c \mod d$ congruences trapézoidales $ax + by \in [c, d] \mod e$

TAB. 1.1 – Domaines numériques abstraits classés par expressivité

La plupart des domaines proposés jusqu'alors considèrent des relations linéaires. Il existe cependant des domaines abstraits pour les égalités polynomiales par exemple [RCK04].

Pour chaque domaine abstrait, un exemple d'abstraction d'ensemble de valeurs concrètes dans $\mathcal{N}^2$ est donné (figures 1.4, 1.5, 1.7, 1.9). À la fin de la section le lecteur trouvera à la figure 1.11 un exemple pour chaque domaine non détaillé dans cette section et à la table 1.2 un résumé des complexités des domaines numériques abstraits principaux.

1.2.1 Intervalles

Une approximation simple consiste à déterminer pour chaque variable l'intervalle minimal dans lequel elle varie.

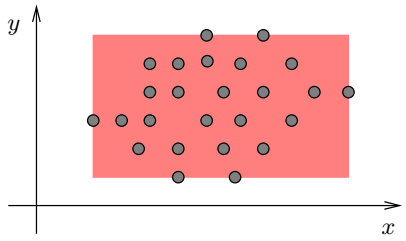


FIG. 1.4 – Abstraction en intervalles

On définit donc le treillis des intervalles [CC76] sur $\mathcal{N}$, avec pour valeurs abstraites

$$\bot_I \cup \{[\mathsf{lb}, \mathsf{ub}] \mid \mathsf{lb} \in \mathcal{N} \cup \{-\infty\} \wedge \mathsf{ub} \in \mathcal{N} \cup \{+\infty\} \wedge \mathsf{lb} \leq \mathsf{ub}\}$$

ordonnées par l'inclusion sur les intervalles. L'opérateur de borne inférieure, noté $\sqcap^I$ est l'intersection sur les intervalles alors que la borne supérieure de deux intervalles consiste à prendre le

plus petit intervalle contenant les deux autres

$$[a,b] \sqcup^I [c,d] = [min(a,c), max(b,d)]$$

Élargissement Il s'agit d'un treillis de hauteur infinie et il est donc nécessaire de définir un opérateur d'élargissement. L'idée est la suivante : si une borne de l'intervalle s'éloigne cela signifie qu'elle peut continuer à s'écarter en prenant n'importe quelle valeur sur cette borne jusqu'à l'infini (intuitivement la variable subit une affectation dans une boucle). Pour éviter ce risque d'itérations infinies, la borne est directement repoussée à l'infini par l'élargissement :

$$[a,b]\nabla^I[c,d] = [\text{si } c < a \text{ alors } -\infty \text{ sinon } a, \text{si } d > b \text{ alors } +\infty \text{ sinon } b]$$

$$\perp_I \nabla^I [a,b] = [a,b]$$

Il s'agit bien d'un élargissement (*prop 2*), une borne déplacée à l'infini ne pouvant faire l'objet d'une nouvelle modification.

1.2.2 Polyèdres

Le treillis des polyèdres convexes a été proposé par [CH78] pour l'analyse de relations linéaires.

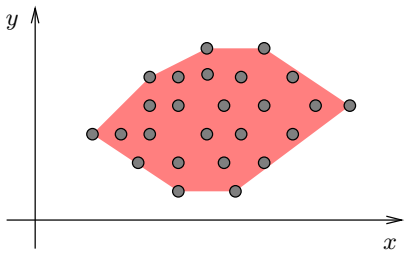


FIG. 1.5 – Abstraction en polyèdres

Représentation Un fait important est la représentation duale des polyèdres, qui peuvent être décrits soit par un système de contraintes d'inégalités linéaires,

$$\sum a_i x_i \leq c \qquad a_i \in \mathcal{N}$$

soit par un système *générateur* où l'ensemble des points du polyèdre sont décrits comme la somme d'une combinaison convexe de sommets et d'une combinaison positive de rayons

$$\sum_{s_i \in \text{Som}} \lambda_i s_i + \sum_{r_j \in \text{Ray}} \mu_j r_j \qquad \lambda_i \geq 0, \mu_j \geq 0, \sum_i \lambda_i = 1$$

La figure 1.6 donne un exemple de polyèdre avec ses deux représentations.

Pour chacune des opérations sur le treillis ou nécessaires pour l'analyse, l'une des deux représentations est plus commode en terme de complexité pour effectuer l'opération. Le problème réside dans le fait que le passage d'une représentation à l'autre a un coût exponentiel (en temps comme en espace : un hypercube à n-dimensions a 2^n sommets alors qu'on peut l'exprimer avec n contraintes).

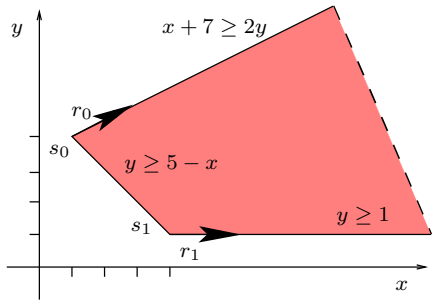


FIG. 1.6 – Représentations duales d'un polyèdre

Afin de réduire l'impact de cette complexité exponentielle, qui se traduit au niveau des opérations – l'intersection et l'union font appel à un changement de représentation, [HMPV03] proposent une factorisation cartésienne des polyèdres, en polyèdres de tailles strictement plus petites définis sur des ensembles disjoints de variables tels qu'il n'existe pas de contraintes entre deux variables de deux de ces ensembles.

Opérations Alors que l'intersection de deux polyèdres convexes est nécessairement convexe, l'union elle oblige à une sur-approximation – à l'instar des intervalles. Il s'agit pour P et Q du plus petit polyèdre convexe contenant P et Q, que l'on appelle l'*enveloppe convexe*.

On définit l'opération de borne inférieure comme l'intersection et celle de borne supérieure comme l'enveloppe convexe de polyèdres.

Le treillis des polyèdres convexes n'est pas complet (cependant c'est un sous-treillis du treillis des convexes qui lui est complet), par exemple, une suite infinie de polyèdres convexes peut avoir pour limite une sphère.

L'opérateur d'élargissement va permettre de rester dans le treillis des polyèdres convexes en évitant des unions infinies. L'élargissement de P par Q est obtenu en gardant dans P toutes les inégalités vérifiées par Q. L'idée est en fait identique à celle mise en oeuvre sur les intervalles : la transformation (translation, rotation, ...) d'une contrainte est projetée à l'infini.

Le domaine abstrait des polyèdres souffre incontestablement de sa complexité exponentielle. Ainsi différentes équipes de recherche ont étudié des cas particuliers des polyèdres que sont les domaines des zones, octogones et octaèdres décrits plus loin.

1.2.3 Zones

Les zones ont été introduites pour la vérification des automates temporisés. Ce sont des automates finis où certaines variables sont des horloges qui évoluent de façon continue et synchrone.

Le contrôle de l'automate est contraint par des invariants exprimés sur les horloges (de la forme $x \prec a$ où $\prec$ est un opérateur dans $\{<, \leq, =, \geq, >\}$ et où a est une constante) au niveau des états. Les transitions sont également conditionnées par des contraintes de la même forme. L'invariant en un état et la condition d'une des transitions de cet état peuvent être tout deux vérifiés à un temps t introduisant ainsi un indéterminisme.

Lorsque l'on souhaite effectuer une analyse d'accessibilité sur les automates temporisés [Dil89] on est amené à considérer les contraintes sur les horloges mais également sur les différences d'horloges (*i.e.* les contraintes $x - y \prec a$).

Une zone est définie comme une conjonction de contraintes de la forme $x \prec a$ ou $x - y \prec a$[3].

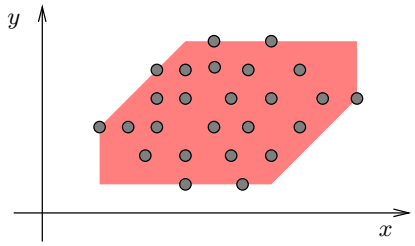


FIG. 1.7 – Abstraction en zones

On représente classiquement les zones par des *matrices de bornes* (DBMs, Difference Bound Matrices), qui sont une représentation matricielle introduite par [Dil89] d'un graphe orienté où les sommets sont les variables et où les arêtes sont valuées par la différence de deux variables incidentes (on introduit l'horloge nulle pour les contraintes $x \prec a$). La figure 1.8 donne un exemple de DBM avec sa représentation sous forme de graphe.

$$\begin{cases} 1 \leq x \\ 1 \leq y \leq 2 \\ 1 \leq y - x \end{cases}$$

	0	**x**	**y**
0	0	-1	-1
x	∞	0	1
y	2	∞	0

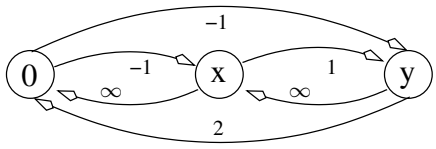


FIG. 1.8 – Un exemple de matrice de bornes (DBM)

Un même système de contraintes peut être représenté par différentes DBMs. Il existe une forme canonique qui est la DBM où les bornes sont minimales. La normalisation s'effectue donc par un calcul des plus courts chemins sur le graphe décrit précédemment (algorithme de Floyd-Warshall [CLRT90]). Sur notre exemple, le résultat est l'apparition des contraintes $x - 0 \leq 3$ et $y - x \leq 1$.

Opérations L'opération de borne inférieure (*resp.* de borne supérieure) s'effectue en prenant le minimum (*resp.* maximum) des bornes sur les deux DBMs. L'opération de borne inférieure est exacte au contraire de la borne supérieure : les DBMs ne sont pas stables par union.

Enfin, l'opérateur d'élargissement pourrait être celui des intervalles étendu aux différences de variables. Cependant un tel opérateur n'a jamais été utilisé, probablement car il n'a pas d'utilité pour l'analyse des automates temporisés.

La complexité des opérations évoquées est en $O(n^3)$ de par l'algorithme de normalisation. Il est à noter que la canonisation est en $O(n^2)$ si elle est effectuée de manière incrémentale.

1.2.4 Octogones

Il s'agit d'une généralisation des zones, proposée par [Min01b], qui permet les contraintes de la forme $x + y \prec a$. Grâce à une implémentation basée sur les DBMs, la complexité des opérations sur le treillis n'a pas augmenté.

[3]Les zones ont donc pour sous-treillis le treillis des intervalles

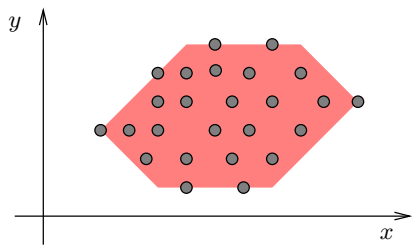


FIG. 1.9 – Abstraction en octogones

D'excellents résultats ont été obtenus avec les octogones [BCC+03] . Ils devraient être ajoutés prochainement à l'outil d'analyse linéaire NBAC – *c.f.* section 3 – du laboratoire, comme alternative aux polyèdres.

1.2.5 Octaèdres

Récemment [CC04] propose un domaine abstrait pour l'analyse des délais dans les circuits synchrones. La correction de tels circuits se base sur des contraintes de temps qui le plus souvent consistent à comparer les délais de deux chemins du circuit. Amener à considérer des *sommes* et des *différences* sur les délais, ils ont choisi d'étudier les inégalités linéaires où les coefficients sont dans $\{-1, 0, +1\}$.

À la différence des octogones dont ils sont une généralisation à n variables, la complexité des opérations n'est plus polynomiale.

L'implémentation de leur domaine est basée sur des diagrammes de décision, nommés OhDD pour *Octahedra Decision Diagram*. Malgré les avantages qu'elle apporte (partage de contraintes[4], diminution de l'espace mémoire utilisé) les résultats sont peu encourageants quant aux gains face aux polyèdres. Les auteurs travaillent actuellement sur une structure de données plus efficace.

1.2.6 Templates

Les templates [SSM05], que l'on traduira par « patrons », sont une famille de domaines. Ils permettent de prendre pour domaine un ensemble d'inégalités linéaires $\sum a_i x_i + c \geq 0$ où les a_i sont choisi *a priori* et *avant* l'analyse (matrice T).

Les patrons sont donc moins puissants que les polyèdres mais permettent d'exprimer un système de contraintes où certaines sont du domaine des intervalles, d'autres du domaine des octogones ou encore des octaèdres, des zones, ...

Ainsi le domaine des zones sur l'ensemble de variables $\{x, y\}$ n'est autre qu'un template particulier :

$$\begin{bmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 1 \\ 0 & -1 \end{bmatrix} \longrightarrow \begin{cases} x + c_1 \geq 0 \\ -x + c_2 \geq 0 \\ y + c_3 \geq 0 \\ -y + c_4 \geq 0 \end{cases}$$

et celui des octogones est donnée par la matrice

[4] À l'instar des BDDs (*Binary Decision Diagrams*) qui sont une représentation compacte des fonctions booléennes de $\mathbb{B}^n$ dans $\mathbb{B}$

$$\begin{bmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 1 \\ 0 & -1 \\ 1 & 1 \\ -1 & 1 \\ 1 & -1 \\ -1 & -1 \end{bmatrix} \longrightarrow \begin{cases} x + c_1 \geq 0 \\ -x + c_2 \geq 0 \\ y + c_3 \geq 0 \\ -y + c_4 \geq 0 \\ x + y + c_5 \geq 0 \\ -x + y + c_6 \geq 0 \\ x - y + c_7 \geq 0 \\ -x - y + c_8 \geq 0 \end{cases}$$

On donne à la figure 1.10 un dernier exemple de matrice de template T.

$$\begin{bmatrix} 1 & 0 & 0 \\ -1 & 0 & -1 \\ 0 & 1 & 0 \\ 1 & 1 & 0 \\ 1 & -1 & 0 \\ 1 & 2 & -1 \end{bmatrix} \longrightarrow \begin{cases} x + c_1 \geq 0 \\ -x - z + c_2 \geq 0 \\ y + c_3 \geq 0 \\ x + y + c_4 \geq 0 \\ x - y + c_5 \geq 0 \\ x + 2y - z + c_6 \geq 0 \end{cases}$$

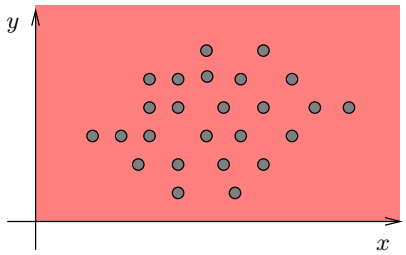
FIG. 1.10 – Un exemple de matrice de template (T)

Une fois ce « patron polyédrique » fixé, les valeurs abstraites ne sont en fait que le vecteur c de constantes complétées de l'ensemble $\{-\infty, +\infty\}$, où la valeur $+\infty$ sur une des contraintes la retire du système.

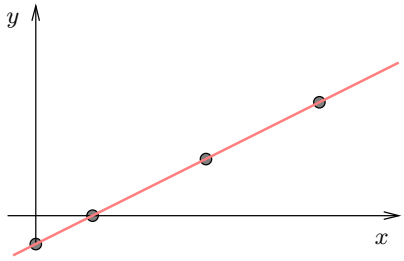
Les vecteurs c sont calculés par programmation linéaire, avec pour fonction objectif la minimisation de c afin d'obtenir le polyèdre minimal contraint par T.

Cette approche devient très intéressante lorsque l'on considère qu'on peut, à chaque point du programme, choisir une matrice T de contraintes différente, diminuant ainsi la complexité en général. Une méthode est proposée pour construire cette matrice à partir du modèle du programme qui permet d'éviter de perdre des invariants non-triviaux.

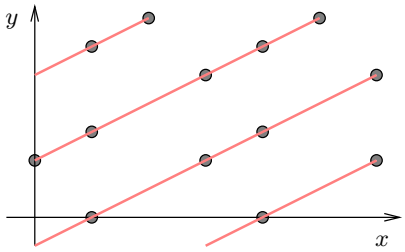
Enfin, la complexité de l'analyse est polynomiale faisant des templates des domaines prometteur.



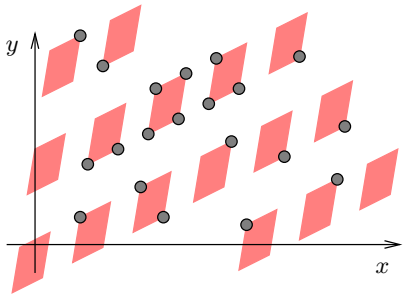
(a) signes $\{x \geq 0, y \geq 0\}$



(b) égalités linéaires $\{x = 2y + 2\}$



(c) congruences linéaires $\{x - 2y = 2 \mod 7\}$



(d) congruences trapézoïdales $\{x - 3y \in [2, 8] \mod 11,\ 8x - 2y \in [9, 20] \mod 33\}$

FIG. 1.11 – Diverses autres abstractions numériques

	intervalles	*polyèdres*	*octogones*	*templates*
temps	$O(n)$	$O(p.2^n)$	$O(n^3)$	$O(p.n^k)$
espace	$O(n)$	$O(p.n)$	$O(n^2)$	$O(p.n)$

n : nombre de variables, p nombre d'inéquations (*fixé* pour les templates), k paramètre de l'opérateur d'élargissement.

TAB. 1.2 – Résumé des complexités en temps et en espace des opérations

1.3 Application à l'analyse statique de programmes

L'interprétation abstraite offre, comme nous l'avons vu, un cadre théorique puissant pour le calcul de points-fixes, problème qui est au coeur de la vérification de propriétés d'accessibilité.

Nous allons utiliser un modèle des programmes relativement général, et de plus haut niveau que les systèmes de transition, que sont les *automates interprétés*.

Après la définition de ce modèle et de sa sémantique, cette section décrit comment se déroule l'analyse par interprétation abstraite des automates interprétés.

1.3.1 Modèle des programmes

Une décomposition naturelle du domaine concret dans le cas de l'analyse de programme est la partition de l'ensemble des états du programme selon ses points de contrôles (ensemble K). À chaque point de contrôle k est donc associé un ensemble de valuations possible des variables.

Un programme peut alors être vu simplement comme une automate modifiant la valuation courante par application d'une série d'actions élémentaires d'un point de contrôle à un autre.

C'est cette vision que traduisent les automates interprétés. Un automate interprété sera constitué d'un ensemble de points de contrôle (dont un point initial), à chacun desquels est associé un ensemble de commandes gardées, agissant sur des variables, et menant à d'autres points de contrôle.

Définition 5 (Garde, action, commande gardée).
- *Une* garde *est une fonction* $g : \{Id \to \mathcal{N}\} \to \{vrai, faux\}$*, donnant la valeur d'une formule logique sur une valuation. On note Gardes l'ensemble des gardes.*
- *Une* action *est une relation binaire sur les valuations. On note Act l'ensemble des actions possibles.*
- *Une* commande *est un quadruplet* $(k, g, a, k') \in (K \times Gardes \times Act \times K)$*. Intuitivement, lorsque le contrôle de l'automate est en* k*, et si la valuation courante* V *satisfait la garde* g (i.e. $g(V) = vrai$)*, l'exécution de la commande fait passer au point de contrôle* k'*, avec une valuation* V' *telle que* $(V, V') \in a$*.*

On notera que les actions peuvent être non-déterministes comme par exemple l'action $y \leftarrow ?$ qui permet d'exprimer une valuation où la valeur d'une certaine variable devient quelconque.

Définition 6 (Automate interprété). *Un automate interprété est un triplet* (K, Com, k_0)*, où* K *est un ensemble fini de points de contrôle*[5]*,* Com *est un ensemble fini de commandes, et* $k_0 \in K$ *est le point de contrôle initial.*

La figure 1.12 donne un exemple d'un programme et de sa modélisation par un automate interprété.

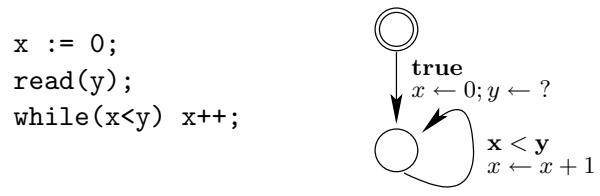


Fig. 1.12 – Un automate interprété

[5]On parle aussi de « places » plutôt que d'états afin de ne pas confondre avec les états du programme.

Sémantique de collection On rappelle que le but de l'analyse est de sur-approximer l'ensemble des états accessibles. On va chercher à décomposer le problème en tirant partie de la structure de l'automate interprété : on s'intéresse plutôt aux valuations accessibles en chaque point de contrôle, *i.e.* aux places de l'automate.

Soit R_k l'ensemble des valuations accessibles au point de contrôle k

$$R_k = \{\rho \mid (k, \rho) \in \mathsf{Acc}(k_0)\}$$

où k_0 est le point de contrôle initial. On peut exprimer les R_k comme les plus petites solutions d'un système d'équations de points-fixe :

$$R_{k_0} = \{Id \to v \mid v \in \mathcal{N}\}, \qquad R_k = \bigcup_{(k',g,a,k)} a(R_{k'} \cap \overline{g})$$

où $\overline{g}$ représente l'ensemble des valuations respectant la garde g *i.e.* $\overline{g} = \{\rho \in Id \to \mathcal{N} \mid g(\rho) = vrai\}$.

L'avantage de cette sémantique (connue sous le nom de « collecting semantic ») des automates interprétés est qu'elle permet de considérer un ensemble d'équations de points fixes, où les opérations sont élémentaires, au lieu d'une équation de point-fixe unique plus compliquée a priori. On parle de *partitionnement* de l'équation.

À présent, si l'on en revient à l'interprétation abstraite, il est visible que l'on a trouvé un moyen de construire systématiquement la fonction abstraite G des équations de points-fixes du système exhibé. Il suffit de définir pour chaque action élémentaire du langage de programmation et chaque garde son équivalent abstrait, que nous noterons respectivement $a^\#$ et $g^\#$. On construit donc une *sémantique abstraite* et on obtient le système d'équations abstraites :

$$R_k^\# = \bigsqcup_{(k',g,a,k)}^{\#} a^\#(R_{k'}^\# \sqcap^\# \overline{g^\#})$$

On mesure à présent toute l'importance de la complexité des opérations d'union et d'intersection dans le domaine abstrait.

Attention, cette sémantique doit-être définie de telle sorte qu'elle préserve le caractère conservatif de l'analyse ce qui se traduit par exemple au niveau d'un action abstraite $a^\#$ par la condition

$$\forall c \in C \quad a(c) \sqsubseteq_C \gamma(a^\#(\alpha(c)))$$

1.3.2 Analyse

Résolution Le système d'équations de points-fixe abstrait peut être résolu de manière parallèle jusqu'à stabilisation de chaque équation. Cependant, il s'agit d'une approche basique.

De prime abord il est évident que l'on peut se servir de la structure de contrôle pour éviter de nombreux calculs. Par exemple, on fera converger les valeurs sur les points de contrôle d'une composante fortement connexe (CFC) avant de considérer les points de contrôle en aval de celle-ci.

Une stratégie d'itération est en fait mise en place à partir de la décomposition que l'on a obtenue et son graphe de dépendance. Elle est formalisée par la notion d'*itération chaotique* [CC77] qui est en fait une suite de parties de K où chaque point de contrôle apparaît infiniment souvent. Cette stratégie $(K_i)_{i\geq 0}$ consiste alors à calculer en parallèle à chaque i les R_k où $k \in K_i$.

Un second point d'une grande importance est la restriction de l'application de l'opérateur d'élargissement.

Sachant la perte d'information que l'opérateur peut induire, la précision de l'analyse sera fortement améliorée en choisissant un sous-ensemble K_∇ de K sur les équations duquel on appliquera l'opérateur d'élargissement.

K_∇ doit être tel que toute boucle du graphe de dépendance – là où se trouvent potentiellement les itération infinies – ait l'un de ses sommet dans cet ensemble. Le choix d'un ensemble minimal de points d'élargissement est un problème **NP**-complet, et on utilise des heuristiques comme le calcul des sous-composantes fortement connexes proposé dans [Bou92]. L'idée est de calculer les CFCs du graphe de dépendance et de choisir pour chacune d'elle un sommet p qui sera un point d'élargissement (on choisit le sommet d'entrée de la CFC découvert dans l'algorithme de Tarjan). Cependant tous les cycles ne seront pas couvert : il peut y avoir des cycles dans une CFC ne passant pas par le sommet p choisi. Pour déterminer un ensemble admissible de point d'élargissement, on ôte les points p du graphe et on effectue de nouveau un calcul des CFCs (d'où le nom de la méthode) pour chacun desquels on choisit un sommet qui sera un point d'élargissement que l'on retire et ainsi de suite jusqu'à ce que le graphe soit décomposé en CFCs triviales.

Synthèse d'invariants, vérification On sait à présent comment analyser par interprétation abstraite les automates interprétés. L'application peut en être la synthèse d'invariants, numériques par exemple en utilisant un treillis abstrait comme celui des polyèdres.

Une autre application est bien entendu la vérification de propriétés. En introduction nous avons brièvement rappelé comment les propriétés de sûreté se ramenaient à la preuve de l'inaccessibilité d'un ensemble d'états Err.

L'inaccessibilité d'un point de contrôle va être repérée par $R_k^\# = \bot^\#$. L'analyse répond alors par « oui, la propriété est vérifiée » si l'ensemble des états de Err est inaccessible. Dans le cas contraire elle répond par « je ne peux conclure ».

On notera que l'échec n'est cependant pas irrémédiable. On peut en utilisant une analyse en *arrière* essayer d'affiner la sur-approximation, en calculant la suite $(X_n)_{n\geq 0}$ définie par

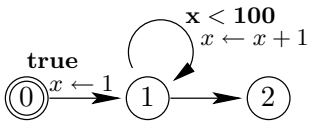
$$X_0 = \hat{\mathsf{Acc}}(Init), \quad Y_n = \mathsf{CoAcc}(Err \cap X_n) \quad X_{n+1} = \hat{\mathsf{Acc}}(Init \cap Y_n)$$

qui donne une sur-approximation de plus en plus précise tant que les X_n et les Y_n ne se stabilisent pas.

1.3.3 Exemple complet

Pour illustrer ce chapitre on donne un exemple complet de l'analyse du programme ci-dessous par interprétation abstraite sur le treillis abstrait des intervalles. On a $K = \{0, 1, 2\}$, $k_0 = 0, Id = \{x\}$ et on prend $\mathcal{N} = \mathbb{Z}$.

```
x := 1;
while(x<100) x++;
```



Si on note $a_1(\{\rho_v : x \to v\}) = \{x \to 1\}$ et $a_2(\{\rho_v : x \to v\}) = \{\rho'_v : x \to \rho_v(x) + 1\}$, la sémantique de collection de l'automate interprété modélisant ce programme est la suivante :

$$\begin{cases} R_0 = \{x \to v \mid v \in \mathbb{Z}\} \\ R_1 = a_1(R_0) \cup a_2(R_1 \cap (x < 100)) \\ R_2 = R_1 \cap (x \geq 100) \end{cases}$$

On peut alors définir le système d'équation de point fixe abstrait où $\oplus$ désigne le décalage d'un intervalle par un second. On remarquera que les gardes ont été remplacées par leur abstraction et que $a_1(R_0)$ a été remplacé par l'intervalle $[1,1]$ directement.

$$\begin{cases} R_0^\# = [-\infty, +\infty] = \top^I \\ R_1^\# = [1,1] \sqcup^I ((R_1^\# \sqcap^I [-\infty, 99]) \oplus [1,1]) \\ R_2^\# = R_1^\# \sqcap^I [100, +\infty] \end{cases}$$

Les étapes de la résolution du système sont données à la table 1.3. L'élargissement a été utilisé ($K_\nabla = \{1\}$) et permet dès la troisième itération d'obtenir un point fixe. C'est là que l'analyse descendante est appliquée et que les intersections avec l'intervalle $[-\infty, 99]$ pour $R_1^\#$ et $[100, +\infty]$ pour $R_2^\#$ améliorent considérablement le résultat : le point fixe final est le plus petit point-fixe.

	init.	*iter 1*	*iter 2*	*élargissement*	*stabilisation*	*séq. descendante*
R_0	$\top^I$	$\top^I$	$\top^I$	$\top^I$	$\top^I$	$\top^I$
R_1	$\bot^I$	$[1,1]$	$[1,2]$	$[1,+\infty]$	$[1,+\infty]$	$[1,99]$
R_2	$\bot^I$	$\bot^I$	$\bot^I$	$[100,+\infty]$	$[100,+\infty]$	$[100,+\infty]$

TAB. 1.3 – Calcul itératif du point fixe

1.3.4 Remarques

Nous clôturons ce chapitre sur l'état de l'art de l'interprétation abstraite, par quelques remarques sur certaines techniques pointues et approches nouvelles.

Test Parmi les travaux basés sur l'interprétation abstraite il faut noter ceux de Bourdoncle[6] qui introduit le formalisme intéressant d'*abstract testing* [Bou93] : il s'agit de trouver par interprétation abstraite et donc avant toute exécution, l'origine ou les conditions nécessaires d'un bug potentiel (*e.g.* prédire qu'une affectation d'une variable d'index peut produire un accès hors des limites d'un tableau) permettant ainsi de gérer efficacement l'occurence du bug lors de l'exécution.

Ceci n'est qu'un exemple où l'interprétation abstraite permet d'éprouver d'autres techniques de validation ou de vérification. Par exemple, les invariants trouvés par interprétation abstraite et surtout les invariants auxiliaires (non explicites dans le source du programme) sont très utiles pour prouver des prédicats inductifs qui pourront être injectés dans une tout autre technique de validation, comme le test guidé par la preuve (*c.f.* [RZC02]).

La combinaison inverse a été étudiée également, où des techniques de test sont appliquées en amont de l'analyse par interprétation abstraite pour extraire un sous-ensemble de la spécification et ainsi améliorer la précision et diminuer la complexité de l'analyse (*c.f.* [Rus02]).

Unions explicites Les opérations du domaine abstrait doivent être suffisamment bien étudiées afin de minimiser les pertes d'informations. Classiquement l'opérateur de borne supérieure perd de l'information, c'est à dire , si on note A le domaine abstrait et C le domaine concret

$$\gamma(x) \sqcup_C \gamma(y) \sqsubseteq_C \gamma(x \sqcup_A y)$$

[6] Qui rappelle très justement : « debugging is typically done "post-mortem", that is, after a bug has occured. »

Pour éviter cette perte d'information il est proposé de manipuler une union de valeurs abstraites plutôt qu'une seule pour représenter un ensemble d'états. Cette technique est par exemple utilisée dans l'outil KRONOS de vérification de systèmes temporisés, développé au laboratoire, où ils manipulent des unions explicites de DBMs. Le problème de cette technique est qu'elle augmente inévitablement la complexité de tous les opérateurs.

Partitionnement dynamique Proposé par Bertrand Jeannet [JHR99], le partitionnement dynamique est une technique permettant de raffiner la structure de contrôle sous analyse, selon la propriété à montrer, jusqu'à ce qu'elle soit assez détaillée pour permettre de conclure à la preuve de la dite propriété.

On peut ici faire une analogie avec les unions de valeurs abstraites, puisque finalement raffiner la structure revient à considérer un nombre croissant de valeurs abstraites.

Plus précisément, au départ la structure considérée est très grossière et ne distingue en fait que les états initiaux et les états d'erreur. Une série d'analyses en avant/en arrière est effectuée pour savoir si les états accessibles depuis les états initiaux sont « déconnectés » des états d'erreur. Si ce n'est pas le cas le résultat des analyses d'accessibilité et de co-accessibilité permet de scinder des états de la structure qui entraînent des approximations trop grossières *i.e.* approximant des états du programme qui n'ont pas le même comportement en terme d'accessibilité. Le processus est itéré jusqu'à ce que les états d'erreurs soient prouvés inaccessibles depuis les états initiaux, ou que la technique échoue, un partitionnement trop précis pouvant mené à une explosion exponentielle de l'analyse.

Chapitre 2

Le domaine abstrait IS

Dans ce chapitre nous définissons formellement notre domaine abstrait numérique pour l'analyse des entiers utilisés comme adresses.

À la section 2.1, nous définissons dans un premier temps, après discussion, les valeurs abstraites appropriées pour ce problème. L'idée initiale est de coupler le treillis abstrait des intervalles avec des propriétés de type égalités/non-egalités ($x = y$, $x \neq y$). Il a semblé judicieux, pour une complexité équivalente, d'ajouter les propriétés d'ordre ($x \leq y$, $x < y$).

Nous développons ensuite les solutions mises en oeuvre afin de traiter efficacement le problème de la canonisation des valeurs abstraites.

Enfin, nous dotons l'ensemble des valeurs abstraites d'une structure de treillis. Sont également définis l'opération d'élargissement et quelques opérateurs utiles à l'étude.

À la section 2.2, afin de pouvoir utiliser le cadre général de l'interprétation abstraite, nous définissons une connexion de Galois et une sémantique abstraite pour une classe de programmes manipulant des entiers.

2.1 Domaine abstrait IS

2.1.1 Point de départ

Lorsque l'on a envisagé la conception d'un domaine abstrait où l'on pourrait exprimer explicitement que deux variables numériques sont différentes, il a fallu considérer les informations nécessaires pour pouvoir inférer ces relations de non-égalité, qu'il faudrait alors inclure dans le domaine.

L'un des objectifs étant d'obtenir un domaine avec une faible complexité, nous avons considéré l'ajout d'un ensemble minimal de relations sur les variables qui est celui des comparaisons sur les opérateurs $<, \leq, =, \geq, >$ et choisi pour base de l'espace numérique le domaine des intervalles.

L'ensemble des contraintes qui composent le domaine est donc défini par la grammaire :

$$\mathit{contrainte} ::= \pm x \leq a \mid x - y \prec 0$$

où $\prec$ est un opérateur dans $\mathsf{S} = \{<, \leq, =, \neq, \geq, >\}$ et où a est une constante dans $\mathcal{V} = \overline{\mathbb{Z}}$, le complété de $\mathbb{Z}$ par $\{-\infty, +\infty\}$[1].

[1] Muni de l'ordre usuel, $\forall a \in \mathbb{Z}, \quad -\infty < a < +\infty$

Il s'agit donc d'un domaine 2-relationnel, que l'on ne peut comparer aux autres domaines numériques abstraits de par sa relation de non-égalité qu'aucun n'inclut. Si on oublie cette dernière on remarquera que le domaine proposé est un affaiblissement des zones (*c.f.* section 1.2.3) puisque l'on ne peut exprimer une différence de deux variables bornée par une constante quelconque.

2.1.2 Valeur abstraite

D'après ce qui précède, on peut définir naïvement une valeur abstraite comme le 7-uplet suivant :

$$(I, =, \neq, <, \leq, \geq, >)$$

où la fonction $I : Id \to \mathcal{V}^2$ associe à une variable un intervalle et où les six autres composantes sont des relations dans $(Id \times Id)$, qui indiquent comme on s'y attend si deux variables satisfont la dite relation.

Un travail préliminaire sur ces valeurs abstraites est de cerner l'ensemble des règles de correction qui s'imposent entre les différentes contraintes que l'on peut exprimer,

Notations On notera que l'ensemble des relations de deux variables sur S peut être ordonné selon une structure de treillis. Ce treillis, que nous nommerons *treillis des comparaisons* est donné à la figure 2.1.

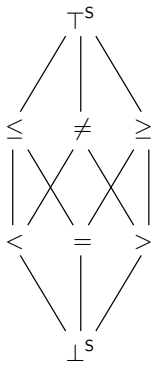


FIG. 2.1 – Treillis des comparaisons $L^{\mathsf{S}} = (\mathsf{S} \cup \{\bot^{\mathsf{S}}, \top^{\mathsf{S}}\}, \sqsubseteq^{\mathsf{S}}, \sqcup^{\mathsf{S}}, \sqcap^{\mathsf{S}})$

Intuitivement le plus grand élément $\top^{\mathsf{S}}$ exprime que l'on n'a pas d'information relationnelle sur les deux variables, alors que le plus petit élément $\bot^{\mathsf{S}}$ exprime une comparaison impossible. Sur le dessin donc, l'opération de borne inférieure, notée $\sqcap^{\mathsf{S}}$, consiste à prendre le plus grand descendant commun et l'opération de borne supérieure, notée $\sqcup^{\mathsf{S}}$ consiste à prendre le plus petit ancêtre commun.

On note $\mathsf{S}^{\bowtie} = \{<, \leq, \geq, >\}$ et on définit les fonctions $\mathsf{lb} : (L^I \setminus \bot^I) \to \mathcal{V}$ et $\mathsf{ub} : (L^I \setminus \bot^I) \to \mathcal{V}$ qui renvoient respectivement la borne inférieure et la borne supérieure d'un intervalle. On définit également Singl le sous-ensemble des intervalles singletons. Enfin, le treillis des intervalles est noté $L^I = (\mathcal{V}, \bot^I, \top^I, \sqsubseteq^I, \sqcup^I, \sqcap^I))$.

Règles de correction On peut à présent énoncer les différentes règles qui régissent une valeur abstraite.

Parmi les quatre premières, [Order] et [Meet1,2] obligent les relations à *respecter la structure du treillis des comparaisons*. Par exemple si $j < i$ alors on doit avoir $j \leq i$ et $j \neq i$. La partie droite de la règle [Eq] exprime l'égalité composante par composante du 7-uplet.

[Order] $\forall \mathsf{s} \in \mathsf{S} \quad j\ \mathbf{s}\ i \Rightarrow \forall \mathsf{s}' \in \mathsf{S},\ \mathsf{s}' \sqsupseteq^{\mathsf{S}} s \quad j\ \mathbf{s'}\ i$
[Meet1] $\forall \mathsf{s}, \mathsf{s}' \in \mathsf{S} \quad \mathsf{s}'' = \mathsf{s} \sqcap^{\mathsf{S}} \mathsf{s}' \quad \mathsf{s}'' \in \mathsf{S} \quad l\ \mathbf{s}\ i \wedge l\ \mathbf{s'}\ i \Leftrightarrow l\ \mathbf{s''}\ i$
[Meet2] $\forall \mathsf{s}, \mathsf{s}' \in \mathsf{S} \quad \bot^{\mathsf{S}} = \mathsf{s} \sqcap^{\mathsf{S}} \mathsf{s}' \quad \neg(l\ \mathbf{s}\ i \wedge l\ \mathbf{s'}\ i)$
[Eq] $j = i \Leftrightarrow (\forall \mathsf{s} \in \mathsf{S} \quad l\ \mathbf{s}\ i \Leftrightarrow l\ \mathbf{s}\ j) \wedge (I(i) = I(j))$

Les règles suivantes sont *intrinsèques aux relations* que l'on considère. La règle $i = i$ que l'on attendait est obtenue par [Rfl1,2] et [Meet1]. On notera les règles [TransNeq1,2] qui auront plus d'intérêt par la suite.

[Neq] $\neg(i \neq i)$
[Rfl1] $i \leq i$
[Rfl2] $j \leq i \Leftrightarrow i \geq j$
[AntiRfl1] $j \neq i \Leftrightarrow i \neq j$
[AntiRfl2] $j < i \Leftrightarrow i > j$
[TransEq] $\forall \mathsf{s} \in \{\leq, \geq\} \quad l\ \mathbf{s}\ j \wedge j\ \mathbf{s}\ i \Rightarrow l\ \mathbf{s}\ i$
[TransNeq1] $\forall \mathsf{s} \in \{<, \leq\} \quad l\ \mathbf{s}\ j \wedge j < i \Rightarrow l < i$
[TransNeq2] $l < j \wedge j\ \leq\ i \Rightarrow l < i$

Les deux règles suivantes dictent *ce que les comparaisons sur les variables imposent sur leurs intervalles* (préfixées par C2I pour « comparison to interval »).

Ce que dit la règle [C2I] en substance c'est qu'une relation entre deux variables, par exemple $i \leq j$ impose que pour toute valeur possible de j on puisse choisir un i tel que la relation soit respectée, donc la plus petite valeur de j doit être supérieure à celle de i (de même pour la plus grande valeur).

La règle [C2Ineq] exprime que lorsque $i \neq j$ et que j a pour intervalle un singleton $[a, a]$, alors aucune borne de l'intervalle de i ne doit être égale à a.

[C2I] $\forall \mathsf{s} \in \mathsf{S}^{\bowtie} \quad i\ \mathbf{s}\ j \Rightarrow (\mathsf{lb}(I(i))\ \mathsf{s}\ \mathsf{lb}(I(j)) \wedge (\mathsf{ub}(I(i))\ \mathsf{s}\ \mathsf{ub}(I(j))$
[C2Ineq] $i \neq j \quad I(j) \in \mathsf{Singl} \Rightarrow I(i) \sqcap^{I} I(j) \neq \mathsf{lb}(I(i)) \neq \mathsf{ub}(I(i))$

Restent les règles qui régissent *ce que les intervalles des variables imposent comme relations* (préfixées I2C). Intuitivement on ne peut inférer une relation que si deux intervalles sont disjoints (règle [I2Cempty]) ou s'ils ont *une* valeur en commun (règle [I2Csingl]) : l'une des variables est alors plus grande (strictement dans le premier cas) que l'autre.

[I2Ceq] $I(i) = I(j) \in \mathsf{Singl} \Rightarrow i = j$
[I2Cempty] $I(i) \sqcap^{I} I(j) = \bot^{I} \Rightarrow \begin{cases} (\mathsf{ub}(I(i)) < \mathsf{lb}(I(j)) \wedge i < j) \\ \vee\ (\mathsf{lb}(I(i)) > \mathsf{ub}(I(j)) \wedge i > j) \end{cases}$
[I2Csingl] $I(i) \sqcap^{I} I(j) \in \mathsf{Singl} \Rightarrow \begin{cases} (\mathsf{ub}(I(i)) = \mathsf{lb}(I(j)) \wedge i \leq j) \\ \vee\ (\mathsf{lb}(I(i))\ =\ \mathsf{ub}(I(j)) \wedge i \geq j) \end{cases}$

Simplification Naturellement on voit que l'on peut restreindre les relations nécessaires à la représentation de nos valeurs abstraites. Bien sûr, des deux relations $<$ et $>$ on en gardera qu'une seule, de même pour $\leq$ et $\geq$ (règles [AntiRfl1,2]). En gardant $<$ et $\leq$ on définit

$$> = \{(i, j) \mid j < i\} \qquad \geq = \{(i, j) \mid j \leq i\} \tag{2.1}$$

En fait, les quatre relations restantes $(=, \neq, <, \leq)$ peuvent être exprimées par l'un des deux

couples de relations suivant : $(\neq, \leq)$ ou $(=, <)$. Le premier peut être obtenu par disjonction du second

$$\leq = \{(i,j) \mid i < j \vee i = j\} \qquad \neq = \{(i,j) \mid i < j \vee i > j\} \tag{2.2}$$

et le second par conjonction du premier

$$< = \{(i,j) \mid i \leq j \wedge i \neq j\} \qquad = = \{(i,j) \mid i \leq j \wedge i \geq j\} \tag{2.3}$$

ce qui se voit aisément sur le treillis des comparaisons.

Les deux solutions ne sont pas symétriques : le couple de relations $(=, <)$ pose un problème important. En effet la relation $\leq$ étant codée par une disjonction, on ne peut exprimer l'information « j est inférieur à i », *i.e.* on ne sait pas si c'est égal ou inférieur strict, sans exprimer que $j < i$ ou $j = i$, alors que ceci n'est pas déterminé[2]. Pour palier à cet inconvénient on pourrait imposer que si $j < i$ *et* $j = i$ on entend que j est inférieur à i, mais cela ne peut-être une solution satisfaisante.

Ces problèmes n'affectent pas la seconde solution, puisque d'après la règle [Meet1] il y a équivalence entre la conjonction de deux comparaisons et la borne inférieure de ces deux comparaisons.

On choisit donc pour représenter nos valeurs abstraites le 3-uplet $(I, \neq, \leq)$, les autres composantes étant données par les équations (2.1) et (2.3).

Règles définitives On notera que certaines des règles de correction deviennent caduques dans cette représentation (règles [Order], [Meet1], [Rfl2] et [AntiRfl2]) d'autres sont affaiblies (la règle [Eq] où l'égalité des relations peut se vérifier uniquement sur l'ensemble $\mathsf{S}^{\#} = \{\neq, \leq\}$ et non tout S) et d'autres peuvent être réécrites (la règle [Meet2] impose par exemple $\neg(i = j \wedge i < j)$ qui peut se réécrire sur $\mathsf{S}^{\#}$).

L'ensemble de règles de correction final obtenu est donné à la table 2.1.

[Meet] $\neg(j \neq i \wedge j \leq i \wedge i \leq j)$
[Eq] $j = i \Leftrightarrow (\forall \mathsf{s} \in \mathsf{S}^{\#} \quad l\ \mathbf{s}\ i \Leftrightarrow l\ \mathbf{s}\ j) \wedge (I(i) = I(j))$
[Neq] $\neg(i \neq i)$
[Rfl] $i \leq i$
[AntiRfl] $j \neq i \Leftrightarrow i \neq j$
[TransEq] $\forall \mathsf{s} \in \{\leq, \geq\} \quad l\ \mathbf{s}\ j \wedge j\ \mathbf{s}\ i \Rightarrow l\ \mathbf{s}\ i$
[TransNeq1] $\forall \mathsf{s} \in \{<, \leq\} \quad l\ \mathbf{s}\ j \wedge j < i \Rightarrow l < i$
[TransNeq2] $l < j \wedge j \ \leq\ i \Rightarrow l < i$
[C2I] $\forall \mathsf{s} \in \mathsf{S}^{\bowtie} \quad i\ \mathbf{s}\ j \Rightarrow (\mathsf{lb}(I(i))\ \mathsf{s}\ \mathsf{lb}(I(j)) \wedge (\mathsf{ub}(I(i))\ \mathsf{s}\ \mathsf{ub}(I(j))$
[C2Ineq] $i \neq j \quad I(j) \in \mathsf{Singl} \Rightarrow I(i) \sqcap^I I(j) \neq \mathsf{lb}(I(i)) \neq \mathsf{ub}(I(i))$
[I2Ceq] $I(i) = I(j) \in \mathsf{Singl} \Rightarrow i = j$
[I2Cempty] $I(i) \sqcap^I I(j) = \bot^I \Rightarrow \begin{cases} (\mathsf{ub}(I(i)) < \mathsf{lb}(I(j)) \wedge i < j) \\ \vee\ (\mathsf{lb}(I(i)) > \mathsf{ub}(I(j)) \wedge i > j) \end{cases}$
[I2Csingl] $I(i) \sqcap^I I(j) \in \mathsf{Singl} \Rightarrow \begin{cases} (\mathsf{ub}(I(i)) = \mathsf{lb}(I(j)) \wedge i \leq j) \\ \vee\ (\mathsf{lb}(I(i)) \ = \ \mathsf{ub}(I(j)) \wedge i \geq j) \end{cases}$

TAB. 2.1 – Règles de correction d'une valeur abstraite

[2]Le problème est identique pour $\neq$: on peut savoir que deux variables ont une valeur différente sans savoir laquelle est strictement supérieure à l'autre.

Représentation par fonctions On souhaite plutôt représenter les relations par des fonctions qui nous permettront de connaître directement pour une variable i l'ensemble des variables j tel que (i, j) appartienne à la relation.

On définit à présent les valeurs abstraites du domaine IS .

Définition 7 (valeur abstraite). *Soit $L^\#$ l'ensemble des valeurs abstraites. Une valeur abstraite $\mathcal{A} \in L^\#$ définie sur l'ensemble de variable Id à valeur dans $\mathcal{V}$ est un 3-uplet de fonctions*

$$\mathcal{A} = (int_\mathcal{A},\ neq_\mathcal{A},\ leq_\mathcal{A})$$

où

- *la fonction $int_\mathcal{A} : Id \to \mathcal{V}^2$ totale associe un* intervalle *dans $\mathcal{V}$ à une variable.*
- *la fonction $neq_\mathcal{A} : Id \to \mathcal{P}(Id)$ associe l'ensemble des variables* non-égales *à une variable.*
- *la fonction $leq_\mathcal{A} : Id \to \mathcal{P}(Id)$ associe l'ensemble des variables* inférieures *à une variable.*

On donne également le plus petit élément du domaine, noté $\bot^\#$, et on définit le plus grand élément, $\top^\# = (\lambda i.\top^I, \lambda i.\emptyset, \lambda i.\{i\})$.

Une dernière de nos considérations sur la définition des valeurs abstraites fut le regroupement par classes d'égalité. Jusqu'alors on considère pour *chaque* variable son intervalle et les ensembles de variables avec lesquels elle est en relation. Or la règle [Eq] nous permettrait de considérer ensemble les variables égales au sein de *classes*. Une valeur abstraite serait alors un ensemble de classes avec les contraintes exprimées entre celles-ci.

Cette approche n'a pas été retenue. Il s'est avéré qu'elle compliquait les formulations mathématiques sans pour autant constituer une différence fondamentale et qu'il s'agissait donc juste d'une question d'optimisation de l'implémentation des valeurs abstraites.

Un problème classique qui se pose pour les valeurs abstraites est celle de leur comparaison. La notion de forme canonique reste donc à définir. C'est l'objet de la section suivante.

2.1.3 Représentation canonique

Canoniser une valeur abstraite peut-être vu comme tirer toutes les conséquences de l'information qu'elle contient. Ceci peut aller jusqu'à la conclusion que ces informations sont contradictoires et le résultat est alors $\bot^\#$. Il s'avère que ce maximum informatif est exactement imposé par les règles de correction, principalement par les règles [C2I∼] et [I2C∼]. Les premières restreignent les intervalles et les secondes amènent des contraintes supplémentaires.

Définition 8 (représentation canonique). *Soit $\mathcal{A} = (int_\mathcal{A},\ neq_\mathcal{A},\ leq_\mathcal{A})$ une valeur abstraite. On définit la fonction $I = int_\mathcal{A}$ et les deux relations*

$$\neq = \{(j, i) \mid j \in neq_\mathcal{A}\} \qquad \leq = \{(j, i) \mid j \in leq_\mathcal{A}\}$$

Alors la représentation de $\mathcal{A}$ est canonique *ssi le 3-uplet $(I, \neq, \leq)$ respecte les règles de correction de la table 2.1.*

Il faut ajouter que cette représentation est canonique si on suppose une représentation canonique de nos ensembles.

2.1.3.1 Canonisation

Nous décrivons la canonisation des valeurs abstraites par un *système de réécriture*. Cette approche permet de prouver aisément que le résultat de la canonisation est bien une valeur abstraite correcte, puisque intuitivement les règles de réécriture (ou *réductions*) sont directement liées aux règles de correction.

Il ne s'agit pas d'une simple traduction des unes vers les autres. Ce que les réductions doivent donner ce sont des solutions par des modifications sur les intervalles et les relations pour qu'une règle enfreinte devienne correcte. Par exemple, si la règle [Neq] impose trivialement une règle réécrivant la valeur abstraite en la valeur $\bot^\#$, la règle [C2I] elle n'exprime pas quelles valeurs doivent prendre les intervalles si elle n'est pas vérifiée.

Les réductions ont été rassemblées selon les modifications qu'elles effectuent sur les valeurs abstraites. Des commentaires accompagnent chacun de ces ensembles de règles de réécriture, donnés dans les tables suivantes :

- Table 2.2 *(vide)* règles réécrivant la valeur abstraite en $\bot^\#$
- Table 2.6 *(consistance)* règles nécessaires mais n'apportant pas d'information.
- Table 2.5 *(intervalle)* règles affectant l'intervalle d'une variable
- Table 2.3 *(non-égalité)* règles ajoutant une contrainte de non-égalité entre deux variables
- Table 2.4 *(inférieur)* règles ajoutant une contrainte d'inégalité entre deux variables

Notations Pour aider à la lecture des règles on introduit une notation graphique des réductions. On note respectivement la relation $j \leq i$ et la relation $j \neq i$ par :

$$i \longrightarrow\!\!\!\!\rhd j \qquad\qquad i \longrightarrow j$$

Les intervalles sont notés par des segments. Le résultat de la réduction est mis en évidence par un trait d'épaisseur double.

On introduit également

- $\mathcal{A}_{|i}$ la restriction de $\mathcal{A}$ à la variable i, *i.e.* le 3-uplet $(int_\mathcal{A}(i), neq_\mathcal{A}(i), leq_\mathcal{A}(i))$. L'égalité sur $(\mathcal{V}^2, \mathcal{P}(Id), \mathcal{P}(Id))$ est définie par l'égalité sur les intervalles et sur les ensembles.
- $\mathcal{A}_{leq}[E/i]$ la subsitution dans $\mathcal{A}$ de l'ensemble $leq_\mathcal{A}(i)$ par E. Si $\mathcal{B}$ est son résultat on a,

$$\begin{array}{rcl} int_\mathcal{B} & = & int_\mathcal{A} \\ neq_\mathcal{B} & = & neq_\mathcal{A} \\ leq_\mathcal{B}(j) & = & \left\{ \begin{array}{ll} leq_\mathcal{A}(j) & \text{si } j \neq i \\ E & \text{sinon} \end{array} \right. \end{array}$$

On définit identiquement les substitutions $\mathcal{A}_{neq}[E/i]$ et $\mathcal{A}_{int}[[a,b]/i]$.

Règles de réécritures Les deux premières règles suffisent à capturer les cas où l'information contenue dans une valeur abstraite est vide. Bien que non explicité, le cas de la règle [Meet] est bien pris en compte comme nous allons le voir.

$$\frac{i \in neq_\mathcal{A}(i)}{\mathcal{A} \to \bot^\#}\to_{bot\_neq} \qquad\qquad \frac{int_\mathcal{A}(i) = \bot^I}{\mathcal{A} \to \bot^\#}\to_{bot\_int}$$

TAB. 2.2 – Canonisation *(vide)*

Fusion Lorsque $j \leq i$ et $i \leq j$, *i.e.* lorsque $i = j$, on doit naturellement avoir $\mathcal{A}_{|i} = \mathcal{A}_{|j}$ (règle [Eq]). Si ce n'est pas le cas, il faut mettre en commun l'information apportée par chacune des variables. Ceci se traduit sur les intervalles par leur intersection et sur les relations par leur union[3]. Cette opération est appelée *fusion*. Si on note $I^{fus} = int_{\mathcal{A}}(i) \cap int_{\mathcal{A}}(j)$, $N^{fus} = neq_{\mathcal{A}}(i) \cup neq_{\mathcal{A}}(j)$ et $L^{fus} = leq_{\mathcal{A}}(i) \cup leq_{\mathcal{A}}(j)$ on peut définir la valeur abstraite $fusion_{\mathcal{A}}(i,j)$ où i et j sont fusionnés dans $\mathcal{A}$:

$\mathcal{B} = \mathcal{A}_{int}[I^{fus}/i][I^{fus}/j]$
$\mathcal{C} = \mathcal{B}_{neq}[N^{fus}/i][N^{fus}/j]$
$fusion_{\mathcal{A}}(i,j) = \mathcal{C}_{leq}[L^{fus}/i][L^{fus}/j]$

Ainsi on définirait la réduction suivante

$$\frac{j \in leq_{\mathcal{A}}(i) \wedge i \in leq_{\mathcal{A}}(j) \quad \mathcal{A}_{|j} \neq \mathcal{A}_{|i}}{\mathcal{A} \rightarrow fusion_{\mathcal{A}}(i,j)}$$

Si on en revient à la prise en compte de la règle [Meet], on voit que si $i \neq j$ et $i = j$ l'opération de fusion va engendrer $i \neq i$ impliquant la réécriture de la valeur abstraite en $\bot^{\#}$ ce qui est le résultat souhaité.

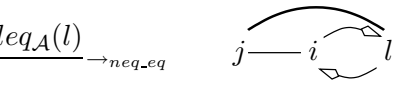
On note que la condition $\mathcal{A}_{|j} \neq \mathcal{A}_{|i}$ de la règle précédemment proposée peut-être morcelée selon si c'est les intervalles ou les ensembles des relations qui diffèrent. Il s'avère même que seul le cas de la relation de non-égalité n'est pas couvert par d'autres règles. En effet grâce au codage de l'égalité de i et j par la conjonction $i \leq j \wedge j \leq i$,

- les règles $\rightarrow_{int\_leq}$ et $\rightarrow_{int\_geq}$ données plus loin (table 2.5) effectuent l'intersection des intervalles des deux variables égales.
- la règle $\rightarrow_{leq\_trans}$ qui est la règle de transitivité de la relation inférieur (table 2.4) effectue la mise en commun des variables inférieures aux deux variables égales.

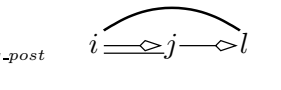
Cette règle peut donc être abandonnée en définissant la règle $\rightarrow_{neq\_eq}$ qui va traiter le cas particulier de la fusion des variables différentes aux deux variables égales.

On donne les réductions qui font apparaître une relation de non-égalité. La première est celle que l'on vient de définir. Les trois autres découlent directement des règles de correction [TransNeq1,2] et [I2Cempty].

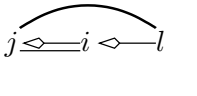
$$\frac{j \in neq_{\mathcal{A}}(i) \wedge j \notin neq_{\mathcal{A}}(l) \quad l \in leq_{\mathcal{A}}(i) \wedge i \in leq_{\mathcal{A}}(l)}{\mathcal{A} \rightarrow \mathcal{A}_{neq}[neq(l) \cup \{j\}/l]} \rightarrow_{neq\_eq}$$



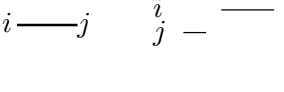
$$\frac{j \in leq_{\mathcal{A}}(i) \cap neq_{\mathcal{A}}(i) \quad l \in leq_{\mathcal{A}}(j) \wedge l \notin neq_{\mathcal{A}}(i)}{\mathcal{A} \rightarrow \mathcal{A}_{neq}[neq_{\mathcal{A}}(i) \cup \{l\}/i]} \rightarrow_{neq\_post}$$



$$\frac{j \in leq_{\mathcal{A}}(i) \cap neq_{\mathcal{A}}(i) \quad i \in leq_{\mathcal{A}}(l) \wedge j \notin neq_{\mathcal{A}}(l)}{\mathcal{A} \rightarrow \mathcal{A}_{neq}[neq_{\mathcal{A}}(l) \cup \{j\}/l]} \rightarrow_{neq\_pre}$$



$$\frac{int_{\mathcal{A}}(i) \sqcap^{I} int_{\mathcal{A}}(j) = \bot^{I} \quad j \notin neq_{\mathcal{A}}(i)}{\mathcal{A} \rightarrow \mathcal{A}_{neq}[neq_{\mathcal{A}}(i) \cup \{j\}/i]} \rightarrow_{neq\_empty}$$



TAB. 2.3 – Canonisation *(non-égalité)*

[3] L'union souhaitée est celle sur toutes les relations *i.e.* toute relation de S. Or ici l'union est effectuée seulement sur $\mathsf{S}^{\#} = \{\neq, \leq\}$. Il s'avère que cette dernière est correcte : *c.f.* page 32 pour une preuve

Les réductions suivantes sont également sans surprise. On remarquera cependant deux points sur la réduction $\rightarrow_{leq\_singl}$

- si deux variables ont pour intervalles des singletons égaux (règle [I2Ceq]), une fusion des deux variables est bien effectuée : la réduction $\rightarrow_{leq\_singl}$ permet en effet d'inférer $j \leq i$ et $i \leq j$.
- si $i \neq j$ et que j a pour intervalle un singleton qui est une des bornes de l'intervalle de i (règle [C2Ineq]) alors la réduction $\rightarrow_{leq\_singl}$ va inférer $j \leq i$ (ou $i \leq j$) rendant possible la réduction $\rightarrow_{int\_neq\_leq}$ (ou $\rightarrow_{int\_neq\_leq}$) qui va repousser la borne inférieure (supérieure) de l'intervalle i, puisque la valeur de i est différente de celle de j qui est déterminée.

$$\frac{j \in leq_{\mathcal{A}}(i) \ \wedge \ l \in leq_{\mathcal{A}}(j) \ \wedge \ l \notin leq_{\mathcal{A}}(i)}{\mathcal{A} \rightarrow \mathcal{A}_{leq}[leq_{\mathcal{A}}(i) \cup \{l\}/i]} \rightarrow_{leq\_trans} \qquad i \longrightarrow j \longrightarrow l$$

$$\frac{int_{\mathcal{A}}(i) \sqcap^I int_{\mathcal{A}}(j) = \bot^I \qquad \mathsf{ub}(int_{\mathcal{A}}(j)) < \mathsf{lb}(int_{\mathcal{A}}(i)) \quad j \notin leq_{\mathcal{A}}(i)}{\mathcal{A} \rightarrow \mathcal{A}_{leq}[leq_{\mathcal{A}}(i) \cup \{j\}/i]} \rightarrow_{leq\_empty} i \longrightarrow j \qquad \begin{matrix} i \\ j \end{matrix} \ \_$$

$$\frac{int_{\mathcal{A}}(i) \cap int_{\mathcal{A}}(j) \in \mathsf{Singl} \quad \mathsf{ub}(int_{\mathcal{A}}(j)) = \mathsf{lb}(int_{\mathcal{A}}(i)) \quad j \notin leq_{\mathcal{A}}(i)}{\mathcal{A} \rightarrow \mathcal{A}_{leq}[leq_{\mathcal{A}}(i) \cup \{j\}/i]} \rightarrow_{leq\_singl} i \longrightarrow j \qquad \begin{matrix} i \\ j \end{matrix} \ \_$$

TAB. 2.4 – Canonisation *(inférieur)*

Cas arithmétique Il reste à définir les réductions portant sur les intervalles. Si lorsque $j \leq i$ il est aisé de voir que la borne inférieure de l'intervalle de i doit être le maximum des bornes inférieures des intervalles de i et j, le cas strict *i.e.* lorsque $j < i$ pose problème.

En effet si la borne inférieure de i devait être modifiée, elle s'exprimerait par une borne ouverte, *i.e.* que le nouvel intervalle de i serait $]\mathsf{lb}(int_{\mathcal{A}}(j), \mathsf{ub}(int_{\mathcal{A}}(i))]$. Hors cela ne respecte pas notre définition des valeurs abstraites.

Cependant, les propriétés que l'on souhaite vérifier portant sur des variables numériques entières, on réécrit l'intervalle en reportant la borne à l'entier suivant. [4].

$$\frac{j \in leq_{\mathcal{A}}(i) \qquad \mathsf{lb}(int_{\mathcal{A}}(j)) > \mathsf{lb}(int_{\mathcal{A}}(i))}{\mathcal{A} \rightarrow \mathcal{A}_{int}[[\mathsf{lb}(int_{\mathcal{A}}(j)), \mathsf{ub}(int_{\mathcal{A}}(i))]/i]} \rightarrow_{int\_leq} \qquad i \longrightarrow j \qquad \begin{matrix} i \\ j \end{matrix} \ \_\_$$

$$\frac{j \in leq_{\mathcal{A}}(i) \qquad \mathsf{ub}(int_{\mathcal{A}}(j)) > \mathsf{ub}(int_{\mathcal{A}}(i))}{\mathcal{A} \rightarrow \mathcal{A}_{int}[[\mathsf{lb}(int_{\mathcal{A}}(j)), \mathsf{ub}(int_{\mathcal{A}}(i))]/j]} \rightarrow_{int\_geq} \qquad i \longrightarrow j \qquad \begin{matrix} i \\ j \end{matrix} \ \_\_$$

$$\frac{j \in leq_{\mathcal{A}}(i) \cap neq_{\mathcal{A}}(i) \qquad \mathsf{lb}(int_{\mathcal{A}}(j)) \geq \mathsf{lb}(int_{\mathcal{A}}(i))}{\mathcal{A} \rightarrow \mathcal{A}_{int}[[\mathsf{lb}(int_{\mathcal{A}}(j)) + 1, \mathsf{ub}(int_{\mathcal{A}}(i))]/i]} \rightarrow_{int\_neq\_leq} \qquad i \Longrightarrow j \qquad \begin{matrix} i \\ j \end{matrix} \ \_\_$$

$$\frac{j \in leq_{\mathcal{A}}(i) \cap neq_{\mathcal{A}}(i) \qquad \mathsf{ub}(int_{\mathcal{A}}(j)) \geq \mathsf{ub}(int_{\mathcal{A}}(i))}{\mathcal{A} \rightarrow \mathcal{A}_{int}[[\mathsf{lb}(int_{\mathcal{A}}(j)), \mathsf{ub}(int_{\mathcal{A}}(i)) - 1]/j]} \rightarrow_{int\_neq\_geq} \qquad i \Longrightarrow j \qquad \begin{matrix} i \\ j \end{matrix} \ \_\_$$

TAB. 2.5 – Canonisation *(intervalle)*

[4] Il ne s'agit pas de se restreindre à une analyse en nombre entier qui serait beaucoup trop coûteuse. Ici on effectue simplement un raffinement lorsque l'on a connaissance que les variables sont entières. Notons que le test du vide d'une valeur abstraite dans le cas arithmétique utilisant la non-égalité est un problème **NP**-complet.

Enfin on donne les deux dernières réductions qui assurent l'anti-reflexivité de $\neq$ et la réflexivité de $\leq$.

$$\frac{j \in neq_{\mathcal{A}}(i) \wedge i \notin neq_{\mathcal{A}}(j)}{\mathcal{A} \to \mathcal{A}_{neq}[neq_{\mathcal{A}}(j) \cup \{i\}/j]} \to_{arfl} \qquad \frac{i \notin leq_{\mathcal{A}}(i)}{\mathcal{A} \to \mathcal{A}_{leq}[leq_{\mathcal{A}}(i) \cup \{i\}/i]} \to_{rfl}$$

TAB. 2.6 – Canonisation *(consistance)*

2.1.3.2 Terminaison et forte normalisation

Notre système de réécriture ayant pour but de réécrire un terme en sa forme canonique, il nous faut prouver que la procédure termine et surtout que, quel que soit l'ordre d'application des réductions, elle converge toujours vers le *même* terme. Nous montrerons alors que ce terme est bien la forme canonique que nous attendions.

Le lecteur attentif aura remarqué que nos réductions ne sont pas des règles de réécriture au sens habituel : ce sont des règles de réécriture conditionnelles. En effet nos règles portent des pré-conditions, ce qui fait qu'elles sont dépendantes du contexte et que donc suivant l'ordre dans lequel on les applique on considère des règles différentes.

On peut alors s'interroger sur le fait que les théorèmes connus soient encore valide. Par exemple si on considérait un contexte avec des références vers les variables il est certain qu'on ne pourrait plus rien dire sur la confluence* du système.

Cependant, nos réductions s'apparentent à de la réécriture de graphes et le contexte n'est pas « volatile ». En effet, on voit aisément que toute réduction activée[5] sera appliquée ultimement (ou une règle engendrant une modification identique).

Terminaison. On montre d'abord la terminaison* du système de réécriture.

Soit $N = 2|Id|^2$ le maximum possible de la somme des cardinaux des ensembles *neq* et *leq* d'une valeur abstraite $\mathcal{A}$ (le pire cas est en effet la valeur abstraite $\bot^\#$ exprimée avec $\forall i\ neq(i) = Id$ et $leq(i) = Id$).

Soit $Id^+ = \{i \in Id \mid \mathsf{ub}(int_{\mathcal{A}}(i)) = +\infty\}$ et $Id^- = \{i \in Id \mid \mathsf{lb}(int_{\mathcal{A}}(i)) = -\infty\}$

Soit $>_{\mathbb{N}\times\mathbb{N}\times\mathbb{N}\times\mathbb{N}}$ l'ordre lexicographique sur $(\mathbb{N} \times \mathbb{N} \times \mathbb{N} \times \mathbb{N})$, qui est noeuthérien, et la fonction de plongement ϕ des valeurs abstraites dans cet ordre :

$$\begin{aligned}
\phi(\mathcal{A}) &= \Big(|Id^-|, |Id^+|, \\
&\qquad \sum_{i \in Id \setminus (Id^- \cup Id^+)} (|\mathsf{ub}(int_{\mathcal{A}}(i)) - \mathsf{lb}(int_{\mathcal{A}}(i))|), \\
&\qquad N - \sum_{i \in Id} (|leq_{\mathcal{A}}(i)|) - \sum_{i \in Id} (|neq_{\mathcal{A}}(i)|)\Big) \\
\phi(\bot^\#) &= (0, 0, 0, 0)
\end{aligned}$$

Ce plongement est une preuve de terminaison du système de réécriture car on a $(\mathcal{A} \longrightarrow \mathcal{B}) \Rightarrow (\phi(\mathcal{A}) >_{\mathbb{N}\times\mathbb{N}\times\mathbb{N}\times\mathbb{N}} \phi(\mathcal{B}))$. En effet à chaque réduction soit une borne d'un intervalle est ramenée

[5] C'est à dire dont la pré-condition est vérifiée

(de +/-∞ faisant décroître $|Id^{+/-}|$ ou d'une constante n faisant décroître la somme des surfaces des intervalles non infinis), soit les intervalles ne sont pas modifiés et le nombre de relations $\neq$ ou $\leq$ augmente (faisant décroître la somme des cardinaux des ensembles *neq* et *leq*). □

Confluence. Le système étant noethérien on peut ramener la preuve de la propriété de confluence à la preuve de la propriété de confluence locale* par le lemme de Newman* .

D'après le théorème de Knuth-Bendix* , pour prouver la confluence locale il suffit de montrer que toute paire critique* est joignable.[6]

Nous allons montrer que toute paire critique de notre système est joignable. L'idée est que toute règle activée sera soit immanquablement appliquée, soit une autre appliquera une transformation plus restrictive, ce qui intuitivement assure toujours la confluence.

Plus formellement, soit $\mathcal{C}$ une valeur abstraite, et $\rightarrow_a$ une règle réduisant $\mathcal{C}$ en $\mathcal{A}$ et $\rightarrow_b$ une règle réduisant $\mathcal{C}$ en $\mathcal{B}$. La paire $(\mathcal{A}, \mathcal{B})$ est une paire critique.

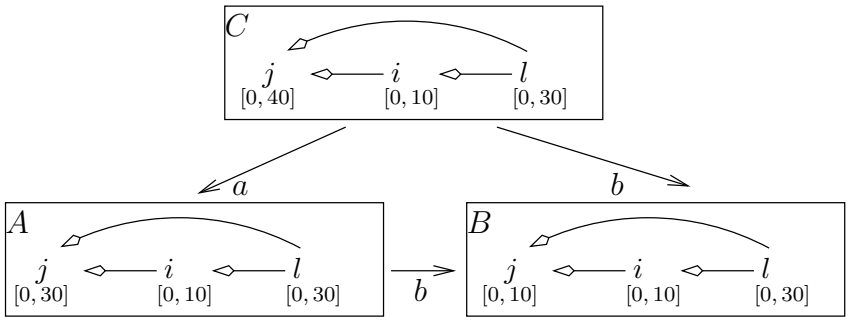
On veut montrer que dans la réduction de $\mathcal{A}$ la règle $\rightarrow_b$ s'appliquera et dans la réduction de $\mathcal{B}$ la règle $\rightarrow_a$. Ainsi leur réductions mèneront à un même terme, $\mathcal{A}$ et $\mathcal{B}$ seront joignables.

Prenons le cas de $\mathcal{A}$. Si la règle $\rightarrow_b$ devait être « désactivée » c'est qu'une autre règle aurait soit,

- ajouter la relation que $\rightarrow_b$ devait ajouter. Aucune relation ne pouvant être retirée la pré-condition de $\rightarrow_b$ n'a pu devenir fausse qu'ainsi. (la relation ajoutée par une règle est toujours imposée non existante dans la pré-condition d'une règle). Ceci revient à appliquer $\rightarrow_b$.
- plus décaler la borne d'un intervalle que ce que $\rightarrow_b$ devait faire. Pour les mêmes raisons la pré-condition de $\rightarrow_b$ n'est devenue fausse qu'à cause de ce décalage plus important. Donc là encore $\rightarrow_b$ a été appliquée.

□

Exemple 1. *Ces propos sont illustrés par l'exemple suivant où l'on a la paire critique $\mathcal{A}, \mathcal{B}$:*



La réduction $\rightarrow_a$ a été désactivée dans $\mathcal{B}$ (qui est canonique). La réduction $\rightarrow_a$ était plus « faible » que $\rightarrow_b$ et donc s'applique dans $\mathcal{A}$, joignant ainsi la paire critique en $\mathcal{B}$.

Notre système étant confluent, il existe au plus une forme normale. La terminaison elle implique qu'il en existe au moins une.

On a donc montré que notre système converge vers une forme normale unique.

2.1.3.3 Correction

Le système de réécriture étant décrit et prouvé fortement normalisant, il reste à montrer que les valeurs abstraites qu'il retourne sont bien correctes.

[6] On pourrait d'ailleurs utiliser l'algorithme de complétion de Knuth-Bendix pour vérifier la confluence de notre système

Démonstration. Soit $\mathcal{A}$ une forme normale du système de réécriture tel qu'une des règles de correction n'est pas vérifiée. Alors on peut appliquer les réductions idoines données à la table 2.7, ce qui contredit le fait que $\mathcal{A}$ est une forme normale. □

La table 2.7 est construite à partir de l'ensemble des remarques formulées tout au long de la section « Représentation canonique » : la page indiquée fait référence à l'endroit où la règle a été discutée.

règle	*réductions à appliquer*	*page*
[Meet]	$\rightarrow_{neq\_eq} \rightarrow_{bot\_neq}$	27
[Eq]	$(\rightarrow_{int\_leq} \rightarrow_{int\_geq})$ ou $\rightarrow_{neq\_eq}$ ou $\rightarrow_{leq\_trans}$	27
[Neq]	$\rightarrow_{bot\_neq}$	-
[Rfl]	$\rightarrow_{rfl}$	-
[AntiRfl]	$\rightarrow_{arfl}$	-
[TransEq]	$\rightarrow_{leq\_trans}$	-
[TransNeq1,2]	$\rightarrow_{postneq} \rightarrow_{preneq}$	-
[C2I]	$\rightarrow_{int\_leq}$ ou $\rightarrow_{int\_geq}$ ou $\rightarrow_{int\_neq\_leq}$ ou $\rightarrow_{int\_neq\_leq}$	-
[C2Ineq]	$\rightarrow_{leq\_singl} (\rightarrow_{int\_neq\_leq}$ ou $\rightarrow_{int\_neq\_geq})$	28
[I2Ceq]	$\rightarrow_{leq\_singl}$	28
[I2Cempty]	$\rightarrow_{leq\_empty} (\rightarrow_{int\_neq\_leq}$ ou $\rightarrow_{int\_neq\_geq})$	-
[I2Csingl]	$\rightarrow_{leq\_singl} (\rightarrow_{int\_leq}$ ou $\rightarrow_{int\_geq})$	-

TAB. 2.7 – Réductions et règles de correction

DÉFINITION 9 (CANONISATION). *On définit la fonction* can *de canonisation*

$$\mathsf{can} : L^{\#} \rightarrow L^{\#}$$

où $\mathsf{can}(\mathcal{A})$ *est la représentation canonique de la valeur abstaite* $\mathcal{A}$ *calulée par réduction sur le système de réécriture défini précédemment.*

Complexité Malheureusement il est quasiment impossible de donner la complexité d'un tel système de réécriture conditionnel. La difficulté est double : il est difficile de cerner les pire cas, et lorsque l'on considère une valeur abstraite il est difficile de savoir qu'elle sera l'ordre de réduction le plus complexe.

On peut cependant être certain que la complexité est au moins en $O(n^3)$, qui est le coût de l'algorithme de fermeture transitive d'un graphe, ce qu'effectue notre réduction $\rightarrow_{leq\_trans}$.

2.1.4 Opérateurs

Notre domaine abstrait étant défini, il reste à le munir des opérateurs attendus pour son utilisation dans le cadre d'une analyse par interprétation abstraite.

Nous allons dans un premier temps définir la borne inférieure et la borne supérieure afin d'obtenir une structure de treillis, puis nous définirons un opérateur d'élargissement et enfin des opérateurs plus spécifiques destinés à l'analyse.

2.1.4.1 Opérateurs classiques aux treillis

Borne inférieure Nous avons déjà aperçu le principe de l'opérateur de borne inférieure lorsque nous avons défini la *fusion* (p. 27) de deux variables au sein d'une même valeur abstraite.

Intuitivement la borne inférieure de deux valeurs abstraites est une valeur abstraite qui regroupe *toutes* les informations connues sur chaque variable dans les deux valeurs abstraites. Elle consiste donc à intersecter les intervalles des variables et à unifier leurs ensembles de relations.

Définition 10 (borne inférieure). *Soit $\mathcal{A}, \mathcal{B} \in (L^\# \setminus \bot^\#)$ deux valeurs abstraites. Soit $\mathcal{C}$ définie par*

$$\mathcal{C} = (\lambda i.int_{\mathcal{A}}(i) \sqcap^I int_{\mathcal{B}}(i), \lambda i.neq_{\mathcal{A}}(i) \cup neq_{\mathcal{B}}(i), \lambda i.leq_{\mathcal{A}}(i) \cup leq_{\mathcal{B}}(i))$$

On définit la borne inférieure de $\mathcal{A}$ et $\mathcal{B}$, notée $\mathcal{A} \sqcap^\# \mathcal{B}$ par

$$\mathcal{A} \sqcap^\# \mathcal{B} = \mathsf{can}(\mathcal{C})$$

$$\bot^\# \sqcap^\# \mathcal{B} = \bot^\#$$

On remarque, comme cela l'a déjà été fait lorsque l'on a décrit la fusion, que l'on effectue une union uniquement sur les deux relations de $\mathsf{S}^\# = \{\neq, \leq\}$ et non sur les six relations de $\mathsf{S} = \{<, \leq, =, \neq, \geq, >\}$ pour obtenir l'union des contraintes des deux valeurs abstraites.

Nous avions émis que ceci était correct. Non seulement c'est le cas comme nous allons le montrer, mais c'est en fait l'union sur S qui serait incorrecte : en effet si l'on a la contrainte $j \neq i$ dans $\mathcal{A}$ et la contrainte $j \leq i$ dans $\mathcal{B}$, une union des contraintes sur S ne rendrait pas compte que $j < i$. Or ceci est *transparent* sur la représentation que nous avons choisit des valeurs abstraites.

Démonstration. Montrons que l'union sur $\mathsf{S}^\#$ est correcte.

Il suffit de voir qu'une fois l'union faite sur S il suffirait en fait de vérifier les règles de correction [Order] et [Meet1] (données page 23) pour obtenir une union correcte des contraintes.

Or ces deux règles de correction sont caduques et donc vérifiées pour la représentation des contraintes sur $\mathsf{S}^\#$. □

On notera enfin qu'effectuer une intersection des contraintes sur $\mathsf{S}^\#$ ou S est par contre strictement équivalent.

Borne supérieure Ce que l'on attend de la borne supérieure de deux valeurs abstraites c'est une valeur abstraite qui ne garde que les contraintes communes aux deux valeurs abstraites. Ainsi l'opération de borne supérieure consiste à intersecter les ensembles de relations de chaque variable et à effectuer l'union de leurs intervalles.

Définition 11 (borne supérieure). *Soit $\mathcal{A}, \mathcal{B} \in (L^\# \setminus \bot^\#)$ deux valeurs abstraites. Soit $\mathcal{C}$ définie par*

$$\mathcal{C} = (\lambda i.int_{\mathcal{A}}(i) \sqcup^I int_{\mathcal{B}}(i), \lambda i.neq_{\mathcal{A}}(i) \cap neq_{\mathcal{B}}(i), \lambda i.leq_{\mathcal{A}}(i) \cap leq_{\mathcal{B}}(i))$$

On définit la borne supérieure de $\mathcal{A}$ et $\mathcal{B}$, notée $\mathcal{A} \sqcup^\# \mathcal{B}$ par

$$\mathcal{A} \sqcup^\# \mathcal{B} = \mathsf{can}(\mathcal{C})$$

$$\bot^\# \sqcup^\# \mathcal{B} = \mathcal{B}$$

On remarquera que si $\mathcal{A}$ et $\mathcal{B}$ sont sous leur forme canonique, il n'est pas nécessaire de canoniser $\mathcal{C}$ pour obtenir leur borne supérieure. En effet si une réduction dans $\mathcal{C}$ est activée impliquant qu'un intervalle soit réduit ou une relation ajoutée, les conditions d'activation de cette réduction sont nécessairement vraies dans $\mathcal{A}$ ou dans $\mathcal{B}$, or ceux-ci sont supposés canoniques ce qui est contradictoire.

Treillis complet Les opérateurs de borne supérieure et inférieure étant défini nous introduisons ici la relation d'ordre sur $L^\#$ pour pouvoir définir le treillis complet $L^\#$ (le treillis est bien complet, le treillis des intervalles étant complet et les ensembles que l'on considère étant finis).

Définition 12 (relation d'ordre). *Soit $\mathcal{A}, \mathcal{B} \in L^\#$ sous forme canonique. On pose*

$$\mathcal{A} \sqsubseteq^\# \mathcal{B} \Leftrightarrow \mathcal{A} \sqcup^\# \mathcal{B} = \mathcal{B}$$

L'opération de borne supérieure de $\mathcal{A}$ et $\mathcal{B}$ n'impliquant pas de canonisation il vient,

$$\mathcal{A} \sqsubseteq^\# \mathcal{B} \Leftrightarrow \forall i \in Id \begin{cases} int_{\mathcal{A}}(i) \sqsubseteq^I int_{\mathcal{B}}(i) \\ neq_{\mathcal{B}}(i) \subseteq neq_{\mathcal{A}}(i) \\ leq_{\mathcal{B}}(i) \subseteq leq_{\mathcal{A}}(i) \end{cases}$$

On notera $\mathcal{A} \|^\# \mathcal{B}$, lorsque $\mathcal{A}$ et $\mathcal{B}$ sont incomparables, i.e. $\neg(\mathcal{A} \sqsubseteq^\# \mathcal{B} \vee \mathcal{B} \sqsubseteq^\# \mathcal{A})$.

$L^\#$ étant un treillis complet on peut donc le définir comme un domaine abstrait, le domaine abstrait IS .

Définition 13 (domaine abstrait). *On définit le treillis complet $L^\#$ dont les éléments sont des valeurs abstraites et où les opérateurs de borne inférieure, supérieure sont les opérateurs $\sqcup^\#$ et $\sqcap^\#$ et où la relation d'ordre est l'ordre $\sqsubseteq^\#$*

$$L^\# = ((Id \to \mathcal{V}^2, Id \to \mathcal{P}(Id), Id \to \mathcal{P}(Id)), \top^\#, \bot^\#, \sqsubseteq^\#, \sqcup^\#, \sqcap^\#)$$

Le domaine abstrait IS est le treillis $L^\#$.

Élargissement On définit enfin un opérateur d'élargissement. Il consiste à appliquer l'élargissement ∇^I du treillis des intervalles (*c.f.* section 1.2.1) sur la partie intervalles des deux valeurs abstraite, alors que les ensembles de relations ne sont pas élargis. Ceux-ci sont donc intersectés comme dans le cas de l'opération de borne supérieure[7]

Nous avions à la section 1.2.3 soulevé l'idée d'un élargissement pour le treillis des zones sur les différences de variables en les élargissant comme on le fait avec les intervalles. Par exemple $(i - j \leq -1)\nabla^{zones}(i - j \leq 0)$ amènerait $i - j \leq +\infty$. Sur notre domaine cela consisterait à oublier toute relation entre i et j si on devait élargir $i < j$ par $i \leq j$. Or comme nous allons le voir un tel opérateur d'élargissement serait préjudiciable à la précision des calculs.

Définition 14 (élargissement). *Soit $\mathcal{A}, \mathcal{B} \in (L^\# \setminus \bot^\#)$ deux valeurs abstraites non vides tq $\mathcal{A} \sqsubseteq^\# \mathcal{B}$. Soit $\mathcal{C}$ définie par*

$$\mathcal{C} = (\lambda i.int_{\mathcal{A}}(i) \nabla^I int_{\mathcal{B}}(i), \lambda i.neq_{\mathcal{B}}(i), \lambda i.leq_{\mathcal{B}}(i))$$

On définit l'élargissement de $\mathcal{A}$ par $\mathcal{B}$, noté $\mathcal{A}\nabla^\#\mathcal{B}$ par

$$\mathcal{A}\nabla^\#\mathcal{B} = \mathsf{can}(\mathcal{C})$$

$$\bot^\#\nabla^\#\mathcal{B} = \mathcal{B}$$

Nous donnons tout de suite un exemple d'application de cet opérateur d'élargissement.

[7] Cependant lorsque l'on effectue $\mathcal{A}\nabla^\#\mathcal{B}$ on a par hypothèse $\mathcal{A} \sqsubseteq^\# \mathcal{B}$ ce qui implique $\forall i, neq_{\mathcal{B}}(i) \subseteq neq_{\mathcal{A}}(i) \wedge leq_{\mathcal{B}}(i) \subseteq leq_{\mathcal{A}}(i)$. L'intersection des relations revient donc à prendre les relations de $\mathcal{B}$

Exemple 2. *Soit*

$$\mathcal{A} = \left(\begin{array}{llll} i \to (& [0,20], & \{j\}, & \{i\} \quad) \\ j \to (& [10,50], & \{i\}, & \{i,j\} \quad) \end{array} \right) \quad \mathcal{B} = \left(\begin{array}{llll} i \to (& [0,30], & \emptyset, & \{i\} \quad) \\ j \to (& [10,50], & \emptyset, & \{i,j\} \quad) \end{array} \right)$$

Avec les notations précédentes on calcule d'abord

$$\mathcal{C} = \left(\begin{array}{llll} i \to (& [0,+\boldsymbol{\infty}], & \boldsymbol{\emptyset}, & \{i\} \quad) \\ j \to (& [10,50], & \boldsymbol{\emptyset}, & \{i,j\} \quad) \end{array} \right)$$

que l'on canonise

$$\mathcal{A}\nabla^{\#}\mathcal{B} = \left(\begin{array}{llll} i \to (& [0,\mathbf{50}], & \emptyset, & \{i\} \quad) \\ j \to (& [10,50], & \emptyset, & \{i,j\} \quad) \end{array} \right)$$

Sur cet exemple on remarque que d'avoir gardé la contrainte $i \leq j$ a permis d'effectuer un élargissement plus fin que si l'on avait oublié cette contrainte. On notera la similitude avec l'exemple d'analyse que nous avions effectué à la section 1.3.3 où la séquence descendante permettait de redécouvrir une borne à un intervalle après l'action de l'élargissement.

L'opérateur proposé paraît donc, bien que plus simple, comme le plus efficient.

Il nous faut prouver qu'il s'agit bien d'un élargissement et donc montrer qu'il remplit les deux propriétés de ces opérateurs (*c.f.* section 1.1.2).

Démonstration.

- *(prop. 1)* On se convainc aisément que $\mathcal{A} \sqcup^{\#} \mathcal{B} \sqsubseteq^{\#} \mathcal{A}\nabla^{\#}\mathcal{B}$. Comme l'a montré l'exemple la borne d'une intervalle peut-être ramenée par la canonisation, mais jamais au delà de l'union des deux intervalles, sinon c'est que $\mathcal{A}$ ou $\mathcal{B}$ n'était pas canonique.
- *(prop. 2)* On doit montrer que l'élargissement de valeurs abstraites croissantes converge en un nombre fini d'itérations. Cette question se ramène à savoir si la canonisation ne peut pas empêcher la convergence sur les intervalles.
  Soit $\mathcal{D}$ le résultat d'un élargissement où la contrainte $i \leq j$ a ramené la borne de l'intervalle de i. Si on élargit $\mathcal{D}$ par une valeur abstraite $\mathcal{E}$ plus grande, soit
  - l'intervalle de j est identique dans $\mathcal{E}$ et la contrainte $i \leq j$ existe toujours. Alors l'intervalle de i ne peut bouger, et donc converge.
  - la contrainte $i \leq j$ n'est plus. On converge trivialement sur i
  - l'intervalle de j a grandi ($\mathcal{E} \sqsupseteq^{\#} \mathcal{D}$) et donc il est élargi et la contrainte $i \leq j$ n'empêchera plus la convergence sur i

  Bien entendu, la borne de l'intervalle j pourrait être elle aussi ramenée par une autre variable l. Cependant nous considérons un nombre fini de variable et donc, la suite d'élargissement converge bien en un nombre fini d'itérations.

□

2.1.4.2 Opérateurs spécifiques

Lors de l'analyse on va fatalement traiter des affectations sur les variables. Toute information connue sur la variable qui va subir l'affectation doit être retirée de l'approximation abstraite courante.

On généralise l'opération de substitution – donnée page 26 – en permettant de changer les trois composantes relatives à une variable en une seule fois, *i.e.* si $\mathcal{B} = \mathcal{A}[u/i]$ alors $\mathcal{B}_{|i} = u$ et $\forall j \neq i, \mathcal{B}_{|j} = \mathcal{A}_{|j}$.

Définition 15 (oubli). *Soit $\mathcal{A}$ une valeur abstraite et i une variable. On définit la fonction* $\mathsf{oubli} : L^\# \times Id \to L^\#$ *par*

$$\mathcal{B} = (int_\mathcal{A}, j \to neq_\mathcal{A}(j) \setminus \{i\}, j \to leq_\mathcal{A}(j) \setminus \{i\})$$

$$\mathsf{oubli}(\mathcal{A}, i) = \mathcal{B}[\top^I_{|i}/i]$$

Cette formulation de l'opérateur est un peu formelle, mais elle a l'avantage de faire ressortir la valeur abstraite $\top^\#$ qui signifie bien la perte d'information induite par l'opérateur.

On souhaite également pouvoir ajouter des variables à une valeur abstraite, en spécifiant sa valeur (par exemple pour considérer une constante importante du programme analysé) ou non, ce qui nous mène à définir l'opérateur suivant.

Définition 16 (création). *Soit $\mathcal{A}$ une valeur abstraite, $j \notin Id$ une variable et a une constante dans $\mathcal{V}$. On définit* $\mathsf{creation} : L^\# \times Id \times \mathcal{V} \to L^\#$ *par*

$$\mathsf{creation}(\mathcal{A}, j, a) = \begin{cases} \mathcal{A}[(\top^I, \emptyset, \{j\})/j] & si\ a = +/-\infty \\ \mathsf{can}(\mathcal{A}[([a,a], \emptyset, \{j\})/j]) & sinon \end{cases}$$

L'existence de cet opérateur implique que les opérateurs de borne inférieure, supérieure et élargissement doivent être étendu au cas où les deux valeurs abstraites ne soient pas définies sur le même ensemble de variables.

Si $\mathcal{A}$ est défini sur l'ensemble de variables $Id_\mathcal{A}$ et $\mathcal{B}$ sur l'ensemble de variables $Id_\mathcal{B}$ lors des opérations $\sqcup^\#$, $\sqcap^\#$ et $\nabla^\#$ on suppose que

$$\forall i \in Id_\mathcal{A} \setminus Id_\mathcal{B} \quad \mathcal{B}_{|i} = \top^\#_{|i} \qquad \forall i \in Id_\mathcal{B} \setminus Id_\mathcal{A} \quad \mathcal{A}_{|i} = \top^\#_{|i}$$

2.2 Cadre de l'interprétation abstraite

Nous avions énoncé à la section 1.1 les principes de l'interprétation abstraite. À la base même de cette théorie, il y a la construction d'un domaine abstrait, ce qui a fait l'objet de la section précédente.

Notre objectif étant l'analyse de programme, notre domaine concret est l'ensemble des parties des valuations. Il reste donc à définir le couple des fonctions d'abstraction et de concrétisation qui doit constituer une connexion de Galois pour que les théorèmes de la théorie soient valides. C'est l'objet de la section 2.2.1.

Dans la section 2.2.2, il nous faudra définir le langage des programmes que l'on souhaite analyser et donner sa sémantique abstraite. Nos programmes seront modélisés par des automates interprétés (1.12) et donc définir cette sémantique abstraite nous amènera à définir les significations abstraites d'une garde et d'une commande du langage.

2.2.1 Fonctions α, γ

On rappelle que Id est un ensemble de variables prenant leurs valeurs dans $\mathcal{N}$. On note L le domaine concret qui est l'ensemble des parties des valuations :

$$L = 2^{Id \to \mathcal{N}}$$

Abstraction Soit $R \in L$, un ensemble de valuations. Si R est vide son abstraction est la valeur abstraite vide $\perp^\#$.

Sinon pour une variable donnée, R donne l'ensemble des valeurs qu'elle peut prendre : pour la partie intervalle, ceci est abstrait par le plus petit intervalle contenant toutes ces valeurs. Pour la partie relationnelle, on inclut une relation dans l'abstraction que si elle est vérifiée pour toute valuation de R.

DÉFINITION 17 (ABSTRACTION). *On définit $\alpha(R)$ l'abstraction de R dans $L^\#$ par*

$$\alpha(R) = \begin{cases} \perp^\# & si\ R = \emptyset \\ (int_R, neq_R, leq_R) & sinon \end{cases}$$

où int_R, neq_R et leq_R sont définis par

$$\begin{aligned} int_R &= \lambda i.[\inf\{\rho(i) \mid \rho \in R\},\ \sup\{\rho(i) \mid \rho \in R\}] \\ neq_R &= \lambda i.\{j \mid (\forall \rho \in R\ \rho(j) \neq \rho(i))\} \\ leq_R &= \lambda i.\{j \mid (\forall \rho \in R\ \rho(j) \leq \rho(i))\} \end{aligned}$$

Concrétisation La concrétisation d'une valeur abstraite $\mathcal{A} = (int_\mathcal{A}, neq_\mathcal{A}, leq_\mathcal{A})$ est intuitive.

On peut la voir comme la construction de l'ensemble des valuations construites par combinaison exhaustive de chaque valeur possible pour chaque variable prise dans sont intervalle auquel on enlève toutes les valuations où les relations exprimées par $neq_\mathcal{A}$ et $leq_\mathcal{A}$ ne sont pas respectées.

DÉFINITION 18 (CONCRÉTISATION). *On définit $\gamma(\mathcal{A})$ la concrétisation de $\mathcal{A}$ dans L par*

$$\gamma(\mathcal{A}) = \begin{cases} \emptyset & si\ \mathcal{A} = \perp^\# \\ \left\{ \rho : Id \to \mathcal{N} \;\middle|\; \begin{array}{l} \forall i\ \rho(i) \in \gamma_I(i) \\ \wedge\ (\forall j \in neq_\mathcal{A}(i)\ \rho(j)\ \neq\ \rho(i)) \\ \wedge\ (\forall j \in leq_\mathcal{A}(i)\ \rho(j)\ \leq\ \rho(i)) \end{array} \right\} & sinon \end{cases}$$

où $\gamma_I(i) = \{x \in \mathcal{N} \mid a \leq x \leq b \quad [a,b] = int_\mathcal{A}(i)\}$

Connexion de Galois On a trivialement l'équivalence suivante :

$$\forall R \in L, \forall \mathcal{A} \in L^\# \qquad \alpha(R) \sqsubseteq^\# \mathcal{A} \Leftrightarrow R \subseteq \gamma(\mathcal{A})$$

qui est la définition d'une connexion de Galois entre le treillis L et le treillis abstrait $L^\#$.

Cette section clôt la définition de notre domaine abstrait puisque l'on a montré qu'il remplissait toutes les conditions nécessaires pour son utilisation dans le cadre de l'interprétation abstraite.

2.2.2 Application aux automates interprétés

L'objectif de cette section est de montrer comment s'effectue l'analyse d'automates interprétés sur notre domaine abstrait.

Pour ce faire nous allons d'abord définir les gardes et les actions que nous autorisons sur nos automates interprétés, puis donner leurs sémantiques abstraites.

Alors le travail sera terminé. L'analyse pourra s'effectuer dans le domaine abstrait, selon le cadre théorique de l'interprétation abstraite décrit à la section 1.3.

2.2.2.1 Syntaxe des gardes et des actions

Nous définissons des expressions basiques pour les gardes et les actions mais appropriées pour pouvoir modéliser des programmes où des variables numériques entières interviennent et où l'on souhaite vérifier l'égalité de deux variables

On choisit donc pour espace des valeurs $\mathcal{N} = \mathbb{Z}$. On ne considère pas la multiplication pour les expressions qui apporterait peu d'information pour nos valeurs abstraites. On définit (les variables sont notées par les lettres i, j et les constantes numériques par les lettres n, m) les expressions arithmétiques (*Expr*),

$$e \ ::= \ i \mid n \mid i \ + \ n$$

et les expressions booléennes (*ExprBool*)

$$b \ ::= \ \mathsf{true} \mid \mathsf{false} \mid \neg b \mid b \ \vee \ b \mid b \ \wedge \ b \mid e \ \mathsf{S} \ e$$

Nos automates interprétés auront pour gardes une expression booléenne dans *ExprBool* et pour unique commande l'affectation d'une expression dans *Expr* à une variable.

$$\text{Com} = \{i := e \text{ avec } e \in \mathit{Expr}, i \in \mathit{Id}\} \qquad \text{Gardes} \subset \mathit{ExprBool}$$

2.2.2.2 Sémantique abstraite

Abstraction des gardes Formellement on peut définir l'abstraction d'une garde $g \in \text{Gardes}$, notée $\mathsf{abs}(g)$ par

$$\mathsf{abs}(g) = \alpha(\overline{g}) \quad \text{où } \overline{g} = \{\rho \in \mathit{Id} \to \mathcal{N} \mid g(\rho) = \mathit{vrai}\}$$

Cependant on souhaite donner une méthode constructive pour trouver l'abstraction d'une expression booléenne b dans notre domaine.

Soit $b \in \mathit{ExprBool}$ une expression booléenne quelconque. Bien entendu on a $\mathsf{abs}(\mathsf{true}) = \top^\#$ et $\mathsf{abs}(\mathsf{false}) = \bot^\#$. On peut décomposer le problème sur les conjonctions et les disjonctions d'expressions booléennes :

$$\mathsf{abs}(b = b_1 \wedge b_2) = \mathsf{abs}(b_1) \sqcap^\# \mathsf{abs}(b_2)$$

$$\mathsf{abs}(b = b_1 \vee b_2) = \mathsf{abs}(b_1) \sqcup^\# \mathsf{abs}(b_2)$$

On est donc ramené à considérer seulement les expressions de la forme $e \ \mathsf{S} \ e$ (les négations $\neg(e \ \mathsf{S} \ e)$ s'y ramenant également).

À la table 2.8 on donne le résultat de la fonction abs pour cinq expressions booléennes permettant de déterminer le résultat de abs sur toutes les contraintes de la forme $i \ \mathsf{S} \ n$ et $i \ \mathsf{S} \ j$ (par conjonction selon le treillis L^S : par exemple $\mathsf{abs}(i < j) = \mathsf{abs}(i \neq j) \sqcap^\# \mathsf{abs}(i \leq j)$).

On remarquera que pour $i \neq n$ on est contraint à créer une nouvelle variable $\_n$ alors que pour les expressions $i \leq n$ et $i \geq n$ l'information peut-être portée par les intervalles seuls.

b	$\mathsf{abs}(b)$	*graphiquement*
$i \neq j$	$\left(\begin{array}{l} i \to (\ \top^I,\ \{j\},\ \{i\}\ ) \\ j \to (\ \top^I,\ \{i\},\ \{j\}\ ) \end{array}\right)$	i —— j
$i \leq j$	$\left(\begin{array}{l} i \to (\ \top^I,\ \emptyset,\ \{i\}\ ) \\ j \to (\ \top^I,\ \emptyset,\ \{j,i\}\ ) \end{array}\right)$	i ◇— j
$i \neq n$	$\left(\begin{array}{l} i \to (\ \top^I,\ \{\_n\},\ \{i\}\ ) \\ \_n \to (\ [n,n],\ \{i\},\ \{\_n\}\ ) \end{array}\right)$	i —— $\_n^{[n,n]}$
$i \leq n$	$(\ i \to (\ [-\infty, n],\ \emptyset,\ \{i\}\ )\ )$	$i^{[-\infty, n]}$
$i \geq n$	$(\ i \to (\ [n, +\infty]\ \emptyset,\ \{i\}\ )\ )$	$i^{[n, +\infty]}$

TAB. 2.8 – Valeur abstraite d'une garde *(1)*

Pour les autres expressions booléennes sur e S e, la table 2.9 donne le résultat de abs pour celles où l'information peut être exprimée dans notre domaine. Pour les autres on a $\mathsf{abs}(b) = \top^\#$.

Ce qui est exploité ici, c'est le fait que dans le cas arithmétique $\forall l > j, l \geq succ(j)$ (*i.e.* dans $\mathbb{Z}$ il n'y a pas de valeur entre j et $j+1$).

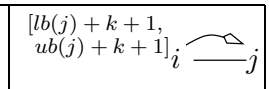
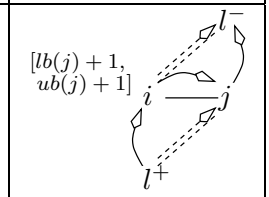
$b \quad (k \in \mathbb{N}^*)$	$\mathsf{abs}(b)$	*graphiquement*
$i < j + 1$	$\mathsf{abs}(i \leq j)$	i ◇— j
$i \geq j + 1$ $i > j + k$ $i = j + k$	$\mathsf{abs}(i > j)$	i ⌒ j

TAB. 2.9 – Valeur abstraite d'une garde *(2)*

Abstraction contextuelle On note que pour $i = j + 1$ son asbtraction ne permet pas, comme le laisserait penser la remarque précédente que si il *existe* des variables l telles que $l > j$, on ait la relation $l \geq i$ dans l'abstraction.

Il suffirait pour cela d'utiliser une valeur abstraite, nommée contexte, pour gagner en précision (si une telle valeur existe).

La table 2.10 donne la valeur abstraite de gardes pour lesquelles le contexte influe. (On étend $\mathsf{abs}(b)$ à $\overline{\mathsf{abs}}(b, \mathcal{C})$).

$b \quad (k \in \mathbb{N}^*)$	$\overline{\mathsf{abs}}(b, \mathcal{C})$	*graphiquement*
$i > j + k$ $i = j + 1 + k$	$\begin{pmatrix} i \to (& a, & \{j\}, & \{i,j\} &) \\ j \to (& \top^I, & \{i\}, & \{j\} &) \end{pmatrix}$ $a = [\mathsf{lb}(int_{\mathcal{C}}(j)) + k + 1, \mathsf{ub}(int_{\mathcal{C}}(j)) + k + 1]$	
$i = j + 1$	$\begin{pmatrix} i \to (& a, & \{j, l^-\}, & \{i, j, l^-\} &) \\ j \to (& \top^I, & \{i, l^+\}, & \{j, l^-\} &) \\ \forall l^+ \to (& \top^I, & \{j\}, & \{l^+, i, j\} &) \\ \forall l^- \to (& \top^I, & \{i\}, & \{l^-\} &) \end{pmatrix}$ $a = [\mathsf{lb}(int_{\mathcal{C}}(j)) + 1, \mathsf{ub}(int_{\mathcal{C}}(j)) + 1]$ l^- variable *t.q.* $l^- \in leq_{\mathcal{C}}(i)$ *i.e.* $l^- < i$ dans $\mathcal{C}$ l^+ variable *t.q.* $j \in (neq_{\mathcal{C}}(l^+) \cap leq_{\mathcal{C}}(l^+))$ *i.e.* $j < l^+$ dans $\mathcal{C}$	

TAB. 2.10 – Valeur abstraite *contextuelle* d'une garde

Abstraction des commandes On rappelle l'existence des opérateurs creation et oubli, défini à la section 2.1.4.2, qui permettent respectivement d'ajouter une variable à une valeur abstraite et d'oublier les informations connues sur une variable.

Pour chaque expression dans *Expr* on souhaite construire, à partir de la valeur abstraite $\mathcal{C}$ sur laquelle agit la commande, une valeur abstraite où l'expression apparaît et récupérer ainsi les relations qu'elle a dans $\mathcal{C}$.

La table 2.11 donne la définition de la fonction $\mathsf{E} : \mathit{Expr} \times L^\# \longrightarrow L^\#$ qui à une expression e et une valeur abstraite $\mathcal{C}$ renvoie une valeur abstraite contenant $\mathcal{C}$ et e.

e	$\mathsf{E}(e, \mathcal{C})$
n	$\mathsf{creation}(\mathcal{C}, \_n, n)$
$i \in Id$	$\mathcal{C}$
$i \notin Id$	$\mathsf{creation}(\mathcal{C}, i, \infty)$
$i + n$	$\overline{\mathsf{abs}}(\_i\_n = i + n, \mathsf{creation}(\mathcal{C}, \_i\_n, \infty))$

TAB. 2.11 – Valeur abstraite d'une expression

Les trois premiers cas sont intuitifs. Pour les expressions de la forme $i + n$ l'idée est, dans un premier temps, de créer une variable $\_i\_n$ dans $\mathcal{C}$ et dans un second temps de calculer la contribution abstraite de la garde $\_i\_n = i + n$ qui va nous donner une valeur abstraite où $\_i\_n$ a les relations attendues de l'expression $i + n$.

L'abstraction de l'affectation $j := e$ dans $\mathcal{C}$ consiste alors à oublier la variable j dans $\mathcal{C}$ et à effectuer une opération de borne inférieure avec la valeur abstraite $\mathsf{E}(e, \mathcal{C})$, où j est par définition remplacée par l'expression e. La borne inférieure crée alors la valeur abstraite attendue pour l'abstraction de l'affectation.

On définit donc

$$\mathsf{abs\_cmd}(i := e)(\mathcal{A}) = \mathsf{oubli}(\mathcal{A}, i) \sqcap^\# \mathsf{E}(e, \mathcal{A})$$

Chapitre 3

Implémentation et résultats

Outre la définition d'un nouveau domaine numérique abstrait pour la vérification de propriétés d'égalité/non-égalité sur les adresses, le second objectif de ce stage aura été l'implémentation d'un prototype d'outil d'analyse de programmes par interprétation abstraite sur ce domaine.

Plutôt que de se lancer aveuglément dans la conception d'un outil ad hoc, nous avons considéré des outils existants, basés sur la théorie de l'interprétation abstraite, dont l'outil PAG[1] (Program Analyzer Generator) développé par l'université de Saarland et l'outil NBAC[2] (Numerical and Boolean Automaton Checker) développé par Bertrand Jeannet durant sa thèse au laboratoire Vérimag.

L'outil PAG permet de spécifier en paramètre un domaine abstrait numérique sur lequel il peut effectuer le calcul de point-fixe. Cette généricité était à l'ordre du jour des développements futurs de NBAC. Cependant nous avons préféré travailler à l'extension des outils d'analyse du laboratoire et donc travailler de concert avec Bertrand Jeannet.

3.1 Implémentation

3.1.1 Environnement d'analyse

NBAC, écrit en OCAML, est un outil de vérification de propriétés de sûreté du langage LUSTRE, langage synchrone flots de données développé à Vérimag. Il permet de façon plus générale par compilation vers ce modèle, l'analyse de programmes asynchrones, de systèmes réactifs déterministes, contenant des booléens et des variables numériques.

Le treillis abstrait utilisé est le produit du treillis des booléens et du treillis des polyèdres convexes (*c.f.* section 1.2.2) : un ensemble d'états est alors représenté par la conjonction d'un BDD et d'un polyèdre convexe.

Outre la possibilité d'effectuer des analyses d'accessibilité et de co-accessibilité, permettant la vérification de propriétés ou encore la synthèse d'invariants, l'originalité de NBAC vient des techniques de partitionnement dynamique qu'il met en oeuvre (*c.f.* section 1.3.4).

Ces techniques étant fortement liées au domaine abstrait utilisé pour l'analyse, il s'est avéré que seule la partie implémentant le calcul itératif de la résolution d'une équation de point fixe sur un treillis pouvait être rendue générique.

Bertrand Jeannet a alors « débranché » cette partie de NBAC, que l'on a appellé *analyseur*, dont la structure – simplifiée – est donnée à la figure 3.1.

[1] `http://www.absint.com/pag/`

[2] `http://www.irisa.fr/prive/bjeannet/nbac/nbac.html`

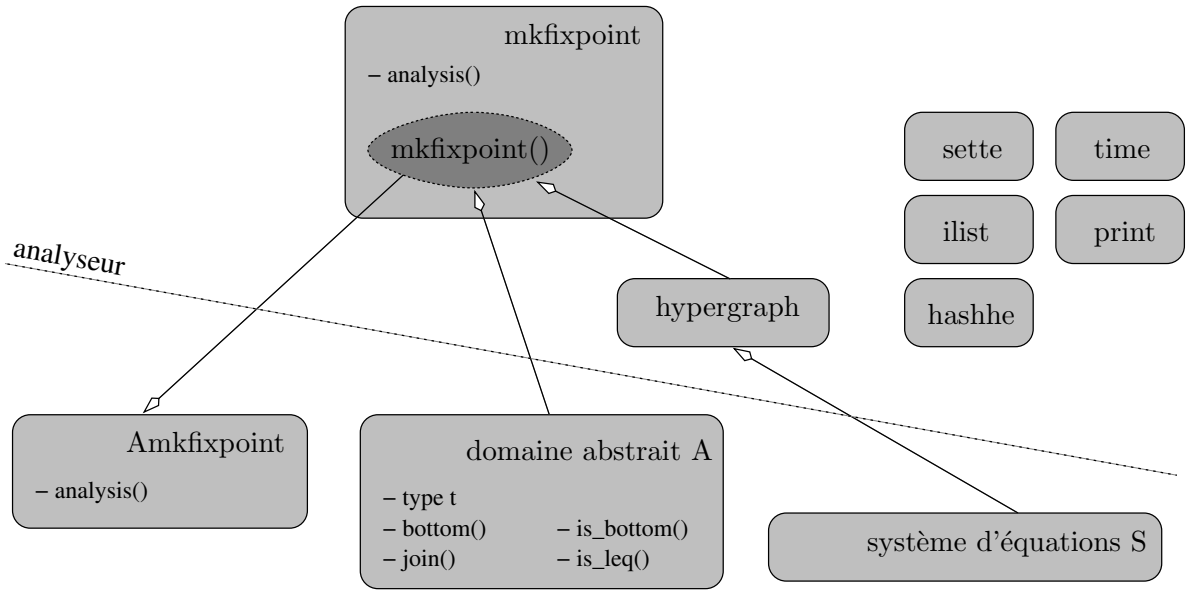


FIG. 3.1 – Analyseur : architecture générale

En haut de la figure on trouve l'ensemble des modules composant l'analyseur. Le module principal `mkFixpoint` (pour « make fixpoint ») fournit un foncteur `mkFixpoint()` qui à un domaine numérique abstrait `A` et un système d'équations `S` renvoie un module `AmkFixpoint` contenant entre autres la fonction `analysis()` qui permet d'effectuer le calcul d'un post-point fixe de `S` dans `A`.

Ces deux arguments du foncteur sont donnés sous forme de modules, que l'on trouve au bas de notre figure. Ils sont donc les modules à fournir, côté utilisateur de l'analyseur, pour pouvoir effectuer une analyse d'accessibilité :

- le module principal est celui qui décrit le *domaine abstrait*. On remarquera qu'il requiert peu de composants. Le `type t` est celui des éléments du domaine abstrait dont on fournit le plus petit, et seuls l'opérateur de borne supérieure et la relation d'ordre du treillis abstrait sont à définir.
- le second module permet d'écrire le *système d'équations* par une structure générique de graphe définie dans le module `hypergraph`. Il s'agit de graphes où les sommets sont les inconnues et où les hyperarcs (arcs pouvant être définis sur plus de deux sommets) portent des fonctions qui à un vecteur d'inconnues délivrent un vecteur de résultats.
  Dans notre cas précis le graphe sera un automate interprété modélisant le programme sous vérification. Dans NBAC par contre le système d'équations n'est initialement pas partitionné : c'est un graphe à un état.

Les autres modules de moindre importance, `ilist`, `sette` et `hashhe` sont des structures permettant, respectivement, la manipulation de listes imbriquées, d'ensembles et de tables de hachage. On notera par exemple que les hypergraphes sont stockés dans des tables de hachages, ceux-ci pouvant être de taille conséquente (dizaines de milliers d'états).

Une fois le module `AmkFixpoint` construit, un certain nombre de paramètres doit être donné à la fonction `analysis()` :

- les valeurs initiales du système d'équations.
- l'opérateur d'élargissement. En le spécifiant ici, on peut comparer des analyses obtenues

par différents opérateurs d'élargissement plus ou moins précis, sans avoir à modifier le coeur du domaine abstrait.
- la stratégie d'itération, écrite dans une `ilist`. Par exemple la stratégie `[[1; [2;3]; 4; [[6;7]; 8]; 6]]` impose de mettre à jour le sommet 1, puis les sommets 2 et 3 jusqu'à stabilisation et ainsi de suite. Les boucles de stabilisation peuvent même être récursives : les valeurs en 6 et 7 doivent être « temporairement » stables avant de continuer en 8. Tout ceci rejoint les considérations faites à la section 1.3.2. sur le calcul des points-fixes d'un système d'équation.
- les points d'élargissement.
- le nombre d'itérations en séquence descendante.

On notera qu'une fonction nommée `strategy_defaut` est fournie permettant de générer automatiquement une stratégie à partir du système d'équation et de trouver les points d'élargissement (par l'heuristique des sous CFCs : *c.f.* section 1.3.2).

Nous organisons notre propos autour des deux modules utilisateurs à fournir. La première section décrit l'implémentation que nous avons faites du domaine abstrait IS et dans une seconde section nous discuterons de la construction de l'équation de point-fixe des programmes qu'on veut analyser.

3.1.2 Implémentation du domaine abstrait

Nous avons implémenté le domaine IS directement selon sa définition mathématique. Ce choix s'avérait être alors le plus judicieux lorsque l'on a débuté l'implémentation. D'autres représentations étaient envisagées, notamment la représentation des relations par des matrices binaires plus coûteuses en espace : ces différentes pistes on aboutit à une idée plus large présentée au chapitre « perspectives » qu'est l'extension du domaine aux DBMs.

Type des valeurs abstraites Les valeurs abstraites sont donc représentées par un 3-uplet de fonctions, ce qui est aisé à écrire en OCaml.

```
type t = { mutable int: var -> Interval.t;
   mutable neq: var -> var Sette.t;
   mutable leq: var -> var Sette.t; }
```

Il est important de signaler l'implémentation qui est faite des ensembles dans le module `Sette`. La structure de données utilisée est l'arbre binaire ordonné équilibré, l'AVL [CLRT90] dont le test de l'existence et l'insertion d'une variable dans un ensemble sont en $O(ln(n))$, ce qui est raisonnable pour notre application.

Opérateurs Nos opérateurs de treillis faisant appel à tous les opérateurs du treillis des intervalles nous avons d'abord implémenté le domaine abstrait des intervalles dans l'analyseur – et effectué quelques analyses sur celui-ci qui ont eu le mérite de faire apparaître plusieurs bugs non-triviaux dans le premier jet de l'analyseur.

Ceci étant, l'implémentation des opérateurs de borne inférieure, supérieure et d'élargissement ne posent pas non plus de difficultés particulières sur le treillis IS , si ce n'est l'appel à la fonction de canonisation, comme nous allons le voir.

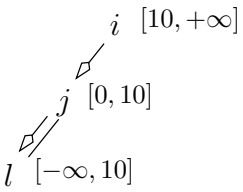
Canonisation Le problème posé était l'implémentation d'un système de réécriture conditionnel. L'algorithme trivial consiste à appliquer récursivement sur la valeur abstraite chaque règle de réécriture dont la pré-condition est vérifiée jusqu'à ce que plus aucune d'entre elles ne soit

activée. Ce processus termine et quel que soit l'ordre dans lequel on applique les réductions il fournit bien la forme canonique d'une valeur abstraite (*c.f.* sections 2.1.3.2 et 2.1.3.3).

De façon générale on peut partitionner l'ensemble des informations d'un terme à canoniser en celles qui n'activent pas de règles du système de réécriture et celles qui les activent. Le premier sous-ensemble sera dit « quasi-canonique » et le second « activant ». Ces ensembles sont ici un ensemble de variables (leur intervalle active une règle) et de relations entre variables.

Exemple 3. *Par exemple la valeur abstraite ci-dessous n'est pas canonique. Elle doit subir deux réductions, l'une modifiant l'intervalle de* l *(*$[-\infty, 9]$*), l'autre ajoutant la relation* $l \leq i$*.*

On a par exemple pour « activants » : $\{intervalle(i), j \leq i\}$ *ou encore* $\{l \leq j\}$ *qui permet à lui seul d'activer les deux réductions.*



Lorsque l'on canonise on peut donc se restreindre à regarder chaque élément de l'ensemble « activant », appliquer les règles de réécriture que cet élément active (ce qui peut augmenter l'« activant »), et recommencer sur le nouvel « activant » jusqu'à ce qu'il soit vide.

C'est exactement ce qu'effectue notre fonction récursive de canonisation `canonical_form` qui prend en paramètres la valeur abstraite à canoniser et la liste des relations et des variables appartenant à l'« activant » courant et qui a pour condition d'arrêt que cette liste soit vide.

```
type relation = Neq of (var * var) | Leq of (var * var) | Int of var

val canonical_form : t -> relation list -> t
```

Ainsi lorsque l'on veut effectuer la borne inférieure de deux valeurs abstraites canoniques `a` et `b`, on construit la valeur abstraite `c` qui consiste en l'intersection des intervalles et l'union des relations – par définition – et on choisit l'une des deux valeurs abstraites `a` ou `b` (supposons `a`) comme l'ensemble « quasi-canonique » de `c`. On fournit alors la liste `todo` des « activants » qui sont les relations existants dans `c` et pas dans `a` et les variables dont les intervalles sont plus petits dans `c` que dans `a`. Le résultat de la borne inférieure est alors donné par `canonical_form c todo`.

On voit bien que le choix entre `a` et `b` est la clé de la diminution des itérations de la fonction de canonisation. Par exemple dans `b` il n'y avait peut-être qu'une ou deux relations alors que `a` en comptait des centaines, ou encore `b` était peut-être définie sur un petit nombre de variable comparé à `a`. (ce qui sera souvent le cas si `b` est l'abstraction d'une garde). Le choix de `b` pour construire la liste `todo` aurait alors été plus judicieux.

Au final la décision est prise en calculant la taille des deux listes `todo` correspondant aux alternatives où `a` ou `b` est l'ensemble « quasi-canonique » de `c`.

3.1.3 Création de l'automate interprété

Le module `hypergraph` fournit toutes les primitives nécessaires pour ajouter/retirer des sommets et des hyperarcs à un graphe et donc permet aisément de construire la structure de l'automate interprété.

S'il est facile d'exprimer cette structure de contrôle, les fonctions sur les hyperarcs doivent être exprimées dans une fonction `apply` qui à un hyperarc (reconnu par une valeur unique de type `'a`) et une valeur abstraite applique la fonction et renvoie la valeur abstraite résultat.

```
val apply : (hyperhedge:'a) -> t -> t
```

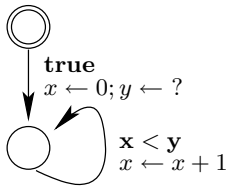
Dans le cadre d'un automate interprété on voit que la fonction `apply` peut-être construite automatiquement à partir des actions et des gardes dont la grammaire est connue.

Nous avons donc créé un module `semAbs` (pour « sémantique abstraite ») qui définit le type des actions et des gardes de la sémantique concrète et fournit deux fonctions, l'une renvoyant l'abstraction d'une garde `abs_guard` et l'autre renvoyant le résultat d'une commande sur une valeur abstraite (*c.f.* section 2.2.2).

```
type expr = Var of var | Num of int | Plus of var * int
type guard = True | False | .... | Or of expr * expr  | .... | Leq of expr * expr
type cmd = Assign of var * expr

val abs_guard : t -> guard -> t
val apply_cmd : t -> cmd -> t
```

Ainsi, à partir d'un fichier représentant l'automate interprété, celui-ci est parsé sur le langage que l'on vient de définir et la fonction `apply` est construite automatiquement comme ci-dessous pour l'automate exemple de la section 1.3.3 que l'on redonne :



On a repéré les hyperarcs par des entiers (`'a = int`), l'hyperarc `1` est l'arc allant du noeud initial au noeud 1, l'hyperarc `2` est la boucle sur le noeud 1 et l'hyperarc `3` l'arc du noeud 1 au noeud final.

On rappelle que la fonction est de la forme $a^{\#}(\mathcal{A} \sqcap^{\#} g^{\#})$ c'est pourquoi pour l'hyperarc `2` et `3` l'opération de borne inférieure `meet` du treillis abstrait apparaît.

```
let apply = fun hedge -> abs_value ->
  match hedge with
       | 1 -> apply_cmd abs_value (Assign "x" (Num 1))
       | 2 -> let abstracted_guard = abs_guard abs_value (Leq_neq "x" (Num 100))
                in apply_cmd (meet abstracted_guard abs_value) (Assign "x" (Plus "x" 1))
       | 3 -> let abstracted_guard = abs_guard abs_value (Leq (Num 100) "x")
                in (meet abstracted_guard abs_value)
```

Il reste à choisir un format d'entrée décrivant des automates interprétés et à implémenter leur compilation en hypergraphe.

Le format d'entrée de Nbac ne satisfait pas du tout notre attente car très éloigné du modèle des automates interprétés. L'idée est d'utiliser le format `oc` en entrée, cible de la compilation de plusieurs langages synchrones (Lustre, Esterel, Argos), qui décrit un automate d'état finis où les transitions portent des ensembles d'actions – transcriptible en automate interprété. Outre la vérification de langage synchrone, l'avantage de cette solution est qu'elle nous permet de vérifier également des systèmes sur puces (plus proches de nos

motivations initiales, où de nombreuses adresses mémoires sont manipulées) décrits en SystemC, grâce à un compilateur de ce langage vers `oc` développé au laboratoire par Matthieu Moy.

Notre compilateur du langage `oc` vers l'analyseur en est à un stade avancé de développement et devrait être terminé dans les prochaines semaines.

Nous avons décidé de nous concentrer sur l'analyse de nombreux programmes simples, écrits spécifiquement pour tester les limites de l'expressivité de notre domaine abstrait et donc valider notre approche. On expose à présent quelques uns de ces exemples et le résultat de leur analyse sur notre outil.

3.2 Résultats expérimentaux

Nous présentons ici les résultats de l'analyse, dans le domaine abstrait IS , de plusieurs programmes avec l'outil *analyseur*.

Pour chacun d'eux, leur automate interprété est fourni et un tableau donne les valeurs abstraites attachées à chaque point de contrôle à la fin de l'analyse[3].

La commande `ERR` dans le programme permet de spécifier les états d'erreurs représentés par des états grisés dans l'automate interprété. Notre but est d'obtenir en ces états la valeur abstraite vide ($\bot^{\#}$), preuve de son inaccessibilité.

La commande `OK` mène à l'arrêt du programme.

Au total cinq exemples sont présentés. Les deux premiers exemples travaillent sur des valeurs initiales quelconques et donc ne font pas intervenir l'abstraction sur les intervalles – ce qui inclut le fait que l'opérateur d'élargissement est sans effet.

Exemple 1
Il s'agit d'un programme extrêmement simple qui permet de vérifier que le domaine IS infère bien l'égalité de deux variables lorsqu'il combine l'information que $i \leq j$ et que $j \leq i$, ce qui est le cas au noeud 5 après le passage des tests aux noeuds 1 et 3.
L'intersection avec la garde $x \neq y$ sur la transition du point de contrôle 5 au point de contrôle d'erreur, mène à la valeur abstraite vide attendue.

	états accessibles
1	$\top$
2	$x < y$
3	$y \leq x$
4	$y < x$
5	$x = y$
ERR	$\bot$

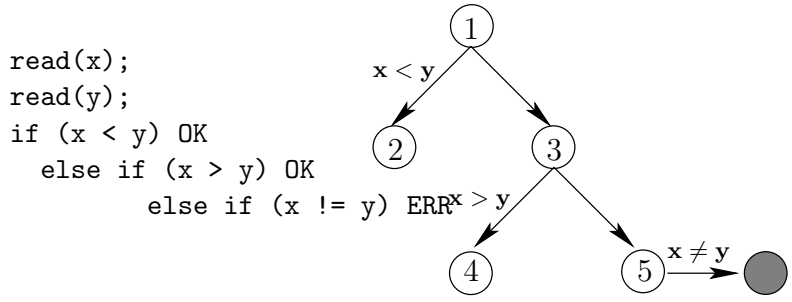


FIG. 3.2 – Exemple 1

[3]On notera que le temps d'exécution est inférieur à une seconde pour toutes ces analyses

Exemple 2
Dans cet exemple on souhaite montrer qu'à l'entrée de la boucle (noeud 3) on a pour invariant la non-égalité de x et de y.
Seul le domaine abstrait IS *peut obtenir ce résultat.* En effet, l'invariant au noeud 3 est l'union des informations connues en 6, en 8 et en 1. Si on note respectivement c_1 et c_2 les deux chemins d'exécution $(3, 4, 5, 6)$ et $(3, 4, 7, 8)$, on voit que sur c_2, le test au noeud 4 implique $x > y$, et que la non-égalité de x et de y est portée tout au long du chemin et donc en 8. Par contre sur c_1, le test au noeud 4 implique $x \leq y$ et si l'information que x est non-égale à y n'avait pas été mémorisée, au noeud 5 on en serait resté à $x \leq y$ (donc une possible égalité de x et de y) et l'on aurait pas obtenu $x < y$ qui permet de porter la non-égalité de x et de y au noeud 6.
Les trois noeuds $1, 6$ et 8 ayant alors pour invariant la non-égalité de x et de y, celui-ci est porté au noeud 3 et rend alors inaccessible le noeud d'erreur.

	états accessibles
1	$\top$
2	$x = y$
3	$x \neq y$
4	$x \neq y$
5	$x < y$
6	$y = z$, $x < y$, $x < z$
7	$x < y$
8	$z < x$, $x < y$, $z < y$
ERR	$\bot$

```
read(x);
read(y);
if (x = y) OK
  else while(1) {
         if (x = y) ERR
         read(z);
         if (x <= y)
           if (y <= z) y := z;
           else if (x <= z) x := z
       }
```

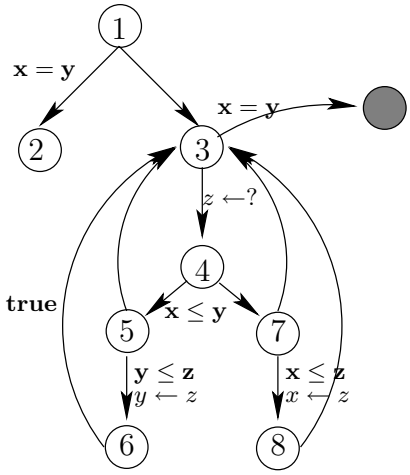


FIG. 3.3 – Exemple 2

Les exemples suivants affectent des valeurs à certaines variables et permettent de voir le rôle joué par les intervalles et l'efficacité de l'opérateur d'élargissement.

Exemple 3 On s'intéresse dans cet exemple essentiellement à vérifier des résultats simples sur l'interaction entre les intervalles et les différentes relations.

Les assignations au noeuds 1 et 2 permettent de vérifier l'abstraction des valeurs concrètes dans le plus petit intervalle les contenant. Avec la mise à zéro de y et l'application de la garde $t \leq y$, on vérifie que l'intervalle de t est bien restreint par y.

Enfin la dernière garde $x < y$ engendre un changement intéressant : en effet la borne inférieure de l'intervalle de x est incrémentée, ce qui a pour effet de rendre les intervalles de z et x comparables, ce qui est détecté : on trouve la relation $z \leq x$ au noeud 5.

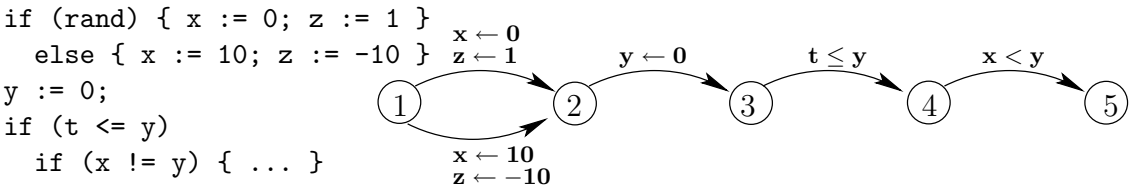


FIG. 3.4 – Exemple 3

	états accessibles
1	$\top$
2	$x \in [0, 10], z \in [-10, 1]$
3	$x \in [0, 10], z \in [-10, 1], y = 0, y \leq x$
4	$x \in [0, 10], z \in [-10, 1], y = 0, y \leq x, t \leq 0, t \leq x, t \leq y$
5	$x \in [1, 10], z \in [-10, 1], y = 0, y < x, t \leq 0, t < x, z \leq x$

Exemple 4
On peut voir le programme suivant comme effectuant pour les *adresses* i, j des parcours sur des intervalles. On veut s'assurer que durant ces parcours les adresses i et j sont différentes : par exemple si l'adresse i est lue et l'adresse j écrite, on garantira qu'un accès en lecture/écriture exclusif est respecté.
Les invariants aux points de contrôle $1, 2, 3, 4$ sont ceux attendus. L'invariant qui nous permet de prouver notre propriété est $i < j$ au point de contrôle 6. Voyons comment il est obtenu :
– la garde $x < y$ sur la transition $1 \to 3$ et la garde $i \leq x$ sur $4 - 5$ impliquent que $i < y$
– l'opérateur d'élargissement, qui est appliqué au point de contrôle 5 permet d'obtenir que $y \leq j$ ce qui implique finalement $i < j$.
On remarquera que l'abstraction de la garde $j \leq y + 2$ n'apporte pas d'information (*c.f.* section 2.2.2) ce qui fait que les invariants sont identiques aux points de contrôle 5 et 6.

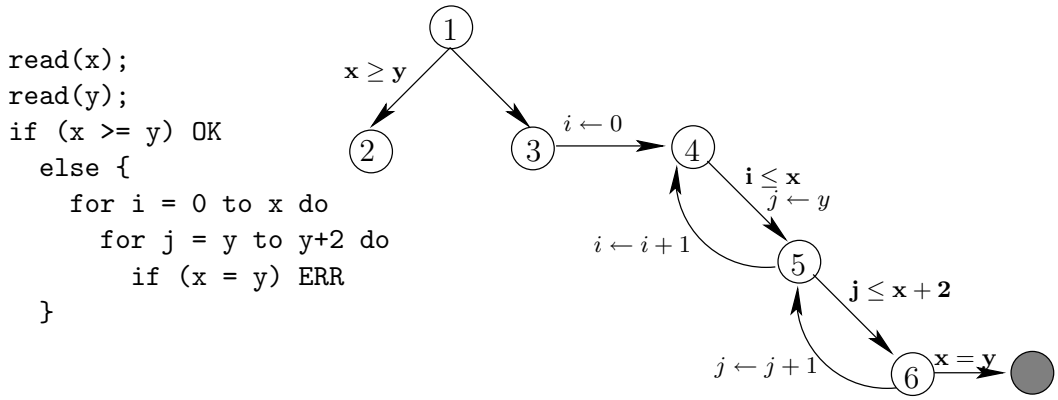
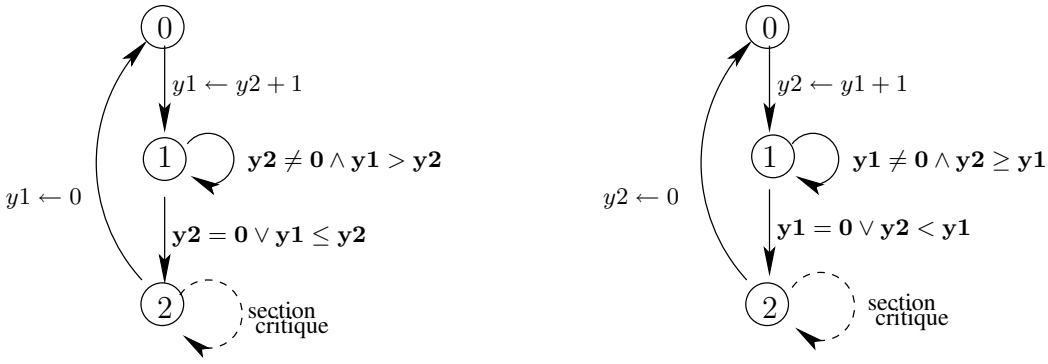


FIG. 3.5 – Exemple 4

	états accessibles
1	$\top$
2	$y \leq x$
3	$x < y$
4	$i \geq 0, x < y$
5,6	$i \geq 0, x \geq 0, x < y, i < y, x < j, i \leq x, y \leq j, i < j$
ERR	$\bot$

Exemple 5
L'algorithme de Bakery permet la synchronisation de deux processus $P1, P2$ par l'intermédiaire de deux variables y_1 et y_2 qui représentent respectivement les priorités des deux processus, de telle façon qu'un seul processus à la fois puisse entrée dans sa *section critique*.



$P1$ $P2$

Nous avons effectué un produit synchronisé des deux automates interprétés modélisant chaque processus pour pouvoir analyser le protocole. On ne détaillera pas le résultat de l'analyse pour chaque point de contrôle mais le point de contrôle 22 est montré inaccéssible ce qui assure que les deux processus ne seront jamais en même temps en section critique.

On notera qu'Antoine Miné expose le même exemple dans [Min01a] et montre la propriété grâce au treillis des octogones.

Cet exemple de vérification de propriétés d'exclusion mutuelle montre que le treillis est particulièrement adapté à ce type d'analyse.

Conclusion et perspectives

Dans la perspective d'analyser une certaine classe de programmes, dont on connaît les caractéristiques précises, nous avons vu qu'il est intéressant de construire des domaines abstraits ad-hoc, qui ont l'avantage d'offrir un bon compromis entre complexité et précision pour l'analyse de leurs propriétés.

Les principales contributions de ce stage dans le domaine de la vérification automatique sont les suivantes :

- La conception d'un nouveau domaine abstrait combinant les intervalles et les propriétés de non-égalité pour une analyse dédiée aux adresses.
- L'implémentation de ce domaine dans le cadre d'un outil de vérification par interprétation abstraite.
- L'utilisation de notre outil dans le cadre d'analyses simples, qui ont montré les intérêts de notre domaine abstrait ; en particulier il était donc judicieux de rajouter des propriétés de non-égalités dans des numériques abstraits.

Ce travail a été l'occasion pour moi de faire une étude en profondeur du domaine théorique de l'interprétation abstraite, et d'avoir une vision d'ensemble des domaines numériques existants, jusqu'aux plus récents. Entre autres, il a fallu dégager les avantages et les inconvénients des différents domaines abstraits en terme de précision et de complexité.

D'autre part, il m'a permis de mesurer la difficulté de la conception d'un domaine abstrait. Les opérations doivent non seulement vérifier les nombreuses propriétés de l'interprétation abstraite, mais elles doivent surtout être pensées en terme de complexité algorithmique.

Perspectives

Les perspectives ouvertes par cette étude sont nombreuses. Ce travail pourra être poursuivi dans les directions suivantes :

- Pour l'instant, les automates interprétés analysés sont entrés selon le format interne de l'analyseur. Nous devons terminer notre compilateur du langage `oc` vers le format interne afin d'analyser des programmes conséquents et vérifier le passage à l'échelle.
  Nos travaux s'insèrent dans le cadre du projet européen Apron (Analyse de PROgrammes Numériques). Ce projet a entre autres pour but l'élaboration d'une interface générique pour les treillis numériques, autour de l'outil Nbac. Après discussion avec Bertrand Jeannet, la non-égalité devrait être prise en compte dans cette interface. Nos travaux ont été utiles à l'amélioration de la généricité de l'analyseur de Nbac et il nous faut continuer dans ce sens.

– Du point de vue algorithmique, la canonisation par un système de réécriture n'est pas efficace. Si l'on voit que nos contraintes sont un affaiblissement des zones, on pourrait proposer une extension de notre domaine aux DBMs, en leur rajoutant une matrice de non-égalité. L'algorithme de canonisation serait une adaptation de l'algorithme de Floyd-Warshall et constituerait une implémentation de notre domaine avec une complexité en $O(n^3)$.
– Il faudra aussi considérer l'utilisation de ce domaine abstrait en parallèle de treillis numériques classiques, pour effectuer des analyses de programmes qui comprennent à la fois des entiers interprétés comme adresses et des entiers « classiques ».
  Enfin, ce treillis pourrait constituer une brique de base pour l'analyse de systèmes concurrents, en particulier pour vérifier des propriétés comme les accès partagés à des variables et les conflits d'accès aux ressources. Les résultats pratiques obtenus sur l'algorithme de Bakery semblent nous encourager vers cette voie de recherche.

Bibliographie

[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 *ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation, PLDI'03*, pages 196–207, San Diego, California, USA, Jun 2003. ACM Press.

[BCM+90] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and J. Hwang. Symbolic model checking : 10^{20} states and beyond. In *Fifth IEEE Symposium on Logic in Computer Science, Philadelphia*, pages 428–439, Jun 1990.

[Bou92] F. Bourdoncle. *Sémantiques des langages impératifs d'ordre supérieur et interprétation abstraite*. PhD thesis, Nov 1992.

[Bou93] F. Bourdoncle. Abstract debugging of higher-order imperative languages. In *ACM SIGPLAN 1993 conference on Programming language design and implementation, PLDI'93*, pages 46–55, New York, 1993. ACM Press.

[BW94] B. Boigelot and P. Wolper. Symbolic verification with periodic sets. In *CAV'94*, Stanford (Ca.), 1994. LNCS 818, Springer Verlag.

[CC76] P. Cousot and R. Cousot. Static determination of dynamic properties of programs. In *2nd International Symposium on Programming*, pages 106–130. Dunod, Paris, France, 1976.

[CC77] P. Cousot and R. Cousot. Abstract interpretation : a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4th ACM Symposium on Principles of Programming Languages, POLP'77*, pages 238–252, Los Angeles, january 1977.

[CC04] R. Clarisó and J. Cortadella. The octahedron abstract domain. In *11th Static Analysis Symposium (SAS)*, volume 3148 of *Lecture Notes in Computer Science*, pages 312–327. Springer-Verlag, aug 2004.

[CH78] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *5th ACM Symposium on Principles of programming languages, POPL'78*, pages 84–96, New York, 1978.

[CLRT90] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Tein. *Introduction to Algorithms*. The MIT Press, Cambridge, Massachusetts, 1990.

[Cou01] P. Cousot. Abstract interpretation based formal methods and future challenges, invited paper. In *"Informatics — 10 Years Back, 10 Years Ahead"*, volume 2000 of *Lecture Notes in Computer Science*, pages 138–156. Springer-Verlag, 2001.

[Dil89] D. L. Dill. Timing assumptions and verification of finite-state concurrent systems. In *Automatic Verification Methods for Finite State Systems, CAV'89*, volume 407 of *Lecture Notes in Computer Science*, pages 197–212. Springer-Verlag, 1989.

[FS00] A. Finkel and G. Sutre. An algorithm constructing the semilinear post* for 2-dim reset/transfer vass. In *25th Int. Symp. Math. Found. Comp. Sci. (MFCS'2000)*, Bratislava, Slovakia, Aug 2000. LNCS 1893, Springer Verlag.

[GJS76] M. R. Garey, D. S. Johnson, and L. J. Stockmeyer. Some simplified np-complete graph problems. *Theor. Comput. Sci.*, 1(3) :237–267, 1976.

[GL93] S. Graf and C. Loiseaux. Program verification and abstraction. In *Joint Conference CAAP/FASE, TAPSOFT'93*. LNCS 668, Springer Verlag, 1993.

[HMPV03] N. Halbwachs, D. Merchat, and C. Parent-Vigouroux. Cartesian factoring of polyhedra in linear relation analysis, jun 2003.

[HS97] W. Harvey and P. J. Stuckey. A unit two variable per inequality integer constraint solver for constraint logic programming. In *Twentieth Australasian Computer Science Conference, ACSC'97*, volume 19, pages 102–111, feb 1997.

[JHR99] B. Jeannet, N. Halbwachs, and P. Raymond. Dynamic partitioning in analyses of numerical properties. In *Static Analysis Symposium, SAS'99*, Venezia, september 1999.

[Kar76] M. Karr. Affine relationships among variables of a program. *Acta Informatica*, 6 :133–151, 1976.

[Lam80] L. Lamport. The "hoare logic" of concurrent programs. In *Acta informatica*, volume 14, pages 21–37, Jun 1980.

[Min01a] A Miné. A new numerical abstract domain based on difference-bound matrices. In *PADO II*, volume 2053 of *LNCS*, pages 155–172. Springer-Verlag, May 2001.

[Min01b] A. Miné. The octagon abstract domain. In *AST 2001 in WCRE 2001*, pages 310–319. IEEE CS Press, October 2001.

[Pra77] V.R. Pratt. Two easy theories whose combination is hard. Technical report, Massachusetts Institute of Technology, Cambridge, sep 1977.

[RCK04] E. Rodríguez-Carbonell and D. Kapur. An abstract interpretation approach for automatic generation of polynomial invariants. In *International Symposium on Static Analysis, SAS'04*, volume 3148 of *Lecture Notes in Computer Science*, pages 280–295. Springer-Verlag, 2004.

[Rus02] V. Rusu. Verification using test generation techniques. In *International Symposium of Formal Methods, FME'02*, pages 252–271, London, 2002. Springer-Verlag.

[RZC02] V. Rusu, E. Zinovieva, and D. Clarke. Verifying invariants more automatically. In *Third International Workshop on Verification and Computational Logic, VCL'02*, 2002.

[SSM05] S. Sankaranarayanan, H. B. Sipma, and Z. Manna. Scalable analysis of linear systems using mathematical programming. In *Proc. of Verification, Model Checking and Abstract Interpretation, VMCAI'05*, volume 3385 of *lncs*, pages 21–47. Springer-Verlag, jan 2005.

Annexes

5.1 Définitions, Lemmes, Théorèmes

5.1.1 Théorie des treillis

Définition 19 (treillis). *Un treillis $(L, \sqsubseteq)$ est un ensemble ordonné tel que deux éléments quelconques de L admettent une borne inférieure (notée $\sqcap$) et une borne supérieure (notée $\sqcup$) dans L.*

Ceci impose l'existence pour $(L, \sqsubseteq)$ d'un plus petit élément (noté $\bot$) et d'un plus grand élément (noté $\top$).

Un treillis sera donc noté $(L, \bot, \top, \sqsubseteq, \sqcup, \sqcap)$

Définition 20 (treillis complet). *Un treillis est dit* complet *si l'existence des bornes inférieure et supérieure est étendue à toute partie non vide de L.*

Les opérateurs de bornes inférieure et supérieure sont étendus aux ensembles.

Définition 21 (continuité sur les treillis). *Une fonction F de $(L, \sqsubseteq, \sqcup, \sqcap)$ dans lui-même est continue* ssi

$$\forall X \subseteq L \quad F(\sqcup X) = \sqcup\{F(x) | x \in X\} \quad et \quad F(\sqcap X) = \sqcap\{F(x) | x \in X\}$$

Définition 22 (profondeur d'un treillis). *On appelle profondeur d'un treillis $(L, \sqsubseteq)$ la longeur maximale des chaînes strictement croissantes de L.*

Si le treillis est de profondeur finie ont dit qu'il satisfait la « propriété de chaînes croissantes ».

Théorème 3 (Kleene). *Soit $(L, \bot, \top, \sqsubseteq, \sqcup, \sqcap)$ un treillis complet et $F : L \to L$ totale et continue. L'équation de point fixe $X = F(X)$ a une plus petite et plus grande solutions[4] données par*

$$lfp(F) = \bigsqcup_{n \geq 0} F^n(\bot) \qquad gfp(F) = \bigsqcap_{n \geq 0} F^n(\top)$$

5.1.2 Réécriture

Définition 23 (confluence). *Une relation binaire $\longrightarrow_R$ sur A est dite confluente si*

$$\forall M, N_1, N_2 \in A \qquad M \xrightarrow{*}_R N_1 \wedge M \xrightarrow{*}_R N_2$$

$$\Rightarrow \exists P \in A \qquad N_1 \xrightarrow{*}_R P \wedge N_2 \xrightarrow{*}_R P$$

où $\xrightarrow{*}_R$ est la fermeture reflexive et transitive de R.

[4] notées *lfp* (*resp. gfp*) pour « least (*resp.* greater) fixed point »

Définition 24 (confluence locale). *Une relation binaire $\longrightarrow_R$ sur A est dite localement confluente si*

$$\forall M, N_1, N_2 \in A \qquad M \longrightarrow_R N_1 \wedge M \longrightarrow_R N_2$$
$$\Rightarrow \exists P \in A \qquad N_1 \xrightarrow{*}_R P \wedge N_2 \xrightarrow{*}_R P$$

Définition 25 (terminaison). *Une relation binaire $\longrightarrow_R$ sur A est terminante, ou noethérienne* ssi *il n'existe pas de suite infinie*

$$a_0 \longrightarrow_R a_1 \longrightarrow_R \ldots \longrightarrow_R a_{n+1} \ldots$$

Lemme 1 (Newman). *Si une relation R termine et est localement confluente alors elle est confluente.*

Définition 26 (paire critique). *Soit une relation binaire $\longrightarrow_R$ sur A. La paire de termes $(M, N) \in A^2, M \neq N$ est une paire critique si*

$$\exists Q \in A \quad Q \longrightarrow_R M \wedge Q \longrightarrow_R N$$

La paire de termes (M, N) est dite joignable si

$$\exists P \in A \qquad M \xrightarrow{*}_R P \wedge N \xrightarrow{*}_R P$$

Théorème 4 (Knuth-Bendix). *Soit un système de réécriture R (relation $\longrightarrow_R$). Alors $\longrightarrow_R$ est localement confluente ssi toutes les paires critiques de R sont joignables.*

5.1.3 Satisfiabilité des systèmes de contraintes

La résolution des systèmes de contraintes de la *seperation theory* ($ax + d \leq by$ avec $a, b \in \{0, 1\}$, $d \in \mathbb{Z}$, [Pra77]) est un problème dans **P**. Il s'avère que les systèmes UTVPI, plus généraux (contraintes de la forme $\pm x \pm y \leq d$), sont également dans **P**. On se reportera à [HS97].

Par contre, la présence de contraintes de non-égalité place le problème dans la classe de complexité **NP** :

Théorème 5. *Le problème de la satisfiabilité d'une solution à un système de contraintes est un problème* **NP***-complet dès lors que des contraintes de non-égalité sont permises.*

Démonstration. Par réduction du problème de la 3-coloration d'un graphe ([GJS76]) au problème de satisfaction.

□

Ressources éléctroniques

http://perso.ens-lyon.fr/mathias.peron/